Migrating Orbix Applications to Orbix 2000 and Orbix 3.3

IONA Technologies
October 2000

M2552

Summary

This migration guide is aimed at customers who have developed and deployed applications based on IONA's Orbix 3 and OrbixWeb 3, and earlier versions of Orbix. This document provides detailed technical guidelines for migrating your CORBA application from the early versions of Orbix and OrbixWeb, which are CORBA 2.1-based, to the Orbix 2000 generation of products, which are CORBA 2.3-based.

There are two main parts in this migration guide:

Part I *Migrating to Orbix 2000*—provides detailed guidelines for migrating Orbix and OrbixWeb to Orbix 2000 (C++ and Java editions), for customers who are ready to migrate to a CORBA 2.3-based system.

Part II *Migrating to Orbix 3.3*—provides detailed guidelines for migrating Orbix and OrbixWeb to Orbix 3.3 (C++ and Java editions), for customers who want to stay with a CORBA 2.1-based system for the moment.

For up-to-date news and advice on migrating to Orbix 2000, see the IONA web site at: `http://www.iona.com/moving_forward`.

Table of Contents

Introduction

Three years ago, IONA undertook an enormous investment in its next-generation product set. The result, announced last October 1999, is the IONA iPortal Suite, a comprehensive platform for building enterprise portals and other large-scale distributed applications. In parallel to the development of Orbix 2000 and the other components of iPortal Suite, IONA has maintained its commitment to the Orbix 3 product family. Orbix 3.3 is the current shipping version of this family. Orbix 3.3 is for those Orbix customers who have deployed applications in the field and who require the latest platform support, fixes to existing product problems and good interoperability between existing applications and Orbix 2000. With both Orbix 2000 and Orbix 3.3 in active use, Orbix 2.3 and OrbixWeb 3.1 are nearing the end of their respective lifecycles. Accordingly, IONA has decided that it will continue support for these products only until March 31st, 2001.

The recommended path for customers upgrading to a new version of Orbix is to move to Orbix 2000. Because Orbix 2000 is a CORBA 2.3-compliant ORB, it offers many new features over previous versions of Orbix:

- Portable interceptor support.
- Codeset negotiation support.
- Value type support.
- Asynchronous method invocation (AMI) support.
- Persistent State Service (PSS) support.
- Dynamic any support.

Orbix 2000 also offers some unique benefits over other commercial ORB implementations, including:

- ORB extensibility using IONA's patented adaptive runtime technology (ART).

  Orbix 2000 has a modular structure built on a micro-kernel architecture. Required ORB modules, *ORB plug-ins*, are specified in a configuration file and loaded at runtime, as the application starts up. The advantage of this approach is that new ORB functionality can be dynamically loaded into an Orbix application without rebuilding the application.

- Improved performance.

  The performance of Orbix 2000 has been optimized, resulting in a performance that is faster than Orbix 3.x and OrbixWeb 3.x in every respect.

- Advanced deployment and configuration.

  Orbix 2000 supports a flexible model for the deployment of distributed applications. Applications can be grouped into configuration domains and organized either as file-based configuration domains or as configuration repository-based configuration domains.

- Rapid application development using the Orbix code generation toolkit.

  The code generation toolkit is an extension to the IDL compiler that generates a working application prototype—based on your application IDL—in a matter of seconds.

Migration Resources

IONA is committed to assisting you with your migration effort to ensure that it proceeds as easily and rapidly as possible. The following resources are currently available:

- IONA's migration web page, including the latest news and links to further resources, is at: `http://www.iona.com/moving_forward`

- This migration guide.

  A technical document providing detailed guidance on converting source code to Orbix 2000 and Orbix 3.3. The document aims to provide comprehensive coverage of migration issues, and to demonstrate how features supported in earlier Orbix versions can be mapped to Orbix 2000 features

- Professional Services migration packages.

  IONA's Professional Services organization has put together a set of consultancy packages that facilitate rapid migration to Orbix 2000 or Orbix 3.3. Details of Professional Services assessment and migration packages are available at: `http://www.iona.com/info/services/ps/migration.htm`

Migration Options

The recommended migration route is to proceed directly to Orbix 2000. The CORBA 2.3 specification, on which Orbix 2000 is based, has been tightened to a degree that standardizes almost every aspect of CORBA programming. Migrating your source code to Orbix 2000 thus represents a valuable investment, because your code will be based on a stable, highly standardized programming interface.

- *Migrating to Orbix 2000*—is recommended in most cases. Orbix 2000 is the latest generation in the family of Orbix products and will be the main focus of development and innovation from now on.
- *Migrating to Orbix 3.3*—is appropriate in some cases, where the effort of migration to Orbix 2000 is not justified. For example, Orbix 3.3 might be an appropriate migration choice for CORBA applications that are approaching the end of their deployed lifespan. IONA's Professional Services division and IONA Customer Support are available to advise you on the most appropriate migration strategy for your system.
- *Mixed Deployment*—is appropriate when a number of CORBA applications are in deployment simultaneously. Some applications might be upgraded to use Orbix 2000 whilst others continue to use Orbix 3.x and OrbixWeb 3.x. This kind of mixed environment requires on-the-wire compatibility between the generation 3 products and Orbix 2000. Extensive testing has been done to ensure interoperability with Orbix 2000.

The following sections summarize the main technical issues that affect each migration path.

Migrating to Orbix 2000

On the client side, the main issue for migration is that the Orbix `_bind()` function is not supported in Orbix 2000. The CORBA Naming Service is now the recommended mechanism for establishing contact with CORBA servers.

On the server side, the basic object adapter (BOA) must be replaced by the portable object adapter (POA). This is one of the major differences between the CORBA 2.1 and the CORBA 2.3 specifications. The POA is much more tightly specified than the old BOA; hence server code based on the POA is well standardized.

Orbix 3.x and OrbixWeb 3.x support a range of proprietary features not covered by the CORBA standard—for example, the Orbix locator, filters, loaders, smart proxies, transformers and I/O callbacks. When migrating to Orbix 2000, the proprietary features must be replaced by standard CORBA 2.3 features. This

migration guide details how each of the proprietary features can be replaced by equivalent Orbix 2000 functionality.

Migrating to Orbix 3.3

Orbix 3.3 is the final release of the IONA's CORBA 2.1-based ORB technology. The Orbix 3.3 product includes both a C++ ORB, formerly Orbix, and a Java ORB, formerly OrbixWeb, in a single package. A number of services are bundled with the Orbix 3.3 product including, the Interface Repository, the CORBA Naming Service, the CORBA Events Service, the CORBA Security Service (OrbixSSL), a DCOM bridge (OrbixCOMet), and an IIOP firewall (Orbix Wonderwall). The CORBA Transaction Service (OrbixOTS) is available separately.

If you choose the migration path to Orbix 3.3, chances are that you will need to deploy Orbix in a mixed Orbix 3.3 / Orbix 2000 environment at some point in the future. Consequently, Orbix 3.3 has been optimized to achieve the best possible degree of on-the-wire interoperability with Orbix 2000.

The main issue for migration to Orbix 3.3 (affecting both Orbix and OrbixWeb legacy code) is that `_bind()` calls must be modified to use the *fully qualified* form of `_bind()`. This is described in detail in the section "Modifications to _bind()" in Part II.

Mixed Deployment

Both Orbix 3.3 and Orbix 2000 have been modified to achieve an optimum level of on-the-wire compatibility between the two products.

Consult the *Orbix 2000 Interoperability Guide* for detailed advice on configuring a mixed deployment. The guide is available from the Orbix 2000 documentation pages: http://www.iona.com/docs/orbix2000/orbix200011.html

Part I
Migrating to Orbix 2000 v1.1

Overview

This part of the migration guide provides technical guidelines on migrating your Orbix or OrbixWeb system to use Orbix 2000. The following topics are discussed:

- IDL Migration.
- Client Migration.
- Server Migration.
- Migrating Proprietary Orbix 3 Features.
- Basic Services.
- Interoperability.
- Administration and Deployment.

IDL Migration

Orbix 2000 supports a number of new IDL data types, notably valuetypes and abstract interfaces. However, the subset of IDL supported by Orbix 3 remains largely unchanged in Orbix 2000. In most cases, legacy IDL can be used in an Orbix 2000 application without making any changes.

A few changes that might affect migration of IDL to Orbix 2000 are described in the following subsections.

The `context` Clause

According to IDL grammar, a context clause can be added to an operation declaration, to specify extra variables that are sent with the operation invocation. For example, the following `Account::deposit()` operation has a context clause:

```
//IDL

interface Account {
    void deposit(in CashAmount amount)
             context("sys_time", "sys_location");
    //...
};
```

The context clause is supported by Orbix 3, but is not supported by Orbix 2000. IDL contexts are generally regarded as type-unsafe and might be removed from a

future revision of the CORBA specification. Orbix clients that use them need to be migrated, to transmit their context information using another mechanism, such as service contexts, or perhaps as normal IDL parameters.

The `opaque` Type

The object-by-value (OBV) specification, introduced in CORBA 2.3 and supported in Orbix 2000, replaces opaques. To ensure rapid migration, replace opaque-based functionality with *custom value types* that allow you to implement your own marshaling rules for values.

The `Principal` Type

The CORBA specification deprecates the `Principal` IDL type; therefore the `Principal` IDL type is not supported by Orbix 2000. However, Orbix 2000 has some limited on-the-wire support for the `Principal` type, to support interoperability with Orbix 3.

