WebSphere Application Server CORBA support

CORBA support

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices”on page 91 .

Contents

CORBA support concept articles 1
WebSphere CORBA support 2
WebSphere CORBA support scenarios 3
CORBA client to WebSphere EJB server 4
WebSphere EJB server (as a CORBA client) to CORBA server 5
WebSphere to 3rd-party ORB interoperation 6
General CORBA interoperation considerations 6
WebSphere to 3rd-party CORBA ORB coexistence 14
The CORBA programming model 14
The CORBA client programming model 16
The CORBA server programming model 22
Interface Definition Language (IDL), usage and implementation 24
WebSphere CORBA value type library for C++ 28
C++ value type library, data type mappings 29
C++ value type library, runtime type information 30
C++ value type library, application programming interface 30
WebSphere Enterprise JavaBeans as clients of 3rd-party CORBA ORBs 31
WebSphere Enterprise JavaBeans as CORBA clients, the CORBA components 32
WebSphere Enterprise JavaBeans as CORBA clients, the CORBA interfaces 32
Writing WebSphere Enterprise JavaBeans as clients of a 3rd-party CORBA ORB 33
Writing CORBA servers for third-party ORBs 34
Problem determination 34
Hints and tips: The activity log 36
An overview of basic CORBA concepts 36
CORBA support task articles 39
Developing a C++ CORBA client 40
Creating IDL files for an Enterprise JavaBean 40
Creating the CORBA client main code (client.cpp) 41
Creating CORBA client main code (client.cpp), adding include statements and global declarations 42
Creating CORBA client main code (client.cpp), adding code to check input parameters 42
Creating CORBA client main code (client.cpp), adding code to initialize the client environment 43
Creating CORBA client main code (client.cpp), adding code to get a pointer to the root naming context 44
Creating CORBA client main code (client .cpp), adding code to access the servant object 45
Creating CORBA client main code (client .cpp), adding code to call methods on the servant object 46
Creating CORBA client main code (client.cpp), adding code to shutdown the client and release resources used 46
Building a C++ CORBA client 47
Developing a CORBA server 49
Storing a logical definition for a CORBA server in the system implementation repository 50
Defining the interface for a CORBA servant class 50
Compiling a CORBA server implementation class IDL (using idlc) 52
Adding declarations to a CORBA servant class definition (servant.ih) 53
Adding code to a CORBA servant implementation (servant_I.cpp) 54
Creating the CORBA server main code (server.cpp) 55
Creating CORBA server main code (server.cpp), adding include statements and global declarations 56
Creating CORBA server main code (server.cpp), adding code to check input parameters 57
Creating CORBA server main code (server.cpp), adding code to initialize the server environment 58
Creating CORBA server main code (server.cpp), adding code to access naming contexts 60
Creating CORBA server main code (server.cpp), adding code to name, create, and bind servant objects 63
Creating CORBA server main code (server.cpp), adding code to create a WSServerShutdown object 64
Creating CORBA server main code (server.cpp), adding code to put the server into a loop to service requests 65
Creating CORBA server main code (server.cpp), adding code to shutdown the server and release resources used 65
Building a C++ CORBA server 67
Specifying runtime properties for C++ CORBA clients and servers 68
Creating your own C++ valuetypes 69
Writing a WebSphere Enterprise JavaBean 71

as a client of a 3rd-party CORBA ORB 71
Writing a WebSphere Enterprise JavaBean as a CORBA client, contacting the client-side ORB 72
Writing a WebSphere Enterprise JavaBean as a CORBA client, locating servant objects 73
Writing a WebSphere Enterprise JavaBean as a CORBA client, invoking a servant object 74
Writing a WebSphere Enterprise JavaBean as a CORBA client, building the Enterprise JavaBean 76
Tasks for problem determination 78
Formatting an activity or trace log 78
Reading a formatted activity log 79
Formatting and merging multiple trace files 81
Filtering the information in a formatted trace file 81
Identifying and resolving CORBA interoperability issues 83
CORBA support example articles 84
Sample: C++ CORBA client of a C++ servant object 85
Sample: C++ CORBA client of an Enterprise JavaBean 86
CORBA interoperation samples 87
C++ value type library, examples 89

CORBA support concept articles

This part contains concept topics about the CORBA support provided by WebSphere Application Server 4.0 enterprise services. These topics are intended to provide background information that you should understand to be able to complete tasks to enable and use the CORBA support.

- "CORBA support task articles" on page 39
- "CORBA support example articles" on page 84
- "CORBA support reference articles" on page

WebSphere CORBA support

WebSphere Application Server provides CORBA support that enables the use of CORBA interfaces between a server object providing a service and a client using the service. In practice this means WebSphere C++ CORBA servers and WebSphere EJB services can be accessed by CORBA clients, and WebSphere CORBA clients can access CORBA servers.

As part of the WebSphere J2EE environment, the C++ CORBA support provides a basic CORBA environment that can bootstrap into the J2EE name space and can invoke J2EE transactions. However, it does not provide its own Naming and Transaction services, for which a C++ CORBA client or server relies on the J2EE environment as a service provider. The C++ CORBA technology is provided on Solaris (Forte C++), AIX (VisualAge for C++), and Windows NT and Windows 2000 (Microsoft Visual C++).

The CORBA support comprises the following two main areas of functionality:

C++ CORBA Software Development Kit (SDK)

You can use the C++ CORBA SDK to build a lightweight WebSphere CORBA server for use with new or existing C and C++ programs. You can also use the SDK to build a WebSphere C++ CORBA client for use with a WebSphere C++ CORBA server, WebSphere EJB server, or with a 3rd-party C++ CORBA server (as part of a CORBA interoperation scenario). For example, you could use the SDK to build a C++ CORBA client to connect a C++ desktop application to a WebSphere EJB server.

WebSphere to 3rd-party CORBA interoperation

You can use the CORBA interoperation functionality to invoke CORBA applications outside of WebSphere from servlets and Enterprise JavaBeans on a WebSphere EJB server (acting as a CORBA client). You can also invoke WebSphere Enterprise JavaBeans from CORBA applications calling out from CORBA outside of WebSphere.

With CORBA interoperation, distributed objects are invoked using the ORB included with WebSphere. This enables the propagation of service contexts such as in-progress transactions. Because these service contexts are potentially usable by the third-party ORB receiving the calls, you might (for example) be able to include processing performed by the third-party ORB within the scope of a WebSphere-initiated transaction.

If CORBA interoperation is inadequate, you can consider ORB coexistence as an alternative solution. Coexistence refers to the ability of two different ORB runtime environments to reside and function in the same process. With coexistence, distributed objects are invoked using a 3rd-party ORB running in the WebSphere environment. This allows the 3rd-party ORB's bootstrapping protocol and vendor-specific APIs to be used.

The CORBA interoperation functionality uses a CORBA value type library for C++, provided as part of the WebSphere CORBA support, that simplifies calls from CORBA to Enterprise JavaBeans. The CORBA interoperation environments supported include VisiBroker C++ 3.3.3, VisiBroker Java 3.4, VisiBroker C++/Java 4.0/4.1, Orbix C++ 3.0.1 (except Solaris), Orbix C++ 3.0.2 (Solaris only), Orbix Web 3.2, and Orbix2000 C++/Java 1.2. The platforms supported include AIX 4.3.3, NT 4.0, Windows 2000, and Solaris 2.7.

To make use of the CORBA support provided by WebSphere Application Server you should be familiar with the CORBA specification and programming model. You should also be

familiar with WebSphere Application Server application development.

The following articles provide the main conceptual information about the CORBA support:

- “WebSphere CORBA support scenarios” on page 3
- “The CORBA programming model” on page 14
- “Interface Definition Language (IDL), usage and implementation” on page 24
- “CORBA value type library for C++” on page 28
- “CORBA services provided” on page 11

WebSphere CORBA support scenarios

The WebSphere Application Server CORBA support enables the use of CORBA interfaces between a server object providing a service and a client using the service. In practice this means WebSphere C++ CORBA servers and WebSphere EJB services can be accessed by CORBA clients, and WebSphere CORBA clients can access CORBA servers, in the following scenarios:

- WebSphere to WebSphere CORBA scenarios.

  These scenarios enable creation of CORBA client/server applications within the WebSphere Application Server environment.

  - WebSphere C++ CORBA client to a WebSphere EJB server.

    This enables a C++ CORBA client to access Enterprise JavaBeans hosted by a WebSphere EJB server. For more information, see “CORBA client to WebSphere EJB server” on page 4 .

  - WebSphere C++ CORBA client to a WebSphere C++ CORBA server.

    This enables a WebSphere C++ CORBA client to access a CORBA server implementation object hosted by a C++ CORBA server within the WebSphere Application Server environment. This CORBA support provides the basic CORBA building blocks from which to create C++ CORBA client/server applications within WebSphere.

  - WebSphere EJB server (as a CORBA client) to a WebSphere C++ CORBA server.

    This enables Enterprise JavaBeans hosted by a WebSphere EJB server to access a CORBA server implementation object hosted by a C++ CORBA server. For more information, see “WebSphere EJB server (as a CORBA client) to CORBA server” on page 5 .

- WebSphere to 3rd-party CORBA interoperation scenarios.

  These scenarios enable 3rd-party applications based on CORBA ORBs to interoperate with WebSphere, allowing such applications to leverage WebSphere-supported open technologies such as Java ServerPages, XML, Java Servlets, and Enterprise JavaBeans. This promotes code reuse, interoperability with existing CORBA-based applications, and reduces the cost of developing new applications.

  - 3rd-party CORBA client to WebSphere EJB server.

    This enables 3rd-party CORBA clients to access Enterprise JavaBeans hosted by WebSphere EJB servers. For more information, see “CORBA client to WebSphere EJB server” on

page 4 .

- WebSphere EJB server (as CORBA client) to 3rd-party CORBA ORB.

  This enables Enterprise JavaBeans hosted by a WebSphere EJB server to access CORBA server implementation objects hosted by 3rd-party CORBA servers. For more information, see "WebSphere EJB server (as a CORBA client) to CORBA server" on page 5 .

- WebSphere to 3rd-party ORB coexistence.

  If a WebSphere to 3rd-party CORBA interoperation scenario is inadequate, you can consider ORB coexistence as an alternative solution. Coexistence refers to the ability of two different ORB runtime environments to reside and function in the same process. For more information, see "WebSphere to 3rd-party CORBA coexistence" on page 14.

CORBA client to WebSphere EJB server

CORBA clients can use the CORBA client programming model to access Enterprise JavaBeans hosted by a WebSphere EJB server, as shown in the following figure:

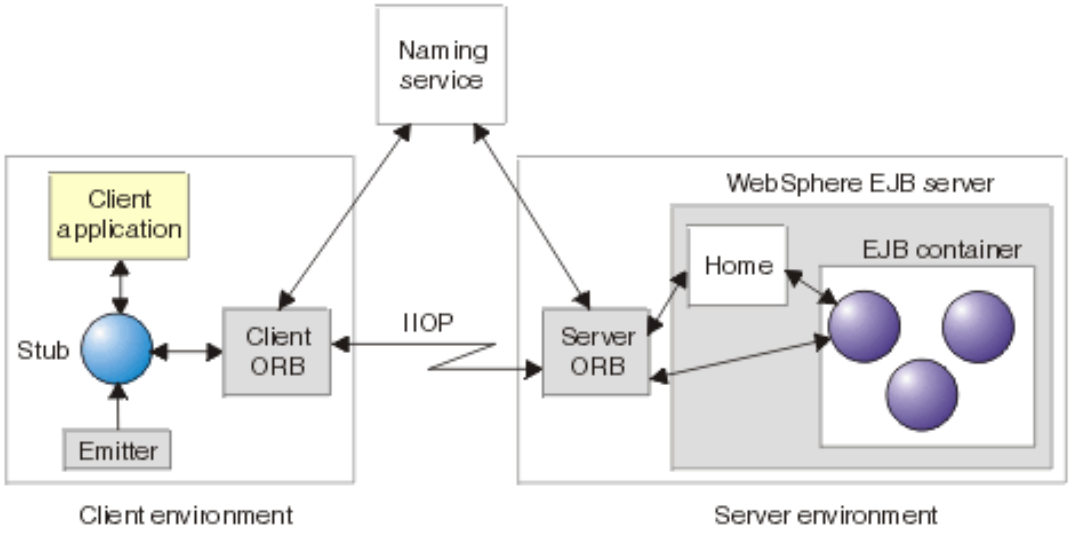

Figure 1 of 8. CORBA client to WebSphere EJB server scenario

The EJB server provides the server implementation objects (Enterprise JavaBeans) that client applications need to access and implements the services that support those objects. The EJB class file is used to generate IDL for the class and its home (this is reverse of the typical CORBA model where IDL is used to generate the object). Serializable objects used in the enterprise JavaBean interface are expressed in IDL as CORBA valuetypes. Implementations for CORBA valuetypes must be provided on the client, so it is important to keep this simple.