Client Migration

Migration of client code from Orbix 3 to Orbix 2000 is generally straightforward, because relatively few changes have been made to the client-side API. Clients that use Orbix-specific features are a special case—they are dealt with in "Migrating Proprietary Orbix 3 Features".

The main issue affecting client migration is that the Orbix `_bind()` function is no longer supported in Orbix 2000 and must be replaced.

Replacing the `_bind()` Function

The `_bind()` function is not supported in Orbix 2000. All calls to `_bind()` must be replaced by one of the following:

- CORBA Naming Service.
- Object-to-string conversion.
- The `ORB::resolve_initial_references()` operation.

CORBA Naming Service

The naming service is the recommended replacement for `_bind()` in most applications. Migration to the naming service is straightforward on the client side. The triplet of (*markerName*, *serverName*, *hostName*), used by the `_bind()` function to locate an object, is replaced by a simple *name* in the naming service.

When using the naming service, an object's name is an abstraction of the object location—the actual location details are stored in the naming service. Object names are resolved using these steps:

1. An initial reference to the naming service is obtained by calling `resolve_initial_references()` with `NameService` as its argument.

2. The client uses the naming service reference to *resolve* the names of CORBA objects, receiving object references in return.

Orbix 2000 supports the CORBA Interoperable Naming Service, which is backward-compatible with the old CORBA Naming Service and adds support for using stringified names.

Object-to-String Conversion

CORBA offers two CORBA-compliant conversion functions:

```
CORBA::ORB::object_to_string()
CORBA::ORB::string_to_object()
```

These functions allow you to convert an object reference to and from the stringified interoperable object reference (stringified IOR) format. These functions enable a CORBA object to be located as follows:

1. A server generates a stringified IOR by calling `CORBA::ORB::object_to_string()`.

2. The server passes the stringified IOR to the client, for example by writing the string to a file.

3. The client reads the stringified IOR from the file and converts it back to an object reference, using `CORBA::ORB::string_to_object()`.

Because they are not scalable, these functions are generally not useful in a large-scale CORBA system. Use them only to build initial prototypes or proof-of-concept applications.

The `ORB::resolve_initial_references()` Operation

The `CORBA::ORB::resolve_initial_references()` operation provides a mechanism for obtaining references to basic CORBA objects, for example the naming service, the interface repository, and so on.

Orbix 2000 allows the `resolve_initial_references()` mechanism to be extended. For example, to access the `BankApplication` service using `resolve_initial_references()`, simply add the following variable to the Orbix 2000 configuration:

```
initial_services:BankApplication = "IOR:010347923849..."
```

Use this mechanism sparingly. The OMG defines the intended behavior of `resolve_initial_references()` and the arguments that can be passed to it. A name that you choose now might later be reserved by the OMG. It is generally better to use the naming service to obtain initial object references for application-level objects.

Callback Objects

Callback objects must live in a POA, like any other CORBA object; hence, there are certain similarities between a server and a client with callbacks. The most sensible POA policies for a POA that manages callback objects are:

Policy Type	Policy Value
Lifespan Policy	`TRANSIENT`
ID Assignment Policy	`SYSTEM_ID`
Servant Retention Policy	`RETAIN`
Request Processing Policy	`USE_ACTIVE_OBJECT_MAP_ONLY`

By choosing a `TRANSIENT` lifespan policy, you remove the need to register the client with an Orbix 2000 locator daemon.

These policies allow for easy management of callback objects and an easy upgrade path. Callback objects offer one of the few cases where the root POA has reasonable policies, provided the client is multi-threaded (as it normally is for callbacks).

IDL-to-C++ Mapping

The definition of the IDL-to-C++ mapping has changed little going from Orbix 3 to Orbix 2000 (apart from some extensions to support valuetypes). The following sections describe the changes that affect legacy Orbix 3 code.

The CORBA::Any Type

In Orbix 2000, it is not necessary to use the type-unsafe interface to `Any`. Recent revisions to the CORBA specification have filled the gaps in the IDL-to-C++ mapping that made these functions necessary. That is, the following functions are deprecated in Orbix 2000:

```
// C++
// CORBA::Any Constructor.
Any(
    CORBA::TypeCode_ptr tc,
    void* value,
    CORBA::Boolean release = 0
);

// CORBA::Any::replace() function.
void replace(
    CORBA::TypeCode_ptr,
    void* value,
    CORBA::Boolean release = 0
);
```

The CORBA::Environment Parameter

The signatures of IDL calls no longer contain the `CORBA::Environment` parameter. This parameter was needed for languages that did not support native exception handling. However, Orbix applications also use it for operation timeouts. Timeout functionality is a quality of service (QoS) policy that is defined in the CORBA 3 Messaging Specification and will be implemented in future releases of Orbix 2000.

System Exception Semantics

Orbix and OrbixWeb clients that catch specific system exceptions may need to change the exceptions they handle when they are migrated to Orbix 2000. Orbix 2000 follows the latest CORBA standards for exception semantics. The two system exceptions most likely to affect existing code are:

When this Happens	Orbix and OrbixWeb Raise	Orbix 2000 Raises
Server object does not exist	`INV_OBJREF`	`OBJECT_NOT_EXIST`
Cannot connect to server	`COMM_FAILURE`	`TRANSIENT`

System exception minor codes are completely different between OrbixWeb 3.2 and Orbix 2000 for Java. Applications which examine minor codes need to be modified to use Orbix 2000 for Java minor codes.

Dynamic Invocation Interface

Orbix-proprietary dynamic invocation interface (DII) functions are not available in Orbix 2000. Code that uses `CORBA::Request::operator<<()` operators and overloads must be changed to use CORBA-compliant DII functions.

Note: Orbix 2000 generated stub code consists of sets of statically generated CORBA-compliant DII calls.

Server Migration

Server code typically requires many more changes than client code. The main issue for server code migration is the changeover from the basic object adapter (BOA) to the portable object adapter (POA).

It is relatively easy to migrate a BOA-based server by putting all objects in a simple POA that uses an active object map; however, this approach is unable to exploit most of the functionality that a POA-based server offers. It is worth while redesigning and rewriting servers so they benefit fully from the POA.

Function Signatures

In Orbix 2000 (C++), two significant changes have been made to C++ function signatures:

- The `CORBA::Environment` parameter has been dropped.
- New types are used for `out` parameters. An `out` parameter of *`T`* type is now passed as a *`T`*`_out` type.

Consequently, when migrating C++ implementation classes you must replace the function signatures that represent IDL operations and attributes.

Object IDs versus Markers

Orbix 2000 uses a sequence of octets to compose an object's ID, while Orbix 3 uses string markers. CORBA provides helper methods `string_to_ObjectId()` and `ObjectId_to_string()` to convert between the two types; hence migration from marker dependencies to Object IDs is straightforward.

CORBA Objects versus Servant Objects

Orbix 2000 introduces the concept of *servant objects*, which are instances of a class that implements an IDL interface.

In Orbix 3 there is no need to distinguish between a CORBA object and a servant object. When you create an instance of an implementation class in Orbix 3, the instance already has a unique identity (represented by a marker) and therefore represents a unique CORBA object.

In Orbix 2000, a distinction is made between the identity of a CORBA object (its object ID) and its implementation (a servant). When you create an instance of an implementation class in Orbix 2000, the instance is a servant object, which has no identity. The identity of the CORBA object (represented by an object ID) must be grafted on to the servant at a later stage, in one of the following ways:

- The servant becomes associated with a unique identity—this makes it a CORBA object, in a similar sense to an object in a BOA-based implementation.
- The servant becomes associated with multiple identities—this case has no parallel in a BOA-based implementation.

The mapping between object IDs and servant objects is controlled by the POA and governed by POA policies.

BOA to POA Migration

The Orbix 3 BOA is replaced by the POA in Orbix 2000. This affects the following areas of CORBA application development:

- Creating an object adapter.
- Defining an implementation class.
- The tie approach.
- Creating and activating a CORBA object.

Creating an Object Adapter

In Orbix 3, a single BOA instance is used. All CORBA objects in a server are implicitly associated with this single BOA instance.

In Orbix 2000, an application can create multiple POA instances (using the `PortableServer::POA::create_POA()` operation in C++ and the `org.omg.PortableServer.create_POA()` operation in Java). Each POA instance can be individually configured, using *POA policies*, to manage CORBA objects in different ways. When migrating to Orbix 2000, you should give careful consideration to the choice of POA policies, to obtain the maximum benefit from the POA's flexibility.

Defining an Implementation Class

The most common approach to implementing an IDL interface in Orbix is to use the inheritance approach. Consider the following IDL fragment:

```
//IDL
module BankSimple {
    Account {
        //...
    };
};
```

The `BankSimple::Account` IDL interface can be implemented by defining a class that inherits from a standard base class. The name of this standard base class for Orbix 3 and Orbix 2000 is shown in Table 1.

Description	Base Class Name
Orbix 3, C++ base class (BOA)	`BankSimple::AccountBOAImpl`
Orbix 2000, C++ base class (POA)	`POA_BankSimple::Account`
Orbix 3, Java base class (BOA)	`BankSimple._AccountImplBase`
Orbix 2000, Java base class (POA)	`BankSimple.AccountPOA`

Table 1 *Standard Base Classes for the Inheritance Approach.*

Consider a legacy Orbix 3 application that implements `BankSimple::Account` in C++ as the `BankSimple_Account_i` class. The `BankSimple_Account_i` class might be declared as follows:

```
// C++
// Orbix 3 Version
// Inheritance Approach
class BankSimple_Account_i : BankSimple::AccountBOAImpl {
public:
    // Declare IDL operation and attribute functions...
};
```

When this implementation class is migrated to Orbix 2000, the `BankSimple::AccountBOAImpl` base class is replaced by the `POA_BankSimple::Account` base class, as follows:

```
// C++
// Orbix 2000 Version
// Inheritance Approach
class BankSimple_Account_i : POA_BankSimple::Account {
public:
    // Declare IDL operation and attribute functions...
};
```

The Tie Approach

The tie approach is an alternative mechanism for implementing IDL interfaces. It allows you to associate an implementation class with an IDL interface using a *delegation approach* rather than an inheritance approach.