In this scenario, when the client wants to call a method on a server object (an Enterprise JavaBean), the following sequence of events occur:

1. When the client environment is started, the client ORB is initialized and the ORB bootstrap process gets access to the naming service (with CORBA CosNaming bindings).
2. When a client application needs to access an Enterprise JavaBean, the client

environment uses the naming service to find the home for the bean.

3. The home locates or creates the Enterprise JavaBean then passes the interoperable object reference (IOR) of the bean back to the client.
4. The client's ORB creates a stub object (local to the client) for the bean and stores the IOR in the stub object.
5. The client uses the stub object to communicate with the remote bean as though it was in the local address space.

For more information about the elements in this scenario, see “An overview of basic CORBA concepts” on page 36.

WebSphere EJB server (as a CORBA client) to CORBA server

Enterprise JavaBeans are Java objects operating in a sophisticated runtime environment (EJB containers). The EJB container manages transactions, security, and other services. These typically correspond to CORBA services. Because EJBs are Java objects they can act as CORBA clients, using the CORBA ORB managed by the container, as shown in the following figure. Enterprise JavaBeans, acting as CORBA clients using the WebSphere Java ORB, benefit from full propagation of service contexts because of the tight integration between the EJB container and the ORB.

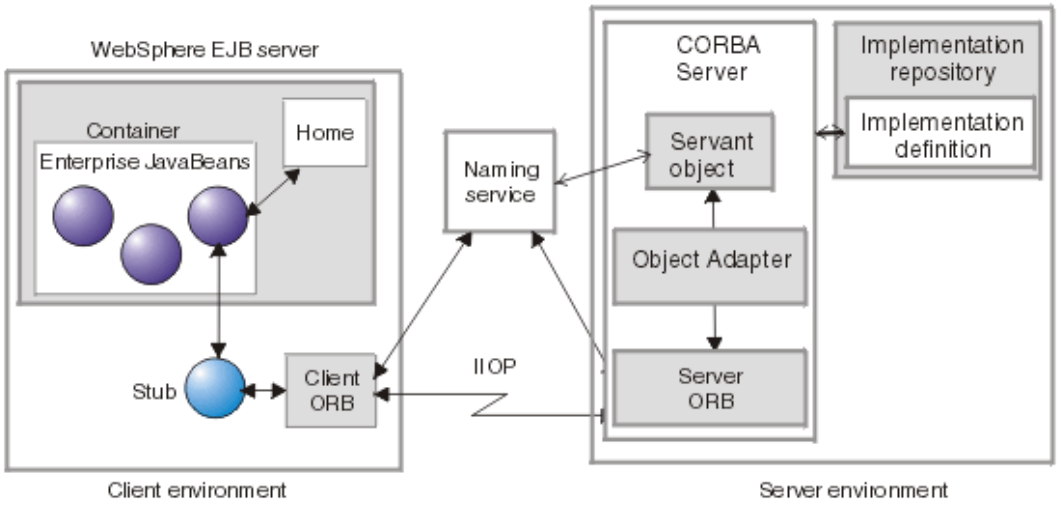

Figure 2 of 8. WebSphere EJB server (as a CORBA client) to CORBA server scenario

In this scenario, when the client (an Enterprise JavaBean on a WebSphere EJB server) wants to call a method on a servant object, the following sequence of events occur:

1. When the client environment is started, the client ORB is initialized and the ORB bootstrap process gets access to the naming service (with CORBA CosNaming bindings).
2. When an Enterprise JavaBean needs to access a CORBA servant object, the WebSphere EJB server uses the naming service to find the servant object then return the interoperable object reference (IOR) of the servant back to the client.
3. The client creates a stub object (local to the client) for the servant and stores the IOR in the stub object.
4. The client uses the IOR to locate the servant and the stub object to communicate with the servant as though it was a local process.

For more information about the elements in this scenario, see “An overview of basic CORBA concepts” on page 36.

WebSphere to 3rd-party ORB interoperation

WebSphere Application Server supports CORBA interoperation between WebSphere (C++ CORBA clients/servers and EJB servers) and 3rd-party CORBA ORBs in the following scenarios that involve programming models supported by WebSphere Application Server.

- 3rd-party CORBA client to WebSphere EJB server.

  This enables 3rd-party CORBA clients to access Enterprise JavaBeans hosted by WebSphere EJB servers. For more information, see “CORBA client to WebSphere EJB server” on page 4 .

- WebSphere EJB server (as CORBA client) to 3rd-party CORBA ORB.

  This enables Enterprise JavaBeans hosted by a WebSphere EJB server to access CORBA objects hosted by 3rd-party CORBA servers. For more information, see “WebSphere EJB server (as a CORBA client) to CORBA server” on page 5 .

If a WebSphere to 3rd-party CORBA server interoperation scenario is inadequate, you can consider ORB coexistence as an alternative solution. Coexistence refers to the ability of two different ORB runtime environments to reside and function in the same process. For more information, see “WebSphere to 3rd-party CORBA coexistence” on page 14.

The following table summarizes the CORBA scenarios supported for interoperation and coexistence between WebSphere Application Server enterprise services and 3rd-party CORBA ORBs:

CORBA client	EJB/CORBA server	Supported
Scenario		***
3rd-party ORB, CORBA C++ language bindings	WebSphere-hosted Enterprise JavaBean	Yes
WebSphere ORB, CORBA C++ language bindings	3rd-party Java CORBA object	Yes
WebSphere ORB, CORBA C++ language bindings	3rd-party C++ CORBA object	Yes
WebSphere ORB, CORBA Java language bindings (EJB server as CORBA client)	3rd-party Java CORBA object	Yes
WebSphere ORB, CORBA Java language bindings (EJB server as CORBA client)	3rd-party C++ CORBA object	Yes
WebSphere ORB coexistent with 3rd-party ORB	3rd-party CORBA object	Yes

General CORBA interoperation considerations

This topic provides an overview of general considerations for interoperation between WebSphere Application Server and 3rd-party CORBA ORBs.

Why CORBA interoperation is an issue

Two CORBA ORBs should be able to interoperate, but in practice interoperation works well

for two ORBs from the same vendor, but not so well for ORBs from different vendors. Even though the CORBA specification is designed to allow vendor ORBs to interoperate, there are many factors that restrict or prevent full interoperation, including:

- Proprietary extensions to Vendor ORB implementations:
    - Use different configuration and initialization procedures
    - Other features affecting interoperability that, while desirable and beneficial, are beyond the scope of the CORBA specification, and therefore are not supported by other ORBs.
- Vendors may implement different levels of the CORBA specification. This is especially problematic for client access to Enterprise JavaBeans, which requires features found in the GIOP 1.2 protocol.
- Vendors may implement different parts of the same specification.
- There may be bugs in an implementation of the CORBA specification.
- Ambiguities in the CORBA specification, where different vendors have interpreted the specification differently. OMG has a process defined to resolve such ambiguities as they are identified, however vendors typically proceed with their own interpretation until such ambiguities are clarified by the OMG.

Even if two ORBs fully conform to the CORBA specification, you usually need to configure each ORB implementation to recognize the presence of the other ORB or, more specifically, to initialize references to objects implemented on the other ORB.

Different host operating systems can impose different configuration and initialization procedures. You should carefully review the relevant documentation from each ORB vendor specific to the host operating environment. ORB vendors support a variety of operating systems, but not all operating systems are supported by any one ORB vendor.

Resolving the CORBA interoperation issue

The CORBA specification and vendor ORBs are continually evolving, and are expected to evolve to the point where interoperability is adequate for most, if not all, scenarios. Meanwhile, it is essential to understand what interoperation works as expected, what does not work, and what options are available for resolving the interoperation issues.

There are many interdependent factors that must be considered when designing a CORBA interoperation solution between ORBs from different vendors, or resolving problems in an existing design.

GIOP and J2EE

Communication between different ORBs is based on the General Inter-ORB Protocol (GIOP) specification, and the Internet Inter-ORB Protocol (IIOP) implementation of the GIOP. For J2EE 1.3, IIOP is required for J2EE products to interoperate; therefore Enterprise JavaBeans must be accessible via the RMI-IIOP protocol. Also, CORBA is the J2EE distributed interlanguage specification, so JavaIDL is required to support J2EE applications calling CORBA applications.

CORBA language binding considerations

Different languages require different language bindings to a Vendor's ORB, and may even require different ORBs from the same vendor. This requires a level interoperability between the ORBs, which should be taken into consideration. The CORBA architecture defines language bindings for a number of languages, including C++, Java, COBOL, PL/I, Smalltalk, and others. CORBA concepts are generally language independent, although valuetype bindings have not yet been defined for all language bindings.

C++ language bindings are available for C++ CORBA clients and servers supporting 1.1 and valuetypes from IIOP 1.2. To aid application development, WebSphere Application

Server enterprise services provides a valuetype library that contains the C++ valuetype implementation for some commonly used Java classes in the java.lang, java.io, and java.util packages. For example, Integer, Float, Vector, Exception, OutputStream, and so on.

CORBA value type considerations

The Java language to IDL specification maps Java serializables to CORBA valuetypes (pass-by-value objects). Therefore every Java serializable to be passed between a client and server (for example, by a CORBA client as a parameter or return value for an Enterprise JavaBean) must be re-implemented in the language of the client. (The implementation for the valuetype must be defined and provided in the language runtime for both the client and the server.) Implementation of Java serializables as valuetypes in C++ or another language can be a significant development effort.

Valuetypes were introduced by the CORBA 2.3 specification and many 3rd-party ORBs do not yet implement the specification, or do not implement it fully.

To aid application development, WebSphere Application Server provides a valuetype library that contains the C++ valuetype implementation for some commonly used Java classes in the java.lang, java.io, and java.util packages. For example, Integer, Float, Vector, Exception, OutputStream, and so on. For more information about the valuetype library provided with WebSphere Application Server, see “WebSphere CORBA valuetype library for C++” on page 28.

Java language to IDL specification

An Enterprise JavaBean is implemented in Java, with no hint of the CORBA architecture in its programming model. The Enterprise JavaBean specification requires that the server implementation be restricted to using those Java language constructs defined as the RMI/IDL subset by the Java language to IDL specification.

By following the Java language to IDL specification, you can create CORBA clients implemented in any programming language for which there is a defined mapping, and for which an ORB supporting valuetypes is available.

When valuetypes are not supported

For languages other than Java, such as C++, the CORBA architecture is often the only viable option for accessing Enterprise JavaBeans.

For session bean interfaces that only use primitive data types, you can use generated IDL files to access the Enterprise JavaBeans even if the client ORB does not support valuetypes. However, the IDL generated from such an Enterprise JavaBean can still include valuetype declarations for exceptions or other entities.

For valuetypes, java.lang.String or any other serializable cannot be used in a parameter or return type. There can also be problems catching exceptions, because they contain valuetypes.

If you decide that the features supported by valuetypes are not needed, consider using the strategies outlined in “Unsupported CORBA data types” on page 10.

CORBA communication protocols (GIOP/IIOP)

The CORBA architecture provides the General Inter-ORB Protocol (GIOP) to define message formats between objects in a distributed environment. The Internet Inter-ORB Protocol (IIOP) is an implementation of GIOP.