In Orbix 2000 (C++) the tie classes are generated using C++ templates. When migrating from Orbix 3 to Orbix 2000, all `DEF_TIE` and `TIE` preprocessor macros must be replaced by the equivalent template syntax.

In Orbix 2000 (Java) the tie approach is essentially the same as in Orbix 3. However, the names of the relevant Java classes and interfaces are different. For example, given an IDL interface, `Foo`, an Orbix 2000 servant class implements the `FooOperations` Java interface and the associated Java tie class is called `FooPOATie`.

Creating and Activating a CORBA Object

To make a CORBA object available to clients, you should:

1. *Create an implementation object*. An implementation object is an instance of the class that implements the operations and attributes of an IDL interface. In Orbix 3, an implementation object is the same thing as a CORBA object. In Orbix 2000, an implementation object is a *servant object*, which is not the same thing as a CORBA object.

2. *Activate the servant object*. Activating a servant object attaches an identity to the object (a marker in Orbix 3 or an object ID in Orbix 2000) and associates the object with a particular object adapter.

In Orbix 3, creating and activating an object are rolled into a single step. For example, in C++ you might instantiate a `BankSimple::Account` CORBA object using the following code:

```
// C++
// Orbix 3
// Create and activate a new 'Account' object.
BankSimple_Account_i * acc1
    = new BankSimple_Account_i("object_id");
```

This step creates the CORBA object and attaches the `object_id` identity to it (initializing the object's marker). The constructor automatically activates the CORBA object.

In Orbix 2000, creating and activating an object are performed as separate steps. For example, in C++ you might instantiate a `BankSimple::Account` CORBA object using the following code:

```
// C++
// Orbix 2000

// persistent_poa - A POA that has already been created
//                  and initialized.

// Step 1:  Create a new 'Account' object.
BankSimple_Account_i * acc1
    = new BankSimple_Account_i();

// Step 2:  Activate the new 'Account' object.
PortableServer::ObjectId_var oid =
   PortableServer::string_to_ObjectId("object_id");
persistent_poa->activate_object_with_id(oid, acc1);
```

Activation is performed as an explicit step in Orbix 2000. The call to `PortableServer::POA::activate_object_with_id()` attaches the `object_id` identity to the object and associates the `persistent_poa` object adapter with the object.

Migrating Proprietary Orbix 3 Features

The Orbix 3 product family provides a rich set of features that extends the core CORBA 2.1 specification and allows you to customize your CORBA application in ways not covered by the standard. Since these proprietary features were first introduced, the CORBA 2.3 standard has evolved to a level of maturity such that the proprietary features are no longer required. Orbix 2000 implements a range of standards-compliant CORBA 2.3 features that replace the proprietary features.

The following proprietary features of Orbix 3 have been removed from Orbix 2000:

- Orbix 3 Locator.
- Filters.
- Loaders.
- Smart Proxies.
- Transformers.
- I/O Callbacks.

If your legacy Orbix 3 application uses one of these proprietary features, you should replace it with a standards-compliant feature of Orbix 2000, as described in the following subsections.

Orbix 3 Locator

The Orbix 3 locator is an Orbix-specific feature that is used in combination with `_bind()` to locate server processes. Because Orbix 2000 does not support `_bind()`, it cannot use the Orbix 3 style locator.

Note: Orbix 2000 has a feature called a locator, which is *not* related in any way to the Orbix 3 locator. The *Orbix 2000 locator* is a daemon process, `itlocator`, that locates server processes for clients.

If your legacy code uses the locator, you must replace it with one of the following Orbix 2000 features:

- The CORBA Naming Service.
- The CORBA Initialization Service.

The CORBA Naming Service

If your legacy code uses the load-balancing feature of the Orbix 3 locator, you can effectively replace this by the `ObjectGroup` feature of the CORBA Naming Service. Object groups are an Orbix-specific extension to the naming service that allow you to register a number of servers under a single name.

Table 2 shows how the Orbix 3 locator maps to the equivalent naming service functionality .

Orbix 3—Locator	**Orbix 2000—Naming Service**
Entry in the locator file, mapping the server name, *`SrvName`*, to a single server host, *`HostName`*: *`SrvName`*`:`*`HostName`*`:`	Object binding in the naming service, mapping a *name* to a single object reference.
Entry in the locator file, mapping the server name, *`SrvName`*, to multiple host names: *`SrvName`*`:`*`Host1,Host2,Host3`*`:`	Object group in the naming service, mapping a *name* to multiple object references.
Overriding functionality of `CORBA::LocatorClass`.	Custom implementation of the `IT_LoadBalancing::ObjectGroup` interface.

Table 2 *Replacing the Orbix 3 Locator by the Naming Service*

The naming service is the preferred way to locate objects in Orbix 2000. It is a standard service and is highly scalable.

The CORBA Initialization Service

The initialization service uses the `CORBA::ORB::resolve_initial_references()` operation to retrieve an object reference from an Orbix 2000 configuration file, *`DomainName`*`.cfg`.

Table 3 shows how the Orbix 3 locator maps to the equivalent initialization service functionality.

Orbix 3—Locator	**Orbix 2000—Initialization Service**

Entry in the locator file, mapping the server name, *`SrvName`*, to a single server host, *`HostName`*:
 	
`SrvName``:`*`HostName`*`:`
	Entry in the *`DomainName`*`.cfg` file, mapping an *`ObjectId`* to a single object reference:
	
	`initial_references:`*`ObjectId`*`:`
	`reference = "IOR:00...";`

Entry in the locator file, mapping the server name, *`SrvName`*, to multiple host names: *`SrvName`*`:`*`Host1,Host2,Host3`*`:`	*No Equivalent*
Override functionality of `CORBA::LocatorClass`.	*No Equivalent*

Table 3 *Replacing the Orbix 3 Locator by the Initialization Service*

The initialization service can only be used as a replacement for the Orbix 3 locator when a simple object lookup is needed.

Filters

Filters are a proprietary Orbix 3 mechanism that allow you to intercept invocation requests on the server and the client side.

Orbix 2000 does not support the filter mechanism. Instead, a variety of Orbix 2000 features replace filters, depending on the purpose for which the filters were used in Orbix 3.

Table 4 summarizes the typical uses of Orbix 3 filters alongside the equivalent features supported by Orbix 2000.

Orbix 3 Filter Feature	**Orbix 2000 Equivalent Feature**
Request logging.	Use portable interceptors.
Piggybacking data on a Request.	Use portable interceptors.
Multi-threaded request processing.	Use a multi-threaded POA and (optionally) a proprietary WorkQueue POA policy.
Accessing the client's TCP/IP details.	*Not supported.*
Security using an authentication filter.	*Full security support will be provided in the Orbix 2000 enterprise edition.*

Table 4 *Orbix 2000 Alternatives to Filter Features*

The following sections discuss how each of the filter features can be implemented in Orbix 2000.

Request Logging

In Orbix 2000, request logging is supported by the new *portable interceptor* feature. Interceptors allow you to access a CORBA request at any stage of the marshaling process, offering greater flexibility than Orbix filters. You can use them to add and examine service contexts. You can also use them to examine the request arguments.

Note: The CORBA Portable Interceptor specification is still undergoing review and might be subject to changes before final ratification.

Piggybacking Data on a Request

In Orbix 3, filters support a feature, *piggybacking*, that enables you to add and remove extra arguments to a request message.

In Orbix 2000, piggybacking is replaced by the CORBA-compliant approach of using *service contexts*. A service context is an optional block of data that can be appended to a request message, as specified in the IIOP 1.1 standard. The content of a service context can be arbitrary and multiple service contexts can be added to a request.

Multi-Threaded Request Processing

In Orbix 3, concurrent request processing is supported using an Orbix *thread filter*. The mechanism is flexible because it gives the developer control over the assignment of requests to threads.

In Orbix 2000, request processing conforms to the CORBA 2.3 specification. Each POA can have its own threading policy:

- `SINGLE_THREAD_MODEL`—ensures that all servant objects in that POA have their functions called in a serial manner. In Orbix 2000, servant code is called only by the main thread, therefore no locking or concurrency-protection mechanisms need to be used.
- `ORB_CTRL_MODEL`—leaves the ORB free to dispatch CORBA invocations to servants in any order and from any thread it chooses.

Because the CORBA 2.3 specification does not specify exactly what happens when the `ORB_CTRL_MODEL` policy is chosen, Orbix 2000 makes some proprietary extensions to the threading model.

The multi-threaded processing of requests is controlled using the Orbix 2000 *work queue* feature. Two kinds of work queue are provided by Orbix 2000:

- *Automatic Work Queue*—a work queue that feeds a thread pool. When a POA uses an automatic work queue, request events are automatically dequeued and processed by threads. The size of the thread pool is configurable.
- *Manual Work Queue*—a work queue that requires the developer to explicitly dequeue and process events.

  Manual work queues give developers greater flexibility when it comes to multi-threaded request processing. For example, prioritized processing of requests could be implemented by assigning high-priority CORBA objects to one POA instance and low-priority CORBA objects to a second POA instance. Given that both POAs are associated with manual work queues, the developer can write threading code that preferentially processes requests from the high-priority POA.

Accessing the Client's TCP/IP Details

Some Orbix 3 applications use Orbix-specific extensions to access socket-level information, such as the caller's IP address, in order to implement proprietary security features. These features are not available in Orbix 2000, because providing access to low-level sockets would considerably restrict the flexibility of CORBA invocation dispatch.

It is recommended that you use an implementation of the security service instead—the security service will be made available with the enterprise edition of Orbix 2000.

Security Using an Authentication Filter

Some Orbix 3 applications use authentication filters to implement security features. In Orbix 2000, it is recommended that you use the security service that will be made available with the enterprise edition of Orbix 2000.

Loaders

The Orbix 3 loader provides support for the automatic saving and restoration of persistent objects. The loader provides a mechanism that loads CORBA objects automatically into memory, triggered in response to incoming invocations.

The Orbix 3 loader is replaced by equivalent features of the Portable Object Adapter (POA) in Orbix 2000. The POA can be combined with a *servant manager* to provide functionality equivalent to the Orbix 3 loader. There are two different kinds of servant manager:

- Servant activator—triggered only when the target CORBA object cannot be found in memory.
- Servant locator—triggered for every invocation.

Taking the `PortableServer::ServantActivator` class as an example, the member functions of `CORBA::LoaderClass` correspond approximately as shown in Table 5.

`CORBA::LoaderClass` Member Function	**`ServantActivator` Member Function**
`save()`	`etherealize()`
`load()`	`incarnate()`
`record()`	*No equivalent function.* An Orbix 2000 object ID (equivalent to an Orbix 3 marker) can be specified at the time a CORBA object is created. This gives sufficient control over object IDs.
`rename()`	*No equivalent function.* An Orbix 2000 object ID (equivalent to an Orbix 3 marker) cannot be changed after a CORBA object has been created.

Table 5 *Comparison of Loader with Servant Activator Class*

A servant locator can also be used to replace the Orbix 3 loader. In general, the servant locator is more flexible than the servant activator and offers greater scope for implementing sophisticated loader algorithms.

Smart Proxies

The Orbix 3 *smart proxies* feature is a proprietary mechanism for overriding the default implementation of the proxy class. This allows applications to intercept outbound client invocations and handle them within the local client process address space, rather than using the default proxy behavior of making a remote invocation on the target object. Smart proxies can be used for such purposes as client-side caching, logging, load-balancing, or fault-tolerance.

Orbix 2000 does not support smart proxies. The primary difficulty is that, in the general case, it is not possible for the client-side ORB to determine if two object references denote the same server object. The CORBA standard restricts the client-side ORB from interpreting the object key or making any assumptions about it. Orbix 3 was able to avoid this limitation by making assumptions about the structure of the object key. This is neither CORBA-compliant nor interoperable with other ORBs.

At best, the ORB can only determine that two object references are equivalent if they have exactly the same server location (host and port in IIOP) and object key. Unfortunately, this may be an unreliable indicator if object references pass through bridges, concentrators, or firewalls that change the server location or object key.

In this case, it is possible for two object references denoting the same CORBA object to appear different to the ORB, and thus have two different smart proxy instances. Since smart proxies are commonly used for caching, having two smart proxy instances for a single CORBA object is unacceptable.

The following sections discuss the approaches you can take to migrate applications that use smart proxies.

Replace Smart Proxies by Equivalent Orbix 2000 Features

Two uses for smart proxies include *logging* and *fault tolerance*. A smart proxy implementation of fault tolerance usually involves caching a standby server's object reference on the client side. Table 6 shows how these smart proxy tasks can be mapped to equivalent features in Orbix 2000.

Orbix 3 Smart Proxy Task	**Orbix 2000 Equivalent Feature**
Logging	Orbix 2000 built-in logging facility or portable interceptors.
Fault Tolerance	Activator-based failover or naming service-based load balancing.

Table 6 *Orbix 2000 Alternatives to Smart Proxy Features*

For logging that requires access to request parameters, portable interceptors can be used in Orbix 2000. Portable interceptors are similar to Orbix 3 filters, but they are more flexible in that they allow you to read request parameters.

Implement Smart Proxy-Like Functionality

In some cases, smart proxy functionality might not map to an equivalent feature in Orbix 2000—for example, a smart proxy that implements client-side caching of data. In cases like this, you have no option but to implement smart proxy-like functionality in Orbix 2000, which can be done as follows:

1. Create a local implementation of the object to be proxified, by writing a class that derives from the client-side stub class.

2. Each time the client receives an object reference of the appropriate type, wrap the object reference with a corresponding smart proxy. Before doing

so, it must determine the target object's identity by making an invocation on the remote target object, asking it for a system-wide unique identifying name. It is this key step that avoids the object identity problem described in "Smart Proxies".

Based on the system-wide unique identifying name, the application can then either create a new smart proxy, or reuse the target object's existing smart proxy. The client application should consistently use the smart proxy in place of the regular proxy throughout the application.

Transformers

Transformers are a deprecated feature of Orbix 3 that allow you to apply customized encryption to CORBA request messages. This could be used to implement a primitive substitute for a security service.

In Orbix 2000, transformers are not supported. It is recommended, instead, that you use the security service that will be made available with the enterprise edition of Orbix 2000.

I/O Callbacks and Connection Management

Orbix 2000 does not allow access to TCP/IP sockets or transport level information. This is incompatible with the Orbix 2000 architecture, which features a pluggable transport layer. You can replace TCP/IP with another transport plug-in such as IP multicast (which is connectionless), simple object access protocol (SOAP), hypertext transfer protocol (HTTP), asynchronous transfer mode (ATM), and so on. For example, the shared memory transport (SIOP) does not use file descriptors or sockets. Because Orbix 2000 has no equivalent to the Orbix `IOCallback` functionality, you must migrate any code that uses it.

The Orbix 3 `IOCallback` functionality is generally used for two main purposes:

- *Connection management*—The number of TCP/IP connections that can be made to a single process is typically subject to an operating system limit. Some form of connection management is required if this limit is likely to be reached in a deployed system.
- *Session management*—The `IOCallback` functionality can be used to implement an elementary session-tracking mechanism. The opening of a connection from a client defines the beginning of a session and the closing of the connection defines the end of the session.

The following subsections discuss how this functionality can be migrated to Orbix 2000.

Connection Management

Orbix 2000 provides an *active connection manager* (ACM) that allows the ORB to reclaim connections automatically, and thereby increases the number of clients that can concurrently use a server beyond the limit of available file descriptors.

IIOP connection management is controlled by four configuration variables:

`plugins:iiop:incoming_connections:hard_limit` sets the maximum number of incoming (server-side) connections allowed to IIOP. IIOP refuses new connections above this limit.

`plugins:iiop:incoming_connections:soft_limit` determines when IIOP starts to close incoming connections.

`plugins:iiop:outgoing_connections:hard_limit` sets the maximum number of outgoing (client-side) connections allowed to IIOP. IIOP refuses new outgoing connections above this limit.

`plugins:iiop:outgoing_connections:soft_limit` determines when IIOP starts to close outgoing connections.

The ORB first tries to close idle connections in least-recently-used order. If there are no idle connections, the ORB closes busy connections in least-recently-opened order.

Active connection management effectively remedies file descriptor limits that has constrained past Orbix applications. If a client is idle for a while and the server ORB reaches its connection limit, it sends a GIOP `CloseConnection` message to the client and closes the connection. Later, the same client can transparently reestablish its connection, to send a request without throwing a CORBA exception.

Note: In Orbix 3, Orbix tended to throw a `COMM_FAILURE` on the first attempt at reconnection; server code that anticipates this exception should be reevaluated against current functionality.

Orbix 2000 is configured to use the largest upper file descriptor limit on each supported operating system (OS). On a UNIX OS, it is typically possible to rebuild the OS kernel to obtain a larger number. However, active connection management should make this unnecessary.

Session Management

Because Orbix 2000 features a pluggable transport layer, it is not appropriate to relate the duration of a client session to the opening and closing of TCP/IP connections from clients. This type of session management, which is typically

implemented using I/O callbacks in Orbix 3, has to be migrated to an alternative model.

Orbix 2000 session management can be implemented using advanced features of the POA. For example, the duration of a client session could be based on a timeout. Clients that are inactive for a certain length of time can have their sessions terminated. This model of session management can be implemented by associating a *servant locator* with a POA, which tracks the length of time for which each session is idle.

Basic Services

Orbix is bundled with some basic services, such as the interface repository and the CORBA Naming Service. Because these service are based mainly on the CORBA standard, there are not many changes between Orbix 3 and Orbix 2000.

Interface Repository

Migrating source code that uses the Interface Repository (IFR) to Orbix 2000 is straightforward. Link the migrated application against the stub code derived from the Orbix 2000 version of the interface repository. No further changes should be necessary.

However, interoperability between Orbix 3 applications and the Orbix 2000 IFR server is not supported. In a mixed Orbix 3 and Orbix 2000 environment, Orbix 3 applications should therefore continue to use an Orbix 3 IFR server. Significant changes were made to the IFR specification in CORBA 2.3, which break interoperability with the older IFR specification, and Orbix 2000 is written to conform to the CORBA 2.3 specification.

CORBA Naming Service

The Orbix 2000 naming service is backward compatible with Orbix 3 in two respects:

- *Source code backward compatibility*—source code that is written to use the standard naming service interfaces can be migrated to Orbix 2000 without modification.
- *On-the-wire backward compatibility*—Orbix 3 applications can interoperate with the Orbix 2000 naming service. If you need to interoperate Orbix 3 applications, it is recommended that you recompile the naming stub code from the Orbix 2000 IDL files.