GIOP includes a Common Data Representation (CDR) that resolves differences between native hardware architectures[1] within such an environment. GIOP supports a number of

simple data types, compound data types, object references, exceptions, and other features, depending upon the version of the specification (as shown in the following table). Nevertheless, early ORBs can encounter interoperability problems related to byte ordering.

If you suspect a GIOP-related interoperability problem, it is reasonably safe to adopt GIOP 1.0; all major ORBs support GIOP at this level. WebSphere supports client ORBs that use GIOP1.0 or GIOP1.1, with or without valuetypes from 1.2. Client ORBs cannot call WebSphere using GIOP1.2. Likewise, WebSphere ORBs can call 3rd-party CORBA servers at GIOP1.1 only. The WebSphere ORB does not accept or produce fragmented messages.

GIOP feature	Data types	GIOP version			WebSphere ORB	
1.0	1.1	1.2	Java	C++		
Simple data types	octet, char, short, unsigned short, long, unsigned long long long, unsigned long long, float, double, boolean, string	Yes	Yes	Yes	Yes	Yes
long double	Yes	Yes	Yes	-	-	
fixed	Yes	Yes	Yes	-	-	
wchar, wstring	-	Yes	Yes[1]	1.1	1.1	
enum	Yes	Yes	Yes	Yes	Yes	
Compound data types	struct, union, array, sequence	Yes	Yes	Yes	Yes	Yes
valuetype	-	-	Yes	Yes	Yes	
CORBA::Object	Yes	Yes	Yes	Yes	Yes	
any	Yes	Yes	Yes	Yes	Yes	
context	Yes	Yes	Yes	Yes	Yes	
exception	Yes	Yes	Yes	Yes	Yes	
Message formats	Request, Reply,	Yes	Yes	Yes	Yes	Yes

[1] Different hardware architectures can have variations in byte ordering and alignment for multi-byte data types within the address space. GIOP provides the means for resolving the differences across platforms.

[2] There are differences in the wchar and wstring encoding betweenGIOP 1.1 and 1.2, which creates interoperability challenges. For detailed information, review the portion of the CORBA specification relating to GIOP and CDR.

GIOP feature	Data types	GIOP version			WebSphere ORB	
1.0	1.1	1.2	Java	C++		
	CancelRequest, LocateRequest, LocateReply, CloseConnection, MessageError					
Fragment	Yes	Yes	Yes	Yes	Yes	
Bi-directional	-	-	-	Yes	-	-

Resolving unsupported CORBA data types

If a client ORB does not support a data type required by a server object, such as an Enterprise JavaBean or C++ servant object, you can use a variety of techniques to resolve this, including the following:

- “Removing a data type that is not needed from the IDL for a server object” on page 10.
- “Using a wrapper to hide an unsupported data type” on page 10.
- “Using the dynamic invocation interface to call the server” on page 11

Removing a data type that is not needed from the IDL for a server object

If a client ORB does not support a data type defined in the IDL file for a server object, and the client does not need to use the associated feature, you can remove the data type from the IDL used to create the client. For example, you can use the following steps to enable the client to use a version of the IDL file to access the server object:

1. Generate the IDL file that represents the server object. For example, to generate IDL for an Enterprise JavaBean, run `rmic -idl` on the Enterprise JavaBean's home and remote interfaces.
2. Make a copy of the IDL file to use with the client.
3. Edit the IDL for the client to remove all references to unsupported data types. This can involve removing exceptions, objects, attributes, and methods (but not individual parameters or return types).
4. Compile the IDL, and link the client with the generated bindings.

Using a wrapper to hide an unsupported data type

You can use a wrapper to hide unsupported data types needed by a server object behind a thin intermediate server object. The wrapper can be a CORBA object or a session bean. If a wrapper is being used to resolve an ORB that does not support valuetypes, then the wrapper should be implemented as a CORBA object to avoid the various extraneous valuetypes generated by the EJB-to-IDL compiler.

A wrapper provides an alternate and supported interface, and delegates its implementation to the original server object. The CORBA client accesses the intermediate wrapper, and the wrapper is deployed on a server that can directly access the target server object. The wrapper interface must be designed such that it provides access to the target object's interface without using valuetypes (for an EJB server) or other unsupported data types.

A wrapper may be the only way to get client access working for some vendor ORBs.

When using wrappers, consider the following points:

- Management: The wrappers must be installed and managed in a CORBA server.
- Lifecycle: For an Enterprise JavaBean, the EJB container manages the lifecycle

automatically. If the wrapper is a CORBA object in the same server as the Enterprise JavaBean, the wrapper's lifecycle must be explicitly managed in your code[1]. In this situation, it is better to put CORBA wrappers in a CORBA server and Enterprise JavaBeans in a different EJB server.

- Data types: The wrappers must convert between Enterprise JavaBean types and IDL types, unless the client uses only primitive types. For example, EJB object references must be converted into IDL object references. Also, Java serializables have to be converted into IDL equivalents.

  The client marshals data into an opaque octet stream and passes it to the IDL wrapper. The IDL wrapper demarshals the data, inflating java objects by value if necessary, and passes data on to the target server object. For a target Enterprise JavaBean, this can be done in a session bean, which is free to use RMI-IIOP and valuetypes while interacting with entity beans.

- Hand coding: Both the wrapper interface design and the wrapper object must be hand-coded.

Using the dynamic invocation interface to call the server

As a last resort, the CORBA Dynamic Invocation Interface (DII) enables a client to make a call to a server without using IDL. Instead, the client makes a call by constructing the method parameters dynamically, storing them as CORBA::Any data types. This mode of access can be useful in the following cases:

- Accessing remote objects, such as Enterprise JavaBeans, when the representative IDL might contain unsupported data types with the following characteristics:
  - The data types would prevent the IDL from being compiled for by the client ORB's idl compiler.
  - The data types are not required for the methods or features that must be accessed.
- The client does not have prior knowledge of the IDL definitions.
- The repository ID of an Enterprise JavaBean interface is redefined; for example some older ORBs do not support the `rmi:` prefix.

CORBA object services

CORBA object services interoperate by delivering context information, with messages, that establishes service state and other parameters. Some older ORBs do not support the passing of such context, or use proprietary context data that cannot interoperate with another server.

Conversely, because a service context is *not* part of the message normally seen at the programmer's level, solutions that involve a break in the normal flow of a message do not automatically propagate a service context. Such solutions include wrapper classes or messages manually propagated across coexistent ORBs. If context propagation is required under such circumstances it must be explicitly or manually managed in the code. If available, request interceptors provide a useful way to propogate contexts.

Naming service

For CORBA applications, WebSphere supports the CORBA CosNaming service, which binds CORBA objects to a public name. Clients are bootstrapped according to the CORBA programming model, CORBA-compliant IORs must be obtained, and server objects must be bound into the CORBA CosNaming service. (For CORBA client access to Enterprise JavaBeans, the EJB home must be bound into the CORBA CosNaming service.)

For more information about the naming service, see "The naming service" on page 12.

[3] For Session Beans, you can manage the wrapper's lifecycle by unexporting and destroying the wrapper when the bean is removed. Since not all beans are eventually removed, like entity beans, the unexport and destruction of the wrapper may have to be explicitly exposed to the client's programming model.

Transaction service

WebSphere supports the Enterprise JavaBean object transaction service (OTS). WebSphere follows the CORBA transaction service specification for propagating transaction contexts, and forwards the transaction context to the server. For interoperation with 3rd-party ORBs, incoming contexts are honored and outgoing transaction contexts are generated as appropriate.

For more information about the transaction service, see "The transaction service" on page 13.

Security service

The CORBA specification does not yet define an accepted interoperable security standard.

The WebSphere C++ CORBA SDK does not support a Security Service. Therefore, a C++ CORBA client, developed using this SDK, cannot authenticate to a secure WebSphere EJB server (so can only access objects which are configured to allow unauthenticated users to call them).

When using CORBA interoperation between WebSphere and a 3rd-party ORB, a security context can flow, although it may be ignored. You can use a coexistent ORB to pass a security context manually, though it is your responsibility to pass relevant context information.

For more information about the security service, see "The security service" on page 13.

The naming service

WebSphere supports the CORBA CosNaming service, which binds CORBA objects to a public name. Clients are bootstrapped according to the CORBA programming model, CORBA-compliant IORs must be obtained, and server objects must be bound into the CORBA CosNaming service. (For CORBA client access to Enterprise JavaBeans, the EJB home is automatically bound into the CORBA Naming Service and is therefore can be accessed through the CosNaming interfaces.)

The naming service provides a mapping between names and object references. When an object is created, it is assigned an object reference, which can be bound with a ::CosNaming::Name name into the namespace managed by the naming service. Any client (or any other object) with access to the naming service can use the associated ::CosNaming::Name name to retrieve the object reference.

The namespace is hierarchical and similar in structure to a file system directory tree. The nodes of the namespace are CORBA::objects (either NamingContext objects or leaf objects). A NamingContext object, or naming context, can contain zero or more bindings of name-object reference pairs. Each object, bound by name into a naming context, can be a leaf object or a subordinate NamingContext in the tree. Subordinate NamingContexts similarly can contain bindings of other NamingContexts and leaf objects.

For example, a servant object called WSLoggerObject1 is bound into naming context called WSLoggerContext (which was created by a CORBA server for the servant objects that it hosts). The WSLoggerContext naming context is bound into the domain naming context called domain, which is bound into the root naming context for the naming service. This could be represented by the object reference domain.WSLoggerContext.WSLoggerObject1 and represented by the hierarchy:

```
        /       (root)
        |
    domain
        |
```

```
WSLoggerContext
        |
WSLoggerObject1
```

This can also be represented by the name string "/domain/WSLoggerContext/WSLoggerObject1"

The transaction service

WebSphere supports the object transaction service (OTS) and follows the CORBA transaction service specification for propagating transaction contexts; it forwards the transaction context from a client to the server. An ORB uses incoming transaction contexts to either handle transactions transparently or ignore transaction contexts that it does not understand.

For transactional support, a CORBA client of an Enterprise JavaBean must rely on one of the following options:

- Container-managed transactions, where the container automatically starts and ends each new transaction
- Bean-managed transactions
- Client-initiated transactions.

WebSphere C++ CORBA clients and servers only provide a client-side transaction service; they can only act as a transactional client to a server which supports a transaction service (for example, a WebSphere EJB server). Objects on a WebSphere C++ CORBA server are not recoverable.

In many cases the Enterprise JavaBean infrastructure within WebSphere automatically initiates transactions, even if the application code does not. If an Enterprise JavaBean calls a CORBA server from within a transaction, the following may happen:

- (Best) the CORBA server resources are coordinated with the transaction, or there are no resources to coordinate.
- The server does not recognize the transaction context, so it is ignored. The application writer must recognize this and code accordingly.
- The server crashes due to the presence of the WebSphere transaction context. The application writer must either design the Enterprise JavaBean to not run from within a transaction context, or use coexistence to ensure that the transaction context is not automatically propagated. To disable automatic creation of transaction contexts: deploy the WebSphere Enterprise JavaBean with a transaction attribute other than the default TX_REQUIRED: use TX_NOT_SUPPORTED, TX_SUPPORTS, or an equivalent transaction attribute that does not force the creation of transaction context.

If an EJB client is to use a CORBA server in the scope of a transaction, consider the following transaction timeout issue. If the Enterprise JavaBean is executing a loop, or if the server object takes an excessive amount of time to execute, and the Enterprise JavaBean executes within the scope of a single transaction, then built-in transaction timeouts can be exceeded. This causes unexpected failures that have nothing to do with CORBA architectural issues. This is natural behavior, but not necessarily expected if you are not familiar with the issues. This can be avoided by creating a new transaction each time through the loop.

For clients to third-party ORBs, you can use the coexistent ORB solution to start 3rd-party transactions.

The security service

The CORBA specification does not yet define an accepted interoperable security standard. You can use a coexistent ORB to pass security context manually, though it is your

responsibility to pass relevant context information.