Orbix 2000 adds a new interface, `CosNaming::NamingContextExt`, which is defined by the CORBA Interoperable Naming Service specification. This interface adds support for using names in stringified format .

The naming service load-balancing extensions provided in Orbix 3 are also present in Orbix 2000. The Orbix 2000 load-balancing interfaces are only slightly different from Orbix 3, requiring small modifications to your source code.

Interoperability

In many cases, it is not practical to migrate all components of a deployed system to Orbix 2000 right away. Orbix 3 and Orbix 2000 applications might need to coexist and interoperate with each other for a period of time. For this reason, Orbix 2000 has been extensively tested to ensure on-the-wire compatibility with Orbix 3 legacy applications.

The minimum versions and patch releases recommended for interoperating Orbix 3 with Orbix 2000 are shown in Table 7.

Product Version and Patch Number
Orbix 3.0.1-20
OrbixWeb 3.2.0-05
Orbix 2.3.5 for OS/390
OrbixCOMet 3.0.1-20

Table 7 *Minimum Orbix 3 Product Versions for Interoperability*

Interoperability with Orbix 2000 is not supported for Orbix versions lower than those shown in Table 7. IONA Product Support can advise you on the optimum Orbix version to use in your mixed Orbix 3 and Orbix 2000 environment.

The Orbix 2000 documentation set includes an *Interoperability Guide* that provides essential advice on how to configure Orbix 3 and Orbix 2000 to interoperate with each other.

The Orbix Protocol (POOP)

Orbix 2000 only supports CORBA-compliant transport protocols such as IIOP. If you have older (pre-Orbix 2.3) systems that rely on POOP, or code that calls `CORBA::Orbix.bindUsingIIOP(0)`, you must change it to use IIOP. Otherwise, the Orbix client cannot invoke on any Orbix 2000 component.

Administration and Deployment

The administration of Orbix 2000 has changed significantly from Orbix 3. The major differences are:

- The Orbix 3 daemon, `orbixd`, is replaced by two Orbix 2000 daemons, the *locator daemon* and the *activator daemon*.

  The locator daemon helps clients to find Orbix 2000 servers.

  The activator daemon launches dormant Orbix 2000 servers in response to a client's request for service.

- The locator daemon locates a CORBA object based on the POA name embedded in an object reference. Hence, POA names play an important role in the configuration of the locator daemon.

  This contrasts with the behavior of the Orbix 3 daemon, which locates a CORBA object based on the server name embedded in an object reference.

- Orbix 2000 introduces the concept of configuration domains and provides the option of configuring a deployed system using either a central configuration server or a file-based configuration.

- The Orbix 2000 command-line administration tools have been unified under a single tool, `itadmin`.

General Command-Line Tools

Table 1 compares the Orbix 3.3 general purpose command-line tools with the Orbix 2000 tools.

Description	Orbix 3.3	Orbix 2000
Show implementation repository (IMR) entry.	`catit`	`itadmin process show`
Security commands.	`chownit` `chmodit`	*No equivalent*
Show configuration	`dumpconfig`	`itadmin config dump`

Description	Orbix 3.3	Orbix 2000
Associate hosts into groups	`grouphosts`	*No equivalent*
C++ IDL compiler	`idl`	`idl`
CodeGen toolkit	`idlgen`	`idlgen`
Java IDL compiler	`idlj`	`idl`
Interface Repository	`ifr`	`itifr`
Kill a server process	`killit`	`itadmin process stop`
List servers	`lsit`	`itadmin locator list`
Create sub-directory in IMR	`mkdirit`	*No equivalent*
Orbix Daemon	`orbixd`	`itlocator/itactivator`
Ping servers	`pingit`	*No equivalent*
List active servers	`psit`	`itadmin process list`
Add definition to IFR	`putidl`	`idl -R`
Register server in IMR	`putit`	`itadmin process create`
Show IFR definition	`readifr`	`itadmin ifr show`
Remove sub-directory from IMR	`rmdirit`	*No equivalent*
Unregister server from IMR	`rmit`	`itadmin process remove`

Description	Orbix 3.3	Orbix 2000
Remove definition from IFR	`rmidl`	`itadmin ifr remove`
Associate servers with groups	`servergroups`	*No equivalent*
Associate hosts with servers	`serverhosts`	*No equivalent*

Table 8 *Comparison of Orbix 3.3 and Orbix 2000 General Command-Line Tools*

Naming Service Command Line Tools

Table 9 compares the Orbix 3.3 naming service command-line tools with the Orbix 2000 tools.

Description	Orbix 3.3	Orbix 2000
Add a member to an object group	`add_member`	`itadmin nsog add_member`
Print IOR of an object group	`cat_group`	*No equivalent*
Print IOR of an object group member	`cat_member`	`itadmin nsog show_member`
Print IOR of given name	`catns`	`itadmin ns resolve`
Remove an object group	`del_group`	`itadmin nsog remove`
Remove a member from an object group	`del_member`	`itadmin nsog remove_member`
List all groups	`list_groups`	`itadmin nsog list`

Description	Orbix 3.3	Orbix 2000
List the members of an object group	`list_members`	`itadmin nsog list_member`
List bindings in a context	`lsns`	`itadmin ns list`
Create an object group	`new_group`	`itadmin nsog create`
Create an unbound context	`newncns`	`itadmin ns newnc`
Select a member of an object group	`pick_member`	*No equivalent*
Bind a name to a context	`putncns`	`itadmin ns bind -context`
Create a bound context	`putnewncns`	`itadmin ns newnc`
Bind a name to an object	`putns`	`itadmin ns bind -object`
Rebind a name to a context	`reputncns`	*No equivalent*
Rebind a name to an object	`reputns`	*No equivalent*
Remove a binding	`rmns`	`itadmin ns remove`

Table 9 *Comparison of Orbix 3.3 and Orbix 2000 Naming Command-Line Tools*

Activation Modes

BOA activation modes, *shared*, *unshared*, *per-method*, *per-client-pid* and *persistent*, are used for a variety of reasons—for example, to achieve multi-threaded behavior in a single-threaded environment, to increase server reliability, and so on. The two most popular modes are:

- *Shared mode*—which enables all clients to communicate with the same server process.

- *Per-client-pid mode*—which enforces a 1-1 relationship between client process and server process, is sometimes used to maximize server availability.

The POA provides three shared activation modes: *always*, *on-demand* and *never*. Migration of source code should be straightforward, because the choice of activation mode has almost no impact on BOA or POA-based server code.

The additional activation modes provided by Orbix 3 (not shared mode) are typically used to achieve some form of load-balancing that is transparent to the client. The Enterprise version of Orbix 2000, to be released in the near future, will include transparent locator-based load balancing over a group of replica POAs. This will answer the needs currently addressed by Orbix 3 activation modes.

Part II
Migrating to Orbix 3.3

Overview

Issues for migrating from Orbix 3.0 to Orbix 3.3 can be broken down into the following categories:

1. APIs and features in Orbix 3.0 that have changed or been eliminated in Orbix 3.3.

2. Considerations for interoperating with Orbix 2000, especially in a heterogeneous environment involving a mixture of Orbix 3.0, 3.3, and 2000 clients and servers.

Note: Most of the improvements made to Orbix 3.3 to improve interoperability with Orbix 2000 (item 2 above), were retrofitted into Orbix 3.0.1 at patch 20.

Many necessary changes are flagged as compile-time errors. It is relatively easy to find and correct these kinds of error and make the function calls conform to the new APIs. A few items have to be searched for manually, because they do not generate compile-time errors. For each of the changed items, the following sections indicate whether the compiler detects the API change or whether a manual search is required.

APIs and Features that have been Removed or Changed

Support for some long-deprecated functions and features has been dropped from Orbix 3.3. Other features have been changed to improve compatibility and interoperability with Orbix 2000.

Modifications to _bind() / bind() in C++ and Java

The `_bind` function is generated for each class representing an IDL interface. `_bind()` / `bind()` provides a primitive, non-CORBA-compliant, way to connect an initial client proxies with a server object. However, the preferred, CORBA-compliant approaches for establishing initial client proxies are to use `resolve_initial_references()`, the Naming Service, Trader Service, or an application-level factory/finder interface. The `_bind()` function has been deprecated for some time.

In Orbix 3.3, `_bind` has been significantly changed, but not altogether eliminated. `_bind` had various complex and esoteric forms, which have been eliminated. The most common and straightforward use of `_bind`, called *fully qualified* `_bind`, can still be used. For functionality beyond fully qualified `_bind`, you need to change code to use fully qualified `_bind` or one of the preferred alternative approaches.

The `_bind` function takes three arguments, and implies a fourth. For example:

```
CORBA::Object_var a = Account::_bind("M:S", "H");
```

supplies the following four pieces of data to the ORB runtime:

- `Account` is the name of the target object interface, or parent interface. The result can be stored in an `Account_var` or a base class pointer of `Account` (for example `Object_var`).
- `M` is the marker of the target object.
- `S` is the name of the CORBA server in which to look for the target object.
- `H` is the host name of the machine on which to look for the CORBA server.

Unsupported Forms of _bind() / bind()

In Orbix 3.0, some `_bind` arguments could be omitted, leading to various `_bind` modes. All of these partially qualified `_bind` modes have been eliminated.