WebSphere to 3rd-party CORBA ORB coexistence

If a WebSphere to 3rd-party interoperation scenario is inadequate, you can consider ORB coexistence as an alternative solution. Coexistence refers to *'the ability of two different ORB runtime environments to reside and function in the same process'*. It is important to understand that the Java ORB Portability Interfaces were designed to facilitate *a selection* of multiple ORBs, with the intent that there would be one ORB loaded at a time. These interfaces were not designed to support multiple ORBs coexisting at the same time.

An advantage of coexistence is that the client does interoperate with the server (using the same ORB or, at least, ORBs from the same vendor) at all levels: IIOP, services, and proprietary features.

However, the runtimes of different ORBs can interact and behave in surprising ways. Older ORBs that define incompatible org.omg.* runtime APIs cannot coexist with WebSphere: the stubs implemented to these APIs do not work in the WebSphere environment. At best, they do not compile; at worst, they compile with hidden side effects. Newer ORB runtimes that use the Java ORB portability interfaces, as defined in the Java language mapping specification for the OMG Interface Definition Language (IDL) 2.3, should coexist better although, even for newer ORBs, some incompatibilities can exist.

To enable coexistence, you can need more handcrafting of code, because the application cannot rely solely on an environment that presupposes one ORB (for example, EJB containers).

Each ORB has its own independent root name context, which may or may not be equivalent. These name contexts are independent, and an object reference available in one should not be expected to be available in the other. However, in theory, an IOR obtained from one naming context should be valid in any ORB.

A coexistent ORB does not propagate service contexts for CORBA services from other ORBs. For example, for the security service, two coexisting ORBs have two different security contexts, authentications, authorizations, and so on.

A 3rd-party client ORB runtime coexisting with a WebSphere EJB server runs independent of the WebSphere EJB container or its controlling infrastructure. Therefore the 3rd-party ORB is unlikely to look for or understand service contexts on the thread of execution, believing itself to be a client residing in its own process. Likewise, the EJB container has no knowledge of the 3rd-party ORB. Thus the WebSphere context is not considered within the execution of a method to an object accessed through a coexisting 3rd-party ORB.

The CORBA programming model

The CORBA programming model describes the artifacts that you develop and implement to enable client applications to interact with server applications in a CORBA environment. In this context, the word client refers to any program running in a process on a client or server computer. The server, a server process, hosts a servant object (in CORBA 2.3 terminology) through which the client accesses business functions. In a C++ CORBA server, the servant object implements the business functions; in an EJB server, the business logic is implemented by an Enterprise JavaBean.

The CORBA programming model, as a distributed-object programming model, is characterized as follows:

Objects

CORBA objects are defined with the OMG Interface Definition Language (IDL). IDL

is compiled to generate client stubs and server skeletons, which map an object's services from the server environment to the client.

Communications protocol

The specification is the General Inter-ORB Protocol (GIOP), of which the Interoperable Inter-ORB Protocol (IIOP) is one implementation.

Object references

CORBA Interoperable Object References (IOR) provide a platform and vendor independent object reference.

Naming service

The CORBA CosNaming service is bootstrapped with resolve_initial_references(). CosNaming binds a CORBA object to a public name.

"The CORBA programming model" on page 15shows the artifacts that you develop and implement for the CORBA programming model.

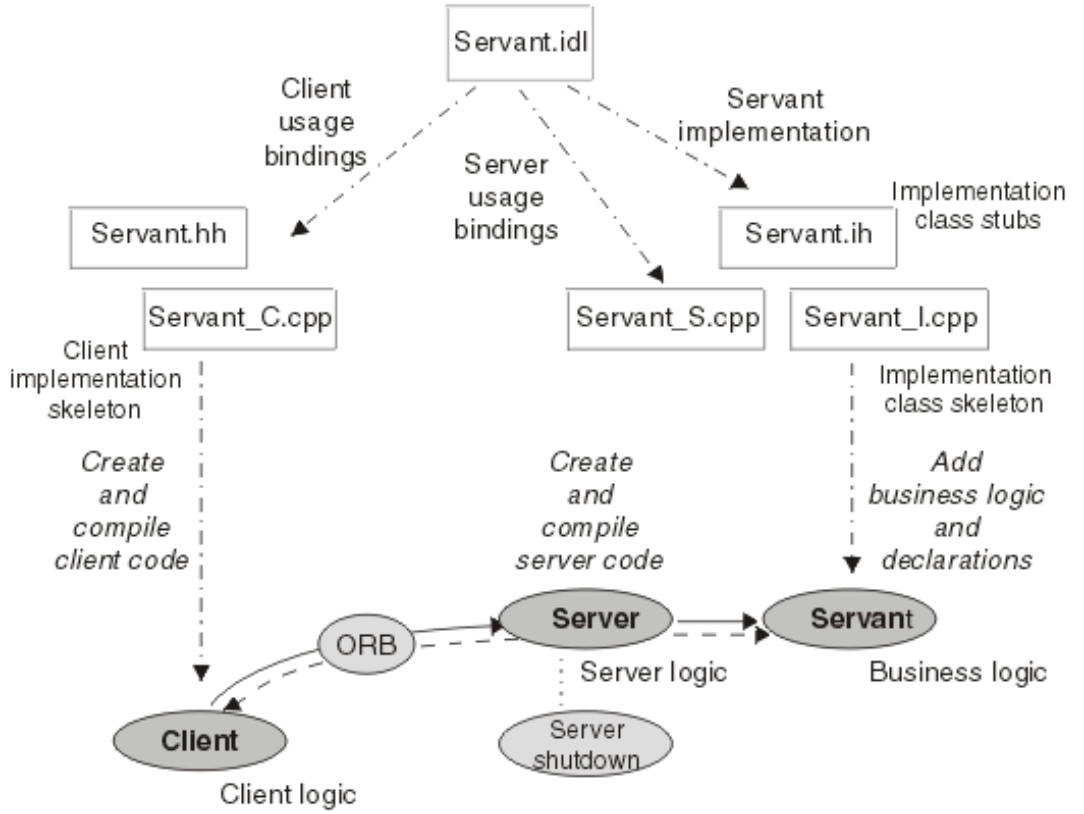

Figure 3 of 8. The CORBA programming model

The CORBA programming model comprises the following two interrelated parts:

- The server programming model describes the interfaces and processes used to develop CORBA server objects that make up the business logic and business data inherent in a server application. Application programmers use the server programming model if they are developing CORBA server objects that perform business functions that are used in the implementation of client objects. For more information about the server programming model, see "The server programming model" on page 22.
- The client programming model describes what client applications do to make use of objects provided by server applications. Application programmers developing tier-1 (client) or tier-2 (server) CORBA applications use the client programming model if they

are developing CORBA clients whose implementation uses either CORBA server objects or Enterprise JavaBeans. For more information about the client programming model, see "The client programming model" on page 16.

In WebSphere Application Server enterprise services, the CORBA client and server programming models are used as follows:

- The CORBA client programming model is used for WebSphere C++ clients, WebSphere Java clients (including WebSphere EJB servers acting as CORBA clients) that want to access a WebSphere C++ Server or 3rd-party CORBA ORB acting as a CORBA server.
- The CORBA client programming model is also used for WebSphere C++ clients that want to access a WebSphere Enterprise JavaBean[1]
- The CORBA server programming model is used for WebSphere C++ servers and 3rd-party CORBA ORBs

The CORBA client programming model

This topic describes the CORBA client programming model, which describes what CORBA clients do to use either CORBA server objects or Java Enterprise JavaBeans. Application programmers use the CORBA client programming model to develop tier-1 (client) or tier-2 (server) CORBA applications whose implementation uses either CORBA server objects or Java Enterprise JavaBeans.

The information about the CORBA client programming model, provided in the following topics, is based on developing C++ CORBA clients:

- "C++ CORBA client, initializing the ORB" on page 16
- "C++ CORBA client, locating the root naming context (bootstrapping)" on page 16
- "C++ CORBA client, locating a servant object" on page 18
- "C++ CORBA client, using a servant object" on page 18
- "C++ CORBA client, locating an EJB home" on page 18
- "C++ CORBA client, using an Enterprise JavaBean" on page 19
- "C++ CORBA client, using client-side object references" on page 19
- "Coding tips for proper CORBA memory management" on page 19
    - "Using object references" on page 20

Examples of client programming are given in the WSLoggerClient sample, which if you selected to install the enterprise services' samples, is installed in the following directory: `WAS_HOME/Enterprise/samples/sampcppsdk`. Information about how to build and run the samples is also provided: see `WAS_HOME/Enterprise/samples/index.htm`.

C++ CORBA client, initializing the ORB

One of the first things that a C++ CORBA client application needs to do when it is started is to initialize the client ORB and return a pointer to it, by calling the CORBA::ORB_init() method. (If necessary, this method creates a new instance of the ORB.) For example, the following code extract initializes the ORB and return a pointers to it:

```
op = ::CORBA::ORB_init(argc, argv, "DSOM");
```

Where argc and argv refer to the properties specified on the command used to start the server. On the CORBA::ORB_init() method you must specify `DSOM` after the parameter *argv*.

C++ CORBA client, locating the root naming context (bootstrapping)

The naming service can be used to manage a directory of objects, to map the name of each object to its associated object reference. To locate a server object somewhere in a CORBA environment, a client can locate the naming service, then use a name to retrieve an

[4] The J2EE server programming model is used for WebSphere EJB servers.

associated object reference from the naming service.

The location of the naming server that provides the naming service, and the number of the port that it uses to communicate with clients and servers, are specified by enterprise services' runtime properties. The values that you specify for the runtime properties must match the equivalent settings used to configure WebSphere Application Server.

Object references are bound into the naming service relative to the root naming context.

When a client is started, it uses a "bootstrapping" operation to locate the naming service then locate the root naming context, as follows:

1. The client calls the ORB::resolve_initial_references("NameService") method, which returns a CORBA::Object.
2. Before the client can use the returned reference as a naming context, the client must narrow the object to the desired class.

For example:

```
objPtr = op->resolve_initial_references( "NameService" );
 rootNameContext = ::CosNaming::NamingContext::_narrow(objPtr);
```

The resolve_initial_references("NameService") method is implemented according to a pre-INS specification. It does work with earlier versions of some 3rd-party ORBs, but not with ORBs that implement the current INS specification.

If the client bootstrapping operation does not establish contact with a remote naming service, you can use alternative strategies to retrieve the IOR of the naming service, as outlined in “Strategies for retrieving the IOR of a remote object” on page 17.

Strategies for retrieving the IOR of a remote object

If the client bootstrapping operation does not establish contact with a remote naming service, you can use the following alternative strategies to retrieve the IOR of a remote object:

- Have the ORB use a name service URL for the Name Service initial reference.

  You can obtain the remote ORB's root name context and stringify it into a file. During ORB initialization, CORBA clients and servers can set the ORB property com.ibm.CORBA.InitialReferencesURL to the URL of the file that contains the stringified IOR of a root naming context. The root naming context is then returned by calling resolve_initial_references("NameService").

- Pass the naming service object reference directly to a client.

  You can write an application to store the stringified IOR of a remote ORB's root naming context into a file. You can then make the file available (for example, by copying) to the client environment. The client can then read the stringified IOR from the supplied file and use the ORB::string_to_object interface to resolve the root naming context. This approach needs to be used only once during initialization, even if the client is to access many different server objects registered with the same naming service. In addition, the IOR for the name server is typically fairly static, so it is relatively simple to manage in a distributed environment.

- Similar to the preceding option, put a stringified IOR for a remote object into a file and have the client read that IOR and use the ORB::string_to_object interface to resolve the object reference, but not use a Name Service at all.
- Name space federation.

  The client can look up an entry in the name server of one ORB, then rebind the

reference in the name server of a different ORB. For example, you can write a utility to look-up an EJB's home in the WebSphere name service, stringify the object reference and write it to a file. You can then use another utility to read this file, destringify the object reference and bind it into a 3rd-party ORB's naming service.

- Bind a remote object into the local Name Space.
- Use a coexistent naming service.

  A client can make use of a coexistent 3rd-party ORB that supports bootstrapping with other ORBs from the same vendor.

C++ CORBA client, locating a servant object

To be able to locate a servant object somewhere in a CORBA environment, a client needs to know the object reference that uniquely identifies the target object.

When an object is created, it is assigned an object reference, which can be bound with a name in the naming service. Any client (or any other object) with access to the naming service can use the associated name to retrieve the object reference.

Object references are bound into the naming service relative to the root naming context. After a client has located the root naming context, it can use the standard CosNaming interface to navigate the name space and retrieve the object reference associated with any name; for example:

```
// Create a new ::CosNaming::Name to pass to resolve().
     // Construct it as the full three-part complex name.
     ::CosNaming::Name *loggerName = new ::CosNaming::Name;
     loggerName->length( 3 );
     (*loggerName)[0].id = ::CORBA::string_dup( "domain" );
     (*loggerName)[0].kind = "";
     (*loggerName)[1].id = ::CORBA::string_dup( "WSLoggerContext" );
     (*loggerName)[1].kind = ::CORBA::string_dup( "" );
     (*loggerName)[2].id = ::CORBA::string_dup( "WSLoggerObject1" );
     (*loggerName)[2].kind = ::CORBA::string_dup( "" );
     ::CORBA::Object_ptr objPtr = rootNameContext->resolve( *loggerName );
     liptr = WSLogger::_narrow( objPtr);
```

If the client bootstrapping operation does not establish contact with a remote naming service, you can use the alternative strategies to retrieve the IOR of a remote object, as outlined in “Strategies for retrieving the IOR of a remote object” on page 17

C++ CORBA client, using a managed object

When a client has retrieved a reference to a servant object, you can invoke methods on that object. For example, `liptr->setFileName("log.out");` calls the setFileName() method on the object identified by the liptr object reference to set the value of the FileName attribute.

C++ CORBA client, locating the EJB home

If a C++ CORBA client is to access an Enterprise JavaBean, it needs to locate the EJB home. After locating the root naming context, the client can use the Naming Service to locate the EJB home. In WebSphere Application Server 3.5, Enterprise JavaBeans were bound into the root naming context that was shared by all hosts using the same administrative database. In WebSphere Application Server 4.0, each host has its own root context. To maintain the semantic compatability between WebSphere Application Server 3.5 and 4.0, all the Enterprise JavaBeans in 4.0 are bound into the domain's legacy root. This is equivalent to the root naming context to which all Enterprise JavaBeans were bound in WebSphere Application Server 3.5.

In WebSphere, the JNDI name for a bean is mapped to the home class for that bean. The JNDI name is specified in the ibm-ejb-jar-bnd.xmi file generated for the deployed bean's jar file. If you run the command *was_root*`/bin/dumpNameSpace`, you can see the mapping

of the JNDI name to the corresponding Java class. For an Enterprise JavaBean, the JNDI name `(top)/ejbhome` is mapped to the home class. The Enterprise JavaBean is located in (top), the WebSphere Application Server 4.0 equivalent to the domain's legacy root. The three components of (top) are "domain", "legacyRoot", and *ejbcontext*. The following code creates a COSNaming Name for the bean's full path:

```
  ::CosNaming::Name *ejbName = new ::CosNaming::Name;
ejbName->length( 4 );
(*ejbName)[0].id = ::CORBA::string_dup( "domain" );
(*ejbName)[0].kind = "";
(*ejbName)[1].id = ::CORBA::string_dup( "legacyRoot" );
(*ejbName)[1].kind = ::CORBA::string_dup( "" );
(*ejbName)[2].id = ::CORBA::string_dup( "ejbcontext" );
(*ejbName)[2].kind = ::CORBA::string_dup( "" );
(*ejbName)[3].id = ::CORBA::string_dup( "ejbhome" );
(*ejbName)[3].kind = ::CORBA::string_dup( "" );
```

The steps to locate an EJB home are:

1. Creating a COSNaming Name for the Enterprise JavaBean's full path:
2. Calling resolve (with Name) on the root naming context
3. Narrowing the object returned by resolve() to the appropriate type, an object pointer to the EJB home.

For an example of locating an EJB home, see the samples article "Tutorial: Creating a user-defined C++ client that uses an EJB" at `WAS_HOME/Enterprise/samples/sampcppsdk/ejbsamp/hellosamp/wsBuildEJBClient.htm` (if you have installed the samples option).

The object pointer to the EJB home can be used to create a Enterprise JavaBean object, as described in “Using an Enterprise JavaBean” on page 19.

C++ CORBA client, using an Enterprise JavaBean

When a client that wants to access an Enterprise JavaBean, it first locates the EJB home as described in “C++ CORBA client, locating the EJB home” on page 18 . The object pointer to the EJB home can then be used to create an Enterprise JavaBean object; for example:

```
ejbPtr = ejbHomePtr->create();
```

When the Enterprise JavaBean object has been created successfully, any of its methods can be called; for example:

```
msg = ejbPtr->message();
```

C++ CORBA client, using client-side object references

A client can pass a local object reference to the server, enabling the server to make a method call back to a client-side object. This callback capability is useful if the client needs to be notified of some event on the server. It is also useful if the server needs customized client functionality to fulfill its implementation.

A callback to the client is conceptually no different from any client/server situation: when a client passes an object reference C to a server S, and S invokes a method on C, the server S is acting as a client, and C is acting as a server.

A callback object can be an Enterprise JavaBean, which the server (acting as client) can access through the various means of client access to an Enterprise JavaBean. In particular, if the server object is implemented in C++, you must deal with all the complexities of calling an Enterprise JavaBean from a C++ client.

Coding tips for proper CORBA memory management

The rule for proper CORBA memory management is that the caller owns all storage.

Memory management in Java is somewhat automatic.

The general model for C++ CORBA memory management on the client is to use _var objects. This means that when an _ptr is returned, it should be placed into an _var by the client. The _var assumes responsibility for the storage pointed to by the _ptr that is placed into the _var. The _var is a class and its destructor runs when the _var goes out of scope.

The other option for C++ CORBA clients is to use the duplicate() and release() methods. The duplicate() method is available for making a copy of a client stub object, while the release() method is used to free the local memory used by a pointer.

For more reference information about C++ CORBA memory management, see “CORBA programming: Storage management and _var types” on page .

Managing the storage of object references

Managing the storage of object references is one of the areas where proper memory management is required. You must use use _var variables or the duplicate and release methods as stated above.

There are also special considerations when passing object references as parameters. The caller is always responsible for allocating storage for object references. The caller is also responsible for releasing of all inout and returned object references.

For inout parameters the caller provides an initial value. If the callee wants to reassign the inout parameter, it must first all the release() operation on the initial input value. To continue to use an object reference passed as an inout, the caller must first duplicate the reference.

CORBA support, handling exceptions

The preferred coding practice for handling errors in C++ and Java is by using exceptions, which is supported by using the standard try and throw logic of exception handling. Handling exceptions is a critical part of the client programming model. The exceptions that are thrown must be understood and handled appropriately by application developers.

In some cases, a server implementation object can encounter an error for which it might need to throw an exception to the client to give the client the opportunity to recover from the error.

This topic provides the following information about handling exceptions:

- “Using appropriate exceptions” on page 20
- “Catching exceptions” on page 21

CORBA support, using appropriate exceptions

CORBA exceptions are used to communicate between server implementation objects and client applications. You must follow specific rules regarding which CORBA exceptions to use. The following abstract CORBA exception classes are defined:

“CORBA::Exception” on page

This is the abstract class that is the base of all CORBA exceptions. Because this class is abstract, it is never thrown. However, it can be used in catch blocks to process all CORBA exceptions in one block.

“CORBA::UserException” on page

This is the abstract class for all CORBA user exceptions and is a subclass of CORBA::Exception. This class should be used as the base class of all user-defined exception classes. The contents of these classes have no special format. Methods that throw these classes must declare their usage in IDL using the `raises` clause.

“CORBA::SystemException” on page

This is the abstract class for all CORBA standard exceptions and is a subclass of

CORBA::Exception. These exceptions can be thrown by any method regardless of the interface specification. Standard exceptions cannot be listed in raises expressions, therefore whether or not an interface throws a system exception is unknown. This means that you should be prepared to handle standard exceptions on all method calls. Each standard exception includes a minor code to provide more detailed information.

Any method can throw a standard exception, even if there are no exceptions declared in the `raises` clause of that method, so a method can throw an exception at any time.

Note: CORBA standard exceptions are a predefined list of exceptions that can be thrown from any method. CORBA has defined the class that provides this support as CORBA::System Exception. For more information about CORBA exceptions, see *The Common Object Request Broker: Architecture and Specification*.

CORBA support, catching exceptions

Client programs are required to handle exceptions, because the default behavior for uncaught exceptions is to end the process. (If the client process ends unexpectedly, suspect an uncaught exception.)

A client program can handle exceptions within the catch clause of a try/catch block that encompasses remote method invocations or calls to ORB services. Typically exception instances are actually instances of either the SystemException or UserException classes.

When deciding how or what exceptions to catch in a client application, consider the following general rules for exception handling:

- Perform as specific error recovery as makes sense. You can perform specific error recovery by proper structuring of `catch` clauses.
- Check for the most specific exceptions first, and most general exceptions last.
- Make use of information that is available in the exception. All CORBA exceptions support the .id() method that returns the exception identifier. System exceptions also provide .minor() and .completed() methods that return the minor code and completion status respectively.

Specific standard exceptions cannot be caught individually. If you need to handle individual standard exceptions, you can do so within a CORBA::SystemException catch block and use the .id() method.

Consider the following simple client example:

```
try
 {
    // Some real code goes here
    foo.boo();
 }
 // Catch and process specific User exceptions
 ...
 // Catch any other User exceptions defined for the method in the
 // `raises' clause
 catch (CORBA::UserException &ue)
 {
    // Process any other User exceptions. Use the .id() method to
    // record or display useful information
    cout << "Caught a User Exception: " << ue.id() << endl;
 }
 // Catch any System exceptions
 catch (CORBA::SystemException &se)
 {
    // Process any System exceptions. Use the .id(), and .minor()
    // methods to record or display useful information
    cout << "Caught a System Exception: " << ue.id() << ": " <<
         ue.minor() << endl;
 }
 catch (...)
 {
    // Process any other exceptions. This would catch any other C++
    // exceptions and should probably never occur
    cout << "Caught an unknown Exception" << endl;
 }
```

The CORBA server programming model

This topic describes the CORBA server programming model, which describes the interfaces and processes used to develop CORBA server objects that make up the business logic and business data inherent in a server application. Application programmers use the server programming model if they are developing CORBA server implementation objects, known as servant objects, that perform business functions used in the implementation of client objects.

The concepts about the server programming model are derived from the following general procedure for developing a CORBA server. The steps link to more detailed concepts. For task information about developing a CORBA server, see “Developing a CORBA server” on page 49.

Examples of server programming are given in the WSLoggerServer sample, for which files are included with WebSphere in the following directory: `WAS_HOME/Enterprise/samples/sampcppsdk`.

1. Specifying the business logic implementation interface for the servant (servant.idl).

   In an IDL (interface definition language) file, you define the public interface to the methods provided by the business logic. This defines the information that a client must know to call and use a servant object. For more information about the IDL definition of an implementation, see “Interface Definition Language (IDL), usage and implementation” on page 24.

2. Compiling the servant IDL (using idlc).

   Compiling the servant IDL file produces the usage binding files to implement and use the servant object within a particular programming language. For example, this creates an implementation template that provides a native, server language class template into which method behavior can be inserted. WebSphere supports CORBA servers implemented in C++.

3. Adding declarations for class variables, constructors, and destructors to the servant class definition (servant.ih).

   The implementation class interface header (servant.ih) created by idlc contains a skeleton class definition, but lacks declarations for class variables, constructors, and destructors. You need to add the missing declarations.

4. Completing the servant implementation (servant_I.cpp).

   The implementation class (servant_I.cpp) created by idlc contains a skeleton implementation definition, which you need to complete by adding the business logic that the servant is to provide.