- *Polymorphic bind*—The interface name could be any base interface of the target object.
- *Anonymous bind*—The marker could be omitted, implying that the ORB could choose any object that satisfied the other criteria.
- *Implied server bind*—The server name could be omitted, implying that the ORB would use the interface name as the server name (`Account` in this case).
- *Locator bind*—The host name could be omitted, implying that the ORB would use the Locator to examine the host or hostgroup files (or a user algorithm) to determine the host the server might be running on.

The above forms of `_bind` could even be combined, such as anonymous polymorphic `_bind`, or anonymous locator `_bind`, and so on.

Fully Qualified _bind() / bind()

Fully qualified `_bind` is the only mode supported in Orbix 3.3. It requires the following:

1. The interface name is exactly the most derived interface of the target object.
2. The marker is specified for the target object.
3. The server is specified for the target object.
4. The host is specified for the target object.

With fully qualified `_bind`, it is generally necessary to set the marker explicitly. Look for where objects are instantiated in the server, with either the BOA or TIE approach, and confirm that a marker string is supplied to the constructor or used in a call to `_marker()`.

It is important to understand what functionality occurs on the client and what functionality occurs on the server:

- The marker, server, and host parameters are verified on the client. If an Orbix 3.3 client calls `_bind` without a marker, server or host, `_bind` throws a `CORBA::BAD_PARAM` System Exception.
- The interface type is verified on the server. If the interface is not specified correctly, an Orbix 3.3 server throws a `CORBA::INV_OBJREF` SystemException.

These semantics have an impact on mixed environments, where Orbix 3.0 / OrbixWeb 3.2 (or earlier) interoperates with Orbix 3.3. When interoperating with an Orbix 3.3 server, an Orbix 3.0 / OrbixWeb 3.2 (or earlier) client:

1. Can use fully qualified bind, without modifying the client.
2. Can omit the host name and server name without modifying the client, because these parameters are resolved on the client.
3. Can omit the marker (anonymous bind) without modifying the client, provided the `Orbix.ENABLE_ANON_BIND_SUPPORT` environment variable is set to `TRUE` on the server (default is `TRUE`). Setting this environment variable to `FALSE` improves the Orbix 3.3 speed.

   The anonymous bind is the most common of the esoteric bind modes. This switchable backwards compatibility eases migration while allowing you, eventually , to take advantage of the Orbix 3.3 performance improvements related to fully qualified `_bind`.
4. Cannot use polymorphic bind. The Orbix 3.3 server would return a `CORBA::INV_OBJREF` system exception in this case. This case requires the client to be modified.

If `CORBA::ORB::collocated` set to `TRUE`, the fully qualified bind requirements are reduced to specifying only the marker and exact interface. Because an in-process object lookup is going to be performed, the host is ignored and the server name can be omitted. Alternatively, if the server name is present, it must be the server name associated with the current server process.

C++ Function Signatures for _bind()

The IDL compiler generates a set of overloaded `_bind` functions to handle the various forms of `_bind`. Some of these have changed because they are no longer needed:

`_bind(const char* markerServer, const char* host, const CORBA::Context& ctx)` remains unchanged.

`_bind()` removed.

`_bind(const char* markerServer = 0, const char* host = 0)` changed to the following (no longer has default args)

```
_bind(const char* markerServer, const char* host)
```

When explicitly binding to the Orbix daemon, `orbixd`, use a `0` (zero) marker value:

```
IT_daemon::_bind("0:IT_daemon", host);
```

When explicitly binding to the IFR, use a marker of the IDL type for the repository object (all IFR object markers are IFR type names):

```
Repository::_bind("IDL\\:iona.com/Repository:IFR", host);
```

Java Method Signatures for bind()

The `idlj` compiler also generates a set of overloaded `bind()` methods to handle the various bind forms. The following have been removed and the remaining `bind()` calls remain the same.

`bind ()` removed.

`bind(org.omg.CORBA.ORB orb)` removed.

Locator

The Orbix locator is a non-CORBA-compliant feature that resolves the host name in `_bind` when no host name is explicitly provided. This functionality is no longer needed with fully qualified `_bind`. The Orbix Locator is actually a set of features, which have changed as follows:

The CORBA::LocatorClass

The `CORBA::LocatorClass` has been removed because it is not needed, now that bind is fully qualified. If you have legacy code that uses a `CORBA::LocatorClass` to provide host resolution logic, you should move that logic to an independent class, and invoke the behavior to resolve the host name before calling `_bind`. Alternatively, you can use configuration variables to specify the host or, preferably, the IOR, of well-known server objects:

```
IT_<ServiceName>_HOST = ". . .";
Common.Services.ServiceName = "IOR: . . .";
```

These objects are accessed through the CORBA-compliant `CORBA::ORB::resolve_initial_references()` function, instead of `_bind`.

The Locator Files and Associated Utilities

The files associated with the locator, `Orbix.hst` and `Orbix.grp`, are of limited usefulness in Orbix 3.3 because they are no longer used by `_bind()`. However, these files are still accessible through operations defined on the `IT_daemon` IDL interface—for example `lookUp()`, `addHostsToServer()`, `addHostsToGroup()` and so on.

The utilities that edit the locator files, `serverhosts`, `servergroups`, `grouphosts`, `lhosts`, are no longer provided with Orbix 3.3. The `Orbix.hst` and `Orbix.grp` files can be edited using a regular text editor instead.

It is no longer meaningful to have an `IT_daemon` entry in the locator files.

Non-Native C++ Exceptions

Only native C++ exceptions are supported. This means that the TRY/CATCH macros are no longer supported and exceptions are not raised via the `CORBA::Environment` variable argument. However, the `CORBA::Environment` variable can still be used as the mechanism to pass a per-call timeout value, if such functionality is needed.

You should search your client and server source code for `TRY/CATCH` macros and convert it to use C++ `try/catch`. Code with `TRY/CATCH` macros will no longer compile. The following example shows a code fragment before and after being migrated to use `try/catch`.

Example 1

Consider the following original code, which uses the `TRY/CATCH` macros to handle an exception raised by an `Account::withdraw()` operation:

Original Code

```
Account_var a = . . .;
TRY
{
    a->withdraw(100.00, IT_X);
}
CATCH(Bank::InsufficientFunds, e)
{
    cout << "insufficient funds" << endl;
}
ENDTRY
```

The `TRY/CATCH` macros declare and use a variable named `IT_X`, which is used to propagate exception information.

Compare the original code with the following revised code, which has been modified to use the native C++ `try/catch`:

Revised Code

```
Account_var a = . . .;
try
{
    a->withdraw(100.00);
}
catch (const Bank::InsufficientFunds& e)
{
    cout << "insufficient funds" << endl;
}
```

Example 2

Another possibility is that the client exception-handling code is written directly, using the CORBA::Environment variable without the TRY/CATCH macros. This typically means the exception handling logic is an if-block, testing the CORBA::Environment variable. For example:

Original Code

```
Account_var a = . . .;
CORBA::Environment e;

a->withdraw(100.00, e);
if (e.is_exception("Bank::InsufficientFunds"))
{
    cout << "insufficient funds" << endl;
}
```

Compare this with the following revised code, which has been modified to use the native C++ try/catch:

Revised Code

```
Account_var a = . . .;

try
{
    a->withdraw(100.00);
}
catch (Bank::InsufficientFunds& e)
{
    cout << "insufficient funds" << endl;
}
```

The second example (using CORBA::Environment) is not as easy to search for as the first example (using TRY/CATCH macros), because there are no TRY/CATCH macros to search for. It is best also to search for explicit usages of CORBA::Environment and is_exception.

Note: The second example still compiles in its original form. It is valid to declare and use CORBA::Environment, but it cannot be used for exceptions.

The impact of not changing the code in the second example can be severe. If the call to withdraw() raises an exception, in the original code the C++ runtime looks for the nearest enclosing try/catch block and does not consider the subsequent if statement.

Throwing Exceptions

Throwing an exception within a server is not done by setting the `CORBA::Environment` variable, but using a `C++ throw`.

Exception Handling in Filters

The `CORBA::Environment` variable can still be used to test and set exceptions within filters, as in Orbix 3.0.

CORBA::ORB::useNativeExceptions

Because only native C++ exceptions are supported, the following functions have been removed:

```
CORBA::Boolean CORBA::ORB::nativeExceptions()
CORBA::Boolean CORBA::ORB::nativeExceptions(Boolean)
```

Code that formerly depended on these functions can assume that `nativeExceptions()` always returns TRUE. For example:

Original Code

```
if (CORBA::Orbix.nativeExceptions())
{
    ... //code for native exceptions being TRUE
}
else
{
    ... //code for native exceptions being FALSE
}
```

Revised Code

```
    ... //code for native exceptions being TRUE
```

Class CORBA::NatExcResetter

Because only native C++ exceptions are supported, the `CORBA::NatExcResetter()` class has been removed, as it is no longer meaningful. Code that uses the `CORBA::NatExcResetter()` class should be deleted.

CORBA::Object Class

The following functions have been removed from `CORBA::Object`, as they are no longer meaningful. `CORBA::Object` is the base class of all generated classes for IDL interfaces. The following were never documented APIs, so, in general, should not have been used by Orbix developers. Any code that uses the following functions, should be deleted.

- `void CORBA::Object::_restate()`
- `void CORBA::Object::_marshall()`
- `void CORBA::Object::_unmarshall()`
- `void CORBA::Object::_fixOnAccess()`
- `CORBA::PPTR* CORBA::Object:_makeDummyPptr()`
- Enumeration `CORBA::Object::OBJECT_STATE`
- `CORBA::Object::Object(const Object*)`

Note: `CORBA::Object::Object(const CORBA::Object&)` is still available. However, `CORBA::Object::operator=(const CORBA::Object&)` is not available.