5. Creating the server main code (server.cpp). You need to create the server code, to define the methods that the server implements. In particular, you need to create the main method, which controls the server runtime by performing the following tasks:

   1. Validating user input
   2. “Initializing the server environment” on page 23
   3. Accessing naming contexts
   4. Creating a servant object
   5. Binding the servant object to the appropriate naming context
   6. Creating a server shutdown object
   7. Going into a wait loop

8. Servicing requests

6. Building the server object and server code. Like anyother programming model, you need to build the modules that the server host can use to run the server and the servant.
7. Storing a logical definition for the server in the system implementation repository (using regimpl).

   Each server needs a unique logical definition in the implementation repository of the host on which the server is to run. The logical definition defines the server alias that is used to control the server.

Initializing the CORBA server environment

One of the first things that a CORBA server application needs to do when it is started is to initialize the server environment, to perform the following actions:

1. Getting a pointer to the implementation repository.

   The implementation repository is a persistent data store of ImplementationDef objects, each representing a logical CORBA server that has been registered in the repository. A server application typically gets a pointer to the implementation repository by using the CORBA::ImplRepository method; for example:

   ```
   ::CORBA::ImplRepository_ptr implrep = new ::CORBA::ImplRepository();
   ```

2. Getting a pointer to the ImplementationDef associated with the server alias.

   The ImplementationDef, which is obtained from the Implementation Repository, describes the server; for example, it specifies a UUID that uniquely identifies the server throughout a network. Each server must retrieve its own ImplementationDef object from the Implementation Repository (using the ImplRepository class), because the ImplementationDef is a parameter required by the BOA::impl_is_ready method. A server application typically gets a pointer to its ImplementationDef by using the CORBA::ImplRepository find_impldef or find_impldef_by_alias method; for example:

   ```
   imp = implrep->find_impldef_by_alias(argv[1] );
   ```

   Where argv[1] is the server alias specified as a string on the command used to start the server.

3. Initializing the communications protocol.

   This action sets the communication protocol that the server supports to SOMD_TCPIP in the ImplementationDef, using the following code extract:

   ```
   imp->set_protocols("SOMD_TCPIP");
   ```

4. Initializing the ORB and BOA.

   This action is used to initialize the ORB and BOA and to return a pointer to each.

   A server application initializes the ORB by calling the CORBA::ORB_init() method, which also returns a pointer to the ORB. (If necessary, this method creates a new instance of the ORB.) For example, the following code extract initializes the ORB and return a pointers to it:

   ```
   op = ::CORBA::ORB_init(argc, argv, "DSOM");
   ```

   Where argc and argv refer to the properties specified on the command used to start the server. On the CORBA::ORB_init() method you must specify `DSOM` after the parameter *argv*.

   A server application initializes the BOA by calling the CORBA::BOA_init() method on the

ORB. For example, the following code extract initializes the BOA and returns a pointer to it:

```
bp = op->BOA_init(argc, argv, "DSOM_BOA");
```

Where argc and argv refer to the properties specified on the command used to start the server. On the BOA_init() method you must specify `DSOM_BOA` after the parameter *argv*.

5. Registering the server application as a CORBA server.

   This action calls the CORBA::BOA::impl_is_ready method to initialize the server application as a CORBA server. This method initializes the server's communications resources so that it can accept incoming request messages and export objects. For example, the following code extract registers the server (with the alias specified on the command used to start the server):

   ```
   bp->impl_is_ready(imp, 0 );
   ```

   Note: The zero (0) value indicates that the server should not register itself with the somorbd daemon, because CORBA servers within WebSphere only support transient objects. This parameter is an IBM extension to the CORBA specification, and should specified only for lightweight servers of transient objects.

CORBA server shutdown objects

When a CORBA server is started, it initializes itself then calls the method execute_request_loop() specifying a blocking mode (::CORBA::BOA::SOMD_WAIT). This puts the server into an infinite wait loop, during which the ORB can transmit requests to and from the servant object hosted bythe server. Because the execute_request_loop() method never returns, the server can never terminate unless it is forced to. A server shutdown object makes it possible to terminate the server gracefully. The server creates a server shutdown object, giving it a string that is used to shutdown the server.

To stop the server, run the WSStopServer program (provided with WebSphere Application Server enterprise services), which tells the ORB to shut the server down. WSStopServer has the following command syntax:

```
WSStopServer server_alias
```

Where *server_alias* is the server alias (defined in the Implementation Respository).

Accessing naming contexts for a CORBA server

Before a CORBA server can create and make available a servant object, it must have a logical name space for the servant object to exist in. This logical name space is a naming context for servant objects. The server can create a new naming context within any location within the root naming context. For example, a server called *servant*Server could create a new naming context called *servant*Context into which the server binds the servant object. Optionally, this context could be located within a domain context, which in turn is located within the root naming context. (You can create a servant context with only the root naming context as its parent, or with one or more intermediary parent contexts.)

Interface Definition Language (IDL), usage and implementation

The interface to a class of objects contains the information that a caller must know to use an object, specifically, the names of its attributes and the signatures of its methods. In the CORBA programming model, the Object Management Group (OMG) Interface Definition Language (IDL) is the formal language used to define object interfaces independent of the programming language used to implement those methods.

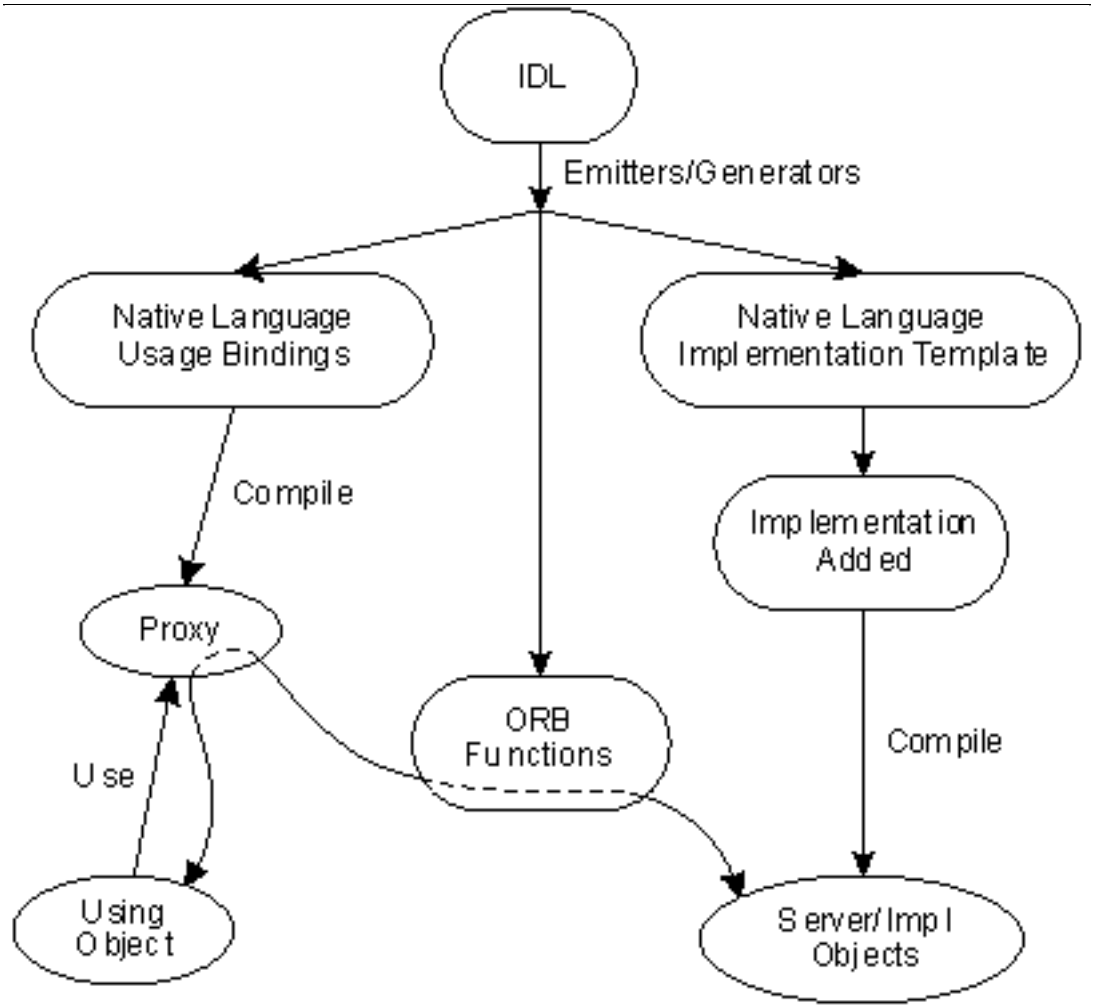

Figure 4 of 8. IDL, usage and implementation

"IDL, usage and implementation" on page 25 is an overview of the relationship between IDL and application development languages. Object providers use IDL to define the interfaces to their objects. The IDL can be directly defined by the object provider or can be produced transparently to the user in application development tools. Code emitters and generators produce the following elements:

- A usage binding that provides a native, client language rendering of the IDL, for example as a C++ class or Java interface. The usage binding is also used to generate a client stub object that through delegation maps the interface onto the server object providing the implementation.
- An implementation template that provides a native, server language class template into which method behavior can be inserted, for example, by editing the file and adding source code. The implementation of a class of objects (that is, the procedures that implement operations and the variables used to store an object's state) is written in the implementor's preferred programming language (for example, C++ or Java).
- Implementation objects such as skeletons and stubs may also be emitted and compiled if the client and server are in different processes or in different languages. These implementation objects provide the functions needed to make interlanguage calls and remote method execution.

The IDL compiler takes as input an IDL file and produces the usage binding files that make it convenient to implement and use objects that support the defined interface within a

particular programming language.

For an Enterprise JavaBean, you can create the IDL files from the bean's interface and home classes.

CORBA C++ client usage bindings

In these bindings, the client usage picture for the IDL types declared in the file T.idl appears as follows. Bold lines enclose files that are generated from IDL. Double lines enclose files that would normally be produced by a programmer or development tool.

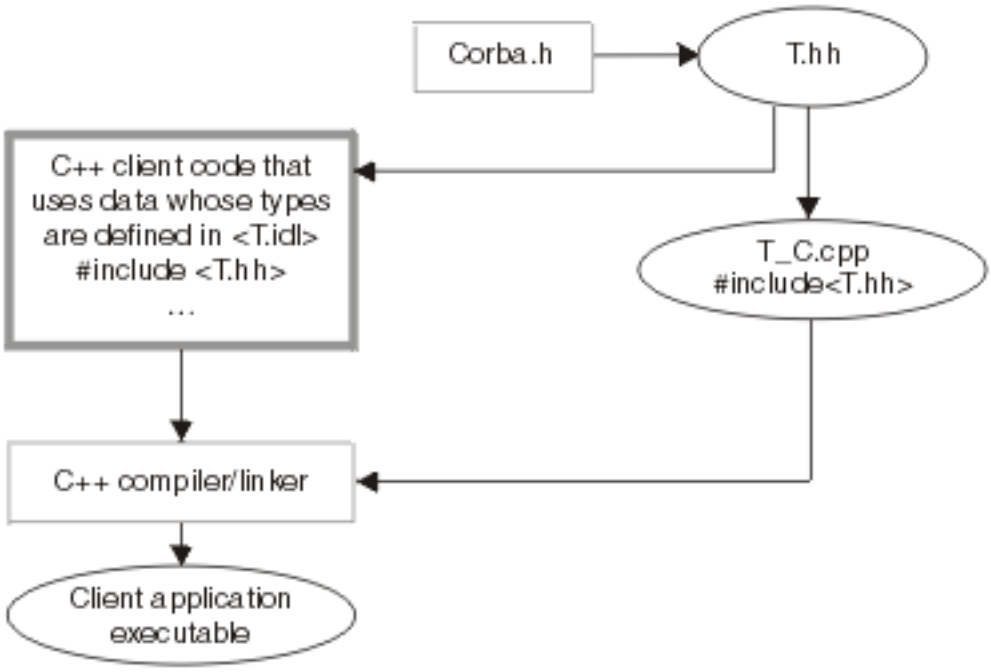

Figure 5 of 8. Client usage pictures for IDL types. The corba.h header file is included in the T.hh file, which is then included in the T_C.cpp file. The T_C.cpp file is then compiled, along with C++ client code that uses data whose types are defined in T.idl and included in T.hh. The final output is the client application executable.