Thread Model

The internal thread model has been updated in Orbix 3.3. This has no direct impact on application-level threads, but has some implications for Orbix configuration.

- *New internal thread model*—replaces the old internal thread model for monitoring the network.
- *New thread API*—controls the number of threads and file descriptors (FDs) used for network connections.
- *Functions dropped from the old thread API*—functions associated with the old internal thread model are no longer supported.
- *New `IOCallback` functions*—warn when a process is running low on FDs or has reached a hard FD limit (all FDs used).
- *Lock model*—The `mt.h` and *`ThreadArch`*`.cxx` source files are no longer supplied with Orbix 3.3.

New Internal Thread Model

The internal thread model for Orbix has been re-designed. This has no effect on the application level thread model that the user interacts with via the `CORBA::ThreadFilter` class. All `ThreadFilter` models, such as per-request, per-object, per-client, and so on, are still usable. The internal thread model is used by Orbix to listen for network connections from clients and to read and write network messages on established connections. One visible advantage of the new thread model is that it is easier for you to configure.

When you run an Orbix application, Orbix starts a number of internal threads in a thread pool. These threads work together to listen for incoming connection attempts from clients and read requests from the network. Ultimately, requests are processed by an application thread, using a thread model written by the user.

The internal network threads use a *leader-follower* design. This means that one thread in the pool is blocked on a call to the low-level TCP/IP `select()`, and when activity occurs, this thread processes it. Simultaneously another thread is dispatched from the pool to perform another low-level TCP/IP `select()`. When a thread completes its current task, it is returned to the pool.

New Thread API

The size of the internal network thread pool is controlled by the `IT_DEF_NUM_NW_THREADS` configuration parameter. The default value is `1`. The user can change this default if a larger initial internal network thread pool is needed.

The following new function can be used to control the number of threads in the internal thread pool:

```
CORBA::Boolean CORBA::ORB::add_nw_threads(
    CORBA::ULong num_threads
)
```

The `add_nw_threads()` function can be used to increase the number of threads in the internal network thread pool at any time. The `num_threads` parameter specifies the number of threads to add to the thread pool—the size of the thread pool can only be increased, not reduced.

The default thread pool size, `1`, is the best setting for most applications. A network thread is responsible for only a little bit of work, which consists of reading the

TCP/IP buffer and depositing the message on an event queue for processing by an application thread.[1]

In general terms, the number of network threads should only be increased if both of the following conditions hold:

1. There are lots of simultaneous requests/replies to a process.
2. A single network thread has insufficient capacity to service the TCP/IP buffers.

Functions Dropped from the Old Thread API

The following API functions associated with the old thread model have been removed:

- `void CORBA::ORB::maxConnectionThreads(CORBA::ULong max)`
- `CORBA::ULong CORBA::ORB::maxConnectionThreads() const`
- `void CORBA::ORB::maxFDsPerConnectionThread(`

  `CORBA::ULong max`

  `)`
- `CORBA::ULong CORBA::ORB::maxFDsPerConnectionThread() const`

New IOCallback Functions

As Orbix opens and closes connections, it consumes file descriptors. File descriptions (FDs) are process-level resources, and are also used for non-Orbix activities, such as file I/O, database access, and so on. The number of FDs in a

---

[1] The network thread is also responsible for re-combining any IIOP-fragment messages (that is, what the network thread hands off is a complete IIOP message). However, IIOP fragments are rarely used—in particular, in Orbix 3, they are never generated (Orbix 3 has the ability to process IIOP fragments but not generate them). It is also important to note that unmarshalling the IIOP message occurs in the application thread, after the hand off from the network thread. So network threads do very little work. The cost of additional network thread on a single-processor machine is a context switch (which is relatively expensive); on multi-processor machines, the network threads could be distributed, across the processors. Of course, with any system, increasing the number of threads is not a guaranteed increase in performance, and depends on hardware and operating system.

process is a limited resource, and in general Orbix cannot assume that all available FDs can be used by Orbix (for example, some may need to be reserved for database activity).

Orbix allows a client or server to receive a callback for certain connection and file descriptor (FD) events. Callbacks exist for opening and closing a connection to another Orbix program. For the new thread model, additional callbacks have been developed to allow the user to monitor the consumption of FDs. The user can specify both *soft* and *hard* limits on the number of FDs Orbix can use.

To receive the new callbacks, define a class that inherits from the Orbix `CORBA::IT_IOCallback` class. The `CORBA::IT_IOCallback` class has been extended with three new callback events that allow the user to monitor the consumption of FDs:

```
// C++
class IT_IOCallback
{
  public:
    ...
    // The following functions are called when the number
    // of FDs used by Orbix hits a soft or hard limit set
    // by the user.

    // The low-watermark (soft limit) has been reached
    virtual void AtOrbixFDLowLimit(int numFDsUsed);

    // The hard limit has been reached.
    // This implies that Orbix is no longer listening for
    // new connections (which would consume another FD).
    virtual void StopListeningAtFDHigh(int numFDsUsed);

    // Orbix has resumed listening after the number of FDs
    // has gone below the hard limit.
    virtual void ResumeListeningBelowFDHigh(
        int numFDsUsed
    );
};
```

The `AtOrbixFDLowLimit()`, `StopListeningAtFDHigh()`, and `ResumeListeningBelowFDHigh()` functions, combined with new configuration variables `IT_FD_WARNING_NUMBER` and `IT_FD_STOP_LISTENING_POINT`, give users flexibility to monitor consumption of FDs:

- When the number of Orbix FDs reaches `IT_FD_WARNING_NUMBER`, either on the way up or the way down, `AtOrbixFDLowLimit()` is called.
- When the number of Orbix FDs reaches `IT_FD_STOP_LISTENING_POINT`, `StopListeningAtFDHigh()` is called.
- When an Orbix FD is freed up or the number of FDs made available to Orbix is increased, `ResumeListeningBelowFDHigh()` is called.

Lock Model

The internal lock model has been changed to use the Orbix 2000 lock classes. The `mt.h` and *`ThreadArch`*`.cxx` files are no longer supplied with Orbix 3.3. Legacy code that uses the classes in `mt.h` must use the previous versions of these files (and consider it application code, not IONA code), or change the code to use a different mechanism.

CORBA::ORB::defaultTxTimeout

The single `CORBA::ORB::defaultTxTimeout()` function has been replaced by two functions. Originally, the function signature was:

```
// C++
// Original 'defaultTxTimeout()' signature
CORBA::ULong
CORBA::ORB::defaultTxTimeout(
    CORBA::ULong val = CORBA::INIFINITE_TIMEOUT,
    CORBA::Environment& env = CORBA::IT_chooseDefaultEnv
);
```

This has been replaced by two functions, one that is an accessor and one that is a mutator:

```
// C++
// Accessor function
CORBA::Ulong
CORBA::ORB::defaultTxTimeout();

// Mutator function
CORBA::ULong
CORBA::ORB::defaultTxTimeout(
    CORBA::ULong val,
    CORBA::Environment& env = CORBA::IT_chooseDefaultEnv
);
```

The orginal accessor-like functionality would also mutate the timeout to `CORBA::INFINITE_TIMEOUT`, for example:

```
CORBA::ULong t = CORBA::Orbix.defaultTxTimeout();
```

Accessing the value and changing it are now clearly separated. It is unlikely that this change affects any client code, but you should verify this by searching for all calls to `defaultTxTimeout()`.

CORBA::Environment Class

Changes have been made to the `CORBA::Environment` class that affect some data members and member functions.

Accessing Data Members

Data members of the `CORBA::Environment` class that used to be public have been made private. The data members are now accessed using accessor/mutator function pairs. For example, the `m_request` data member:

```
// C++
CORBA::Request* CORBA::Environment::m_request
```

is now accessed using the following functions:

```
// C++
CORBA::Request* CORBA::Environment::request();
void CORBA::Environment::request(CORBA::Request*);
```

The `m_timeout` data member:

```
// C++
CORBA::ULong CORBA::Environment::m_timeout
```

is now accessed using the following functions:

```
// C++
CORBA::ULong CORBA::Environment::timeout() const;
void CORBA::Environment::timeout(CORBA::ULong val);
```

Attempting to access the `m_request` or `m_timeout` member variables directly generates compiler errors. Your code should be changed to use the accessor/mutator functions instead.

Member Functions Removed

Three `CORBA::Environment` functions, which were only needed to support the TRY/CATCH macros, have been removed:

```
// C++
void CORBA::Environment::propagate()
void CORBA::Environment::acknowledge()
CORBA::Boolean CORBA::Environment::uncaught()
```

The following function has been removed:

```
// C++
void Request::mk_arg(CORBA::TypeCode_ptr, void*)
```

It was supplied only on NT, and was redundant. This should not affect your code.

CORBA::CollocateResetter Class

The default `CORBA::Environment` parameter in the `CollocateResetter` constructor has been removed. The function signature is now:

```
// C++
CORBA::CollocateResetter::CollocateResetter(Boolean
tmpSetting)
```

This is unlikely to affect your code. Any occurrences will be flagged as compiler errors, which can be easily fixed by removing the `CORBA::Environment` argument passed to the constructor.

Fixed Data Type

Orbix 3.0 and 3.3 support the IDL fixed data type. This data type maps to a C++ class. The function signatures for the Orbix 3.0 fixed data type class conform to the original OMG specification, but it turns out there were errors in the specification. Orbix 3.3 corrects these errors by changing the function signatures.

The changes primarily concern the use of references in return types. For example, the original specification uses:

```
// C++
template<unsigned short d, short s>
class CORBA_Fixed<d, s>
{
    public:
    template<unsigned short d, short s>
    CORBA_Fixed<d, s> operator= (const CORBA_Fixed<d, s>&
val);
};
```

which defines `operator=()` with the wrong return type. A basic assignment would work, but complex (rarely coded) expressions would potentially fail. For example, consider the following assignment statement:

```
// C++
CORBA_Fixed<d, s> x = 0;
CORBA_Fixed<d, s> y = 0;

(x = y)++;  //expect x equal to 1, y equal to 0;
            //in reality x would be 0, and y would be 0.
```

The expression fails, because the assignment return value is a new (temporary) instance of `CORBA_Fixed<d, s>`, instead of a `CORBA_Fixed<d, s>&` reference to the left-hand side, `x`, of the expression.

The `operator=()` assignment operator should have the following signature:

```
// C++
template<unsigned short d, short s>
CORBA_Fixed<d, s>& operator=(
    const CORBA_Fixed<d, s>& val
);
```

The implementation of the fixed data type class now throws a `CORBA::DATA_CONVERSION` system exception whenever the attempted operation would exceed the bounds described by the IDL fixed data type.

Processing CORBA.h

The CORBA.h header aggregates the various CORBA header files. The included class declarations have been segmented into more files, resulting in a greater number of files, although the total number of declarations in those files has decreased. The increased segmentation should help you to locate specific header files more easily (for example, when confirming an API signature).

Access to the runtime API is provided by a single `#include <CORBA.h>` line, as before—no changes are needed as a result of this reorganization.

In Orbix 3.0, users have to `#define EXCEPTIONS` to use the full range of CORBA system exceptions. This is no longer necessary in Orbix 3.3. Continuing to `#define EXCEPTIONS` does no harm (code still compiles and runs), but it is superfluous.

In Orbix 3.0, users have to `#define WANT_ORBIX_FDS` to use the full range of APIs for Orbix internal file descriptors. This is no longer necessary in Orbix 3.3. Continuing to `#define WANT_ORBIX_FDS` does no harm (code still compiles and runs), but it is superfluous.

Some operating system header files conflict with CORBA.h. This problem occurred in Orbix 3.0 as well, but in Orbix 3.3 the user has more control over the mechanism for resolving the conflict. For example, some operating systems headers have a line, `#define minor`, in them. But `minor` is used as a function name on the exception class. Since the preprocessor makes the macro substitution first, this creates an error in the code. Orbix 3.0 would `#undef` the conflicting macros. This is still done in Orbix 3.3. However, all of the `#undef`s have been grouped together in `CORBA.h`, and are controlled by an `ORBIX_DONT_UNDEF` macro, for example:

```
// In CORBA.h
#ifndef ORBIX_DONT_UNDEF
#undef minor
. . .
#endif
```

Nothing needs to be done if you want the standard symbols to be `#undef`'ed as they always have been. However, if you want more control over this (for example, to `#undef` the symbols yourself, and then re-`#define` them later) you can `#define ORBIX_DONT_UNDEF` prior to `#include <CORBA.h>`.

DEF_TIE and TIE macros

The original versions of the DEF_TIE and TIE macros were superseded by new versions in Orbix 2.0. The original macros have been removed from Orbix 3.3. Legacy code using the original macros should be modified to use the newer macros. This requires a straightforward search-and-replace.

For an IDL interface, Account, and an implementation class, Account_i, the original DEF_TIE and TIE macros were of the following form, taking both the interface name and implementation class name as parameters:

```
DEF_TIE(Account, Account_i)
TIE(Account, Account_i)
```

The newer DEF_TIE and TIE macros (introduced in Orbix 2.0) use the interface name as part of the macro name, and only have the implementation class name as a parameter:

```
DEF_TIE_Account(Account_i)
TIE_Account(Account_i)
```

Your code should be searched for TIE, and any occurrences of the old macros changed to the newer form. The old macros generate a compile-time error in Orbix 3.3

Interoperability with Orbix 2000

The following items describe features added to Orbix 3.3 to enhance interoperability with Orbix 2000. Most of these features were retrofitted to Orbix 3.0.1, and are available in Orbix 3.0.1 patch 20 or later.

A detailed guide to interoperability between Orbix 3.3 and Orbix 2000 v1.1 is available from the Orbix 2000 documentation pages in the *Orbix 2000 Interoperability Guide*, http://www.iona.com/docs/orbix2000/orbix200011.html

IDL Compiler Switches

Two new IDL compiler switches have been added to make the Orbix 3.3 IDL compiler recognize the same keyword set as Orbix 2000 and employ the new IDL-to-C++ mapping. Orbix 3.3 continues to support the CORBA IDL that Orbix 3.0 supports. These switches provide additional consistency to a user working in a mixed Orbix 3.3/Orbix 2000 environment.

Orbix 2000 supports a richer set of IDL data types than Orbix 3.3, giving rise to additional IDL keywords. For example, `valuetype` is an IDL keyword in Orbix 2000 associated with objects-by-value (OBV). Orbix 3.3 does not provide OBV, and hence does not recognize `valuetype` as a keyword. Therefore, in Orbix 3.3., `valuetype`, could be used as an interface name, operator name, and so on. However, with this usage of `valuetype` allowed by the Orbix 3.3 IDL compiler, the same IDL would not compile under Orbix 2000 due to conflict with the keyword. A switch can be used in the Orbix 3.3 IDL compiler to increase its keyword set. This would have the effect of causing the above usage of `valuetype` to be an IDL compile-time error. The switch is `-k23`.

The original OMG mapping for IDL-to-C++ dealt with the presence of C++ keywords in the IDL by prefixing _ (underscore) to the C++ identifier. For example if `int` is used as the name of an IDL interface operation, it is mapped to a C++ `_int()` function (`int()` would be illegal C++). Orbix 3.3 still employs this mapping by default. However, Orbix 2000 uses a new OMG mapping for IDL-to-C++, which employs a leading `_cxx_` instead of a leading underscore. The mapping change does not affect interoperability. For convenience, you might wish to employ the same mapping in all of your C++ code, regardless of whether it is Orbix 3.3 or Orbix 2000. The Orbix 3.3 IDL compiler has a `-cpp_prefix` switch to control whether a leading underscore or leading `_cxx_` is used for C++ keywords in the IDL.

GIOP/IIOP Level Environment Variable

In Orbix 3.3, the GIOP/IIOP level defaults to 1.1 instead of 1.0. This is a change from Orbix 3.0. This should cause no problems, and provides increased interoperability with Orbix 2000.

Note: Orbix 3.0.1patch20 does not incorporate this change.

Invalid Object Reference and Object Not Exist System Exceptions

Orbix 3.0 uses the `INV_OBJREF` CORBA system exception to indicate that an object cannot be located in a server. This might be because the IOR is malformed (and hence can never be used to locate an object), or the server (or loader) could not find an object with the given marker. These two cases should really be distinguished, and in Orbix 2000 they are. In Orbix 2000, `INV_OBJREF` is used only to indicate that the IOR is malformed, and `OBJECT_NOT_EXIST` is used when an object for a valid IOR cannot be located.

Orbix 3.3 continues to work as Orbix 3.0 did. This is acceptable for an Orbix 3.3-only environment, which relies exclusively on `INV_OBJREF`. However, consider a mixed Orbix 3.3/Orbix 2000 environment. If an Orbix 3.3 client that is written to handle only `INV_OBJREF` exceptions connects to an Orbix 2000 server, the server might raise an `OBJECT_NOT_EXIST` exception, which the client is not prepared to handle. An Orbix 3.3 configuration variable can be used to indicate if `OBJECT_NOT_EXIST` exceptions should be mutated into `INV_OBJREF` exceptions by the ORB. This allows the client to continue to work exclusively with `INV_OBJREF` exceptions. The configuration variable can also be used on an Orbix 3.3 server to indicate whether `INV_OBJREF` exceptions should be mutated into `OBJECT_NOT_EXIST` exceptions for Orbix 2000 clients.

The configuration variable is `USE_ORBIX3_STYLE_SYSTEM_EXCEPTIONS`, and defaults to `TRUE`.

Note: This feature was retrofitted into Orbix 3.0.1 patch 20.

Communications Failure/Transient System Exceptions

Orbix 3.0 uses the `COMM_FAILURE` system exception to indicate that a remote call could not be processed, due to low-level communications failure. The failure might have occurred as the client was sending the request (implying that the request did not reach the server), or the failure might have occurred while the client was waiting for the reply (implying that the request did reach the server). These two

cases should really be distinguished, and in Orbix 2000, they are. In Orbix 2000, TRANSIENT is used for communication failures that imply that the target did not receive the message, and COMM_FAILURE is used for communication failures that imply that the target did receive the message, but an expected reply was not received.

The USE_ORBIX3_STYLE_SYSTEM_EXCEPTIONS configuration variable can be used in Orbix 3.3 to have it distinguish COMM_FAILURE and TRANSIENT.

The USE_ORBIX3_STYLE_SYSTEM_EXCEPTIONS configuration variable defaults to TRUE.

Note: This feature was retrofitted into Orbix 3.0.1 patch 20.

Further Reading

1. Henning, Michi and Stephen Vinoski. *Advanced CORBA Programming with C++*. Addison Wesley Longman, 1999.

2. IONA Technologies. *Orbix 2000 Interoperability Guide*. `http://www.iona.com/docs/orbix2000.html`

3. Object Management Group (OMG). *The Common Object Request Broker: Architecture and Specification, Revision 2.3*. 1998.

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