The corba.h header file defines the C++ mappings for primitive IDL data types and other types required by the bindings, within a scope called CORBA. For more information about the scope see “IDL name scoping” on page . These types are implemented in a shared library that can be linked with a client application. A client application is created by compiling/linking emitted bindings and client code to produce an executable file.

The C++ bindings for the IDL types defined in the file T.idl are represented by a set of declarations in the emitted T.hh header file. The classes declared in T.hh that support client code are implemented by the code emitted into a corresponding T_C.cpp implementation file. The pair of files T.hh and T_C.cpp thus collectively provide the client bindings for T. (To minimize the number of generated files, some types used by servers are also declared in T.hh).

In general, the C++ bindings map non-primitive IDL types to C++ classes that implement constructors, destructors, assignment operators, and other functionality. Auxiliary classes are also sometimes defined, such as classes to automate storage management for array elements, sequence elements, and structure and union fields. The names of these auxiliary classes are not specified by CORBA, because specially-designed conversion operators and

copy constructors hide their existence from client code. These classes are not of interest to programmers that use the bindings.

CORBA C++ server usage bindings

To allow IDL interfaces to be implemented in C++, server-side bindings are emitted. The resulting classes work with the client bindings. The following figure illustrates the module structure of the server-side bindings, assuming that interface T is declared in the file T.idl.

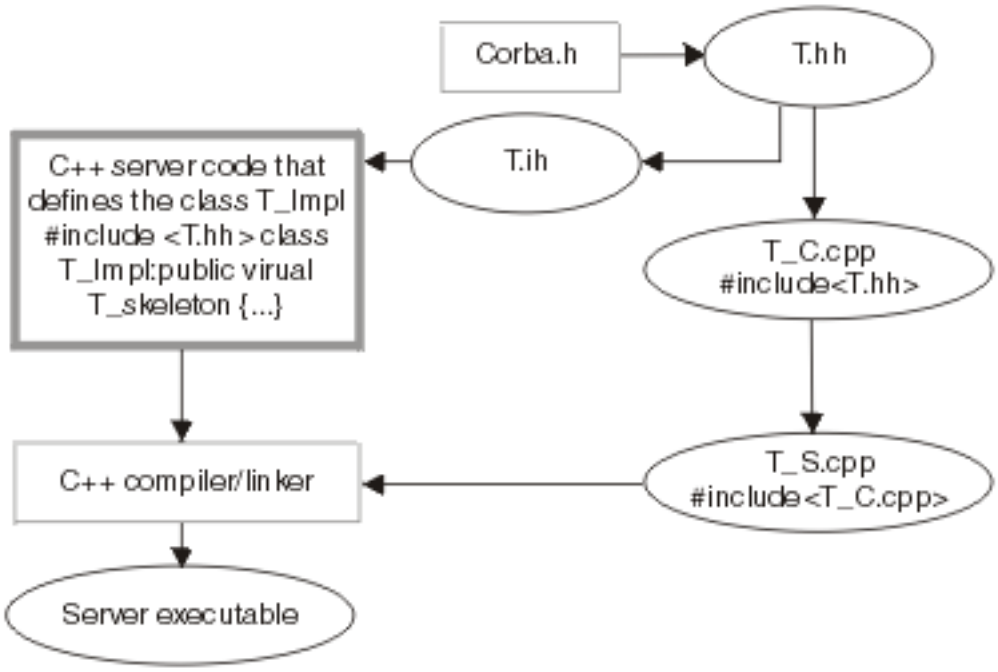

Figure 6 of 8. Module structure of server-side bindings. The corba.h header file is included in the T.hh file, which is then included in the T_C.cpp file. The T_C.cpp file is then included in the T_S.cpp file, which is then compiled, along with C++ server code. The C++ server code defines the class T_Impl, includes the implementation header file, T.ih (which includes the T.hh file), and defines the class T_Impl:public and the virtual T_skeleton. The final output is the server executable.

The differences between this figure and the client-side figure presented in Client C++ Bindings are:

- An emitted T_S.cpp file that provides server-side implementation bindings is compiled.
- Server-side C++ code (written by a programmer) defines the implementation for the operations introduced and inherited by the T interface.

The T_S.cpp file provides an implementation for the class T_Dispatcher. This class inherits from the dispatcher classes corresponding to T's parents. An instance of this class contains a T_ptr that addresses the T_Impl object upon which it will dispatch operation invocations. Each target object (for example, each T_Impl instance) exported by a server must have a corresponding dispatcher object, whose purpose is to receive a CORBA::Request object, determine what method is being invoked, stream the method arguments out into local variables, invoke the method on the target object, then stream the results back into the request so these can be returned to the caller.

The target object for a T_Dispatcher is an instance of the T_Impl class, which subclasses from (at least) the T_Skeleton class defined by the implementation bindings (in the file T.hh). The T_skeleton class inherits from the T interface class and the skeleton classes

corresponding to T's parents. As a result, T_skeleton inherits all the methods that T_Impl must support. Furthermore, this is done in a way that forces T_Impl to indeed provide implementations for all of these methods.

Take notice of the fact that the class name T_Impl is entirely arbitrary. The implementation class may have any name. Also note that the implementation class is not nested within any of the C++ classes that might be used to provide nesting scopes corresponding to IDL modules within which the interface T is defined. Thus, naming conflicts are a concern. A simple solution is to use underscores to concatenate module names with the name of the implemented interface. For example, if the interface T is defined within module M, then the implementation class name M_T_Impl can be used.

If the programmer responsible for T_Impl desires, the implementations (and supporting instance data) for any or all of T's parents can be inherited from their implementation classes, using C++ inheritance. Alternatively, T_Impl can provide its own implementation for the operations inherited into T. The image below graphically illustrates these options from an IDL snippet, using a dotted inheritance line to show optional C++ inheritance.

```
interface A
 {
   ...
 };
 interface B : A
 {
   ...
 };
```

Figure 7 of 8. Inherited implementation. Object_ORBProxy (a proxy implementation for CORBA::Object) is derived from objects A and B through five levels of inheritance, in a process which involves both pure (or skeleton) and proxy implementations of objects A and B at different levels.

The above figure focuses on C++ classes. The term pure is based on the use of this word in C++ to describe virtual functions that have no implementation (denoted in C++ by assigning a 0 to the name of the virtual function). Classes with pure virtual functions cannot be instantiated. Therefore, the skeleton classes require a subclass to provide complete implementations for all virtual functions in an interface. The dotted line in the previous figure indicates one way that B_Impl can provide implementations for the pure virtual functions inherited from A_Skeleton (using B_Skeleton). One way of viewing the skeleton classes is that they "turn off" proxy behavior and require subclasses to explicitly provide an alternative (non-proxy) implementation.

WebSphere CORBA value type library for C++

The Java Language to IDL specification maps Java serializables to CORBA value types. Therefore every Java serializable to be passed by a CORBA client as a parameter or return value for an Enterprise JavaBean must be re-implemented in the language of the client. Implementation of Java serializables as value types in C++ or another language can be a significant development effort.

To aid application development, WebSphere Application Server provides a valuetype library that contains C++ valuetype implementations for some commonly used Java classes in the java.lang, java.io, java.util, and javax.ejb packages. For example, Integer, Float, Vector, Exception, OutputStream, and so on. The valuetype library supports the WebSphere C++ ORB.

These classes represent an established hierarchy in the Java language and are implemented to preserve the inheritance relationship that exists in certain Java packages.

These classes enable CORBA programmers to use the WebSphere C++ classes in the same way they would use their Java counterparts. Constructors in the original Java classes do not need to be mapped to the IDL definitions and the C++ bindings; when mapped, constructors become init methods on the factory classes.

The IDL compiler always provides a pointer type definition for each type. For example, for a valuetype class T, `typedef T * T_ptr`. Unlike mapping for interfaces, the reference counting for valuetype must be implemented by the instance of the valuetypes. The IDL compiler also generates a _var class, which you can use instead of the _ptr. The _var class for a valuetype automates the reference counting; that is, it automatically manages the memory associated with the dynamically allocated object reference. When the T_var object is deleted, the object associated with T_ptr is released. When a T_var object is assigned a new value, the old object reference pointed to by T_ptr is released after the assignment takes place. A casting operator is also provided to allow you to assign a T_var to a type T_ptr.

For more information about the WebSphere CORBA value type library for C++, see the following topics:

- "C++ valuetype library, data type mappings" on page 29
- "C++ valuetype library, runtime type information" on page 30
- "C++ valuetype library, application programming interface" on page 30
- "C++ valuetype library, methods implemented" on page
- "C++ value type library, examples" on page 89
- "Creating your own C++ valuetypes" on page 69

C++ value type library, data type mappings

The WebSphere CORBA valuetype library for C++ provides mappings for the following primitive data types:

IDL Type	C++ Type
short	CORBA::Short
long	CORBA::Long
long long	CORBA::LongLong
unsigned short	CORBA::UShort
unsigned long	CORBA::ULong
unsigned long	CORBA::ULongLong
float	CORBA::Float
double	CORBA::Double
long double	CORBA::LongDouble
char	CORBA::Char
wchar	CORBA::Wchar
boolean	CORBA::Boolean
octet	CORBA::Octet

Note: The rmic -idl utility maps a Java byte to an IDL octet, and a Java char to an IDL wchar.

Objects behave somewhat differently, as shown in the following examples (Java type-> IDL

type-> C++ type):

- Array

```
byte[]-> ::org::omg::boxedRMI::seq1_octet-> ::org::omg::boxedRMI::seq1_octet*
```

- String

```
java.lang.String-> ::CORBA::WstringValue-> ::CORBA::WstringValue*
```

- Standard Java objects

```
Java.util.Enumeration-> abstract valuetype Enumeration-> ::java::util:: Enumeration
```

The IDL definition for the Enumeration valuetype (as generated by the rmic -idl utility) is:

```
module java {
  module util {
    abstract valuetype Enumeration;
  };
};
```

C++ value type library, runtime type information

Some of the classes in the WebSphere value type library contain methods that accept instances of a superclass. For such cases, the library use a C++ dynamic_cast to determine the type of the passed object; for example:

```
CORBA::Boolean equals(CORBA::ValueBase& arg0)
 {
    ...
        OBV_java::lang::Integer*  argInteger =
dynamic_cast<OBV_java::lang::Integer*>(& arg0) ;
        ...
  }
```

This functionality allows you to perform type inquiries just as you would in Java using the *'instance of'* operator.

Note: However, for this code to work, a polymorphic hierarchy must exist (that is, at least one virtual function must be implemented in the class hierarchy).

Another possible approach is to use the *'typeid()'* operator of the type_info class; for example:

```
#include <typeinfo>
#include <iostream>
using namespace std;
class Test1 { __ };
class Test2 : Test1 {_..};
void main(void)
{
Test2* ptr = new Test2();
cout << typeid(*ptr).name() << endl; //yields the string "class Test2"
}
```

Depending on the compiler that is used, you must enable certain options in order for this functionality to work properly. For example, for MSVC++ the /GR option must be added to the compiler settings.

C++ value type library, application programming interface

The WebSphere valuetype library for C++ implements the methods listed in “C++ valuetype library, methods implemented” on page . Because the implemented classes are derived from generated classes, the member functions they contain differ slightly from those in the java classes. For example, in java the class `java.io.FilterOutputStream` extends the abstract class `java.io.OutputStream`, so it must provide definitions for all abstract methods specified in the superclass. However, in the valuetype library hierarchy `java_io_FilterOutputStream_Impl` is derived from `java_io_OutputStream_Impl`; a concrete class that defines the methods of the generated class `::java::io::OutputStream`.

The types used in the signatures of these methods are derived from the OMG Specification.

The semantics of each of the valuetype methods conforms exactly to those of their Java counterparts. For a more detailed function specification of each method, see Sun's javadoc.

For each valuetype, there is a corresponding factory class. You should use the creation methods of a factory class (class name with _init or _factory suffix) to create instances of a valuetype (unlike the normal practice of using constructors to create objects in Java). Except in two cases, each creation method of the factory classes corresponds to a constructor in the Java counterparts of the valuetypes.

In addition to the valuetype classes, a utility class called VtlUtil is defined to provide several common methods to print debugging messages, to handle exceptions, to get registered factory objects, and to make transformation between C++ strings and the ::CORBA::WstringValue objects.

Note: You can reuse a registered factory object with the com::ibm::ws::VtlUtil::getFactory() method instead of creating a new factory every time.

The vtlib.h header file contains the definitions of all the factory classes and the VtlUtil class. These classes are defined in the com::ibm::ws name space.

For an example of using a registered factory object, the com::ibm::ws::VtlUtil::getFactory() method, and the creation methods of a factory, see “C++ valuetype library, examples” on page 89.

WebSphere Enterprise JavaBeans as clients of 3rd-party CORBA ORBs

Enterprise JavaBeans running in WebSphere Application Server can act as clients to CORBA servers running on third-party ORBs. The following articles describe the concepts of such an EJB application and cover the inclusion of CORBA invocations in an Enterprise JavaBean:

- “The CORBA components” on page 32describes in general terms the CORBA components of the EJB application: the interfaces, their use by the client, and their implementation by the server.
- “The CORBA interfaces” on page 32describes the CORBA interfaces in more detail.
- “Writing WebSphere Enterprise JavaBeans that act as CORBA clients ” on page 34 describes the CORBA-related tasks in calling CORBA servers from an enterprise bean.

The discussion here is necessarily general, particularly about 3rd-party CORBA servers. Servers written for third-party ORBs must take into account the differences between the ORBs, including supported services, administrative concerns, and other issues. In general, the structure of the servers remains constant, so this discussion is mostly structural in nature.

Because 3rd-party CORBA servers vary, so must the clients that use them. The CORBA calls integrated into an enterprise bean are determined, to some degree, by the requirements of the server and its ORB. Again, the structure remains relatively constant, but the details vary. The structure of the code for the enterprise bean still follows the standard EJB programming model; the type of work the code must perform to be an enterprise bean is unchanged. The use of CORBA calls to another server is in addition to the bean code, and this document concentrates on that addition.

Readers are assumed to be familiar with the writing of enterprise beans. For detailed information on that topic, see “Writing Enterprise Beans in WebSphere” in the WebSphere Application Server Advanced Edition infocenter. .

WebSphere Enterprise JavaBeans as CORBA clients, the CORBA

components

This article describes the main CORBA components for interoperation between enterprise beans running within WebSphere Application Server and CORBA servers running on third-party ORBs and uses a sample application to illustrate the description. The application consists of the following standard components:

The interfaces

The sample application provides two CORBA IDL interfaces. The Primitive interface exercises the primitive CORBA data types, for example, characters, octets, integers, and floats, and the Complex interface exercises the CORBA data types any and Object. The methods in each interface are designed to test the interoperability of that data type. The client sends the server a value; the server modifies the value in a predictable way and returns the value to the client. The client checks the returned value against a locally computed expected value. If they match, then the data type has been successfully transferred between the WebSphere and third-party ORB environments. For more information about the interfaces, see "WebSphere Enterprise JavaBeans as CORBA clients, the CORBA interfaces" on page 32.

The server

The server is a CORBA server running on a third-party ORB. It implements the methods in the two CORBA interfaces in servant objects and makes the servant objects available for clients to use. The implementations are very simple: when the client sends a value, the server modifies it in a predictable way, for example, adding a fixed value to an integer or negating a Boolean, and returns the new value to the client. For more information about the server and its implementation, see "Writing CORBA servers for third-party ORBs" on page 34.

The client

The client using the CORBA servant interfaces is an enterprise bean (a stateless session bean) running in WebSphere Application Server. On the request of its own client, the enterprise bean establishes contact with the CORBA server and invokes methods in one of the CORBA interfaces. Typically, it issues one call to each method and compares the returned results with the expected results. To compute the expected results, the client locally does the same work as the server, using the same code. If the client receives the same value from the server, then the two applications are interoperating properly for the data type being tested. For more information on the server and its implementation, see "Writing WebSphere Enterprise JavaBeans that act as CORBA clients " on page 33.

WebSphere Application Server provides many sample implementations of interoperation with 3rd-party ORBs. These articles use only one representative implementation and discusses the pieces of the client and server relevant to the interoperability issue. It does not discuss the standard parts of the application; for example, the EJB programming model. Writing CORBA servers is discussed in general terms, but the detail is beyond the scope of these articles and the specifics vary with the ORB being used. Although code from a particular implementation is used to illustrate the discussion, it is intended only to help clarify the discussion. It is not expected to work as source code for every ORB.

Additionally, the sample application is used simply as a framework for illustrating the issues surrounding interoperability. The details of the application design and implementation are not discussed. In some cases, the code extracts are distilled from several methods to make clear the logical flow of events rather than to represent the actual call sequence in one specific implementation.

WebSphere Enterprise JavaBeans as CORBA clients, the CORBA interfaces

The following example defines two CORBA interfaces. These interfaces, written in the CORBA interface-definition language (IDL), define methods to test the interoperability of CORBA data types between an enterprise bean running in the WebSphere Application Server and a CORBA server running on a third-party ORB. The Primitive interface defines the methods for testing the primitive CORBA data types, for example, characters, octets, integers, and floats. The Complex interface defines similar methods for the CORBA data types any and Object.

For each data type, there are three methods; each one returns the value in a different way. For example, the Primitive interface defines the following methods for a short integer:

```
...
interface Primitive
{
   const string serviceName = "primitive";
   const string testShortName = "short";
   short testShortIn(in short argin);
   void testShortOut(in short argin,
                     out short argout);
   void testShortInOut(inout short arginout);
   // Parallel methods for other data types
   ...
}
...
```

In the first method, the client sends the value in by using an `in` argument, and the server returns the result as the value of the method. In the second method, the client sends the value in the first argument, and the server returns it in by using the `out` argument.. In the third method, the client sends the value in by using an `inout` argument, which the server modifies to return the value. For each data type in each interface, a similar set of methods exists. The methods in the Complex interface are structured similarly.

Writing WebSphere Enterprise JavaBeans as clients of a 3rd-party CORBA ORB

Enterprise JavaBeans hosted by WebSphere Application Server can act as clients of CORBA servers, including servers running on third-party ORBs. These EJB clients follow the same design model as any other enterprise beans. In general, a minimal enterprise bean consists the following developer-provided code:

- A home interface, which provides bean-creation methods
- A remote interface, which defines the business methods available to the clients of the bean
- The bean class, in which the methods required by the Enterprise JavaBeans specification are implemented, including implementations of the methods required by the home and remote interfaces.

During deployment, additional code is generated for the container in which the bean resides. This code is specific to the container, for example, WebSphere Application Server, and is completely independent of any calls the bean makes to non-WebSphere-based servers.

When the enterprise bean acts as a client to a server, the code supporting that exchange resides in the implementation of the business methods of the bean. This includes the code for getting a reference to the client-side WebSphere ORB, contacting the server or desired servant object, and calling the methods in the CORBA IDL interface between the enterprise bean and the CORBA server.

Typically, the least straightforward part of using a CORBA server on a third-party ORB from an enterprise bean in the WebSphere environment is establishing the connectivity between the enterprise bean and the server. After connectivity has been established, making remote invocations is simply a matter of calling the methods in the CORBA IDL interface.

The technique used to establish connectivity between the enterprise bean and the server depends on the design of the application and the techniques supported in common between

the two ORBs. Common techniques include the following:

- Having the enterprise bean contact the name service used by the remote ORB and look up the desired servant object.
- Using a client-side property to hold an initial object reference (IOR) to the name service of the remote ORB. The client reads the property and uses the IOR to contact the name service, from which it can look up the desired servant object.
- Having the server write a file that contains the name-service IOR. The client reads the file and uses the IOR to contact the name service, from which it can look up the desired servant object.
- Having the server write a file that contains the IOR to the servant objects. The client reads the file for the desired object and uses the IOR to contact the object directly.

Writing CORBA servers for third-party ORBs

CORBA servers can be written for ORBs other than the ORB provided with WebSphere Application Server. These servers follow the same general structure as a server written for the WebSphere ORB, but differences between ORBs mean differences in the details. Writing a server for a specific third-party ORB requires familiarity with the ORB; such a server is not a WebSphere-based application at all.

In general, a minimal CORBA server comprises the following code:

- A main routine, which typically does the following:
    - Initializes the ORB within the server
    - Establishes a way for clients to contact the server
    - Establishes a way for the clients to locate the servant objects
    - Puts the server into a listening state, ready to accept requests from clients
- Servant classes, which implement the interfaces between the client and server

Each server also requires some amount of application-specific code; this can include initialization code, helper methods, and a limitless range of other material. The techniques used in a particular server written for a specific ORB are determined by the needs of the application and by the ORB itself.

The methods used to initialize the ORB and to put the server into the listening state are usually standard CORBA methods, but there can be variations across ORBs and across versions of the same ORB. The mechanisms used by clients to locate servers and implementation objects vary with the design of the application and with the naming facilities supported in an environment. Two common choices are to have the server register naming contexts for itself and its servant objects in a CORBA name service, from which the client can retrieve them, and to have the server register its naming contexts and write them to files as interoperable object references (IORs), which the client retrieves and uses to re-create the naming contexts.

After the server is written, it must be compiled and run. The steps involved in compiling and running a server written for a third-party ORB depend heavily on the requirements of the ORB, the platform on which the server runs, the language in which the server is written, and the design of the application.

Problem determination

When running, WebSphere Application Server enterprise services records information about its activities in its activity log. The activity log on each host captures events that show a history of the enterprise services' activities on that host. Some of the entries in the log are

informational and others report on system exceptions, such as returned CORBA exceptions.

If you encounter enterprise services runtime errors, it is often useful to read the activity log and try to diagnose the problem yourself. After this, if you still need assistance from IBM to help you diagnose problems, you can provide the formatted activity log output to your IBM service personnel.

There is one activity log for each host machine. By default, the activitycpp.log file resides in the WebSphere Application Server enterprise services' service subdirectory, *wasee_install*\services; where *wasee_install* is the directory into which you installed WebSphere Application Server enterprise services on the host. You can specify an alternative location on the com.ibm.CORBA.activityLogDirectory property in the properties file identified by the SOMCBPROPS environment variable. You can also specify the maximum size of the activity log on the com.ibm.CORBA.activityLogMaxSize property. The activity log file is created automatically when the first log entry needs to be written.

Because the activity log is an accumulation of information, it always contains extraneous data. Some activity log entries report serious failures, but many of them only report on the execution of activities, expected exceptions, or warnings of potentially dangerous situations. For example, in most instances, lower level components write an entry in the activity log when they decide to throw an exception, even when the caller of the lower level component is prepared to handle the exception and continue processing on a normal code path. Although all these entries on activities, handled exceptions, and warnings can make it difficult to read the log, sometimes they do provide useful data to help you determine the exact cause of the problem that you are diagnosing.

Before you can read the contents of an activity log, you must format the log file. For more information about formatting an activity log, see “Formatting an activity or trace log” on page 78.

For more information about the data fields of entries in the activity log, see “Fields in a formatted activity log entry” on page .

If you need to do low-level debugging of problems identified in the activity log, you can turn on tracing for appropriate components then format and study the detailed information generated.

Note: Trace is used by or for IBM service personnel to assist in collecting data in possible defect situations. This support should only be used under direction of IBM service personnel. Incorrectly setting trace attributes for objects can result in performance degradation for normal operation.

If you turn on tracing for a component type, extra detailed information is recorded in one or more trace logs for the host. Multiple trace files can be generated if needed. By default, the trace log files are stored in the WebSphere Application Server's service subdirectory. You can specify an alternative location on the com.ibm.CORBA.traceLogDirectory property in the properties file identified by the SOMCBPROPS environment variable. You can also specify the maximum size of a trace log on the com.ibm.CORBA.traceLogMaxSize property. Trace log files are created automatically, and have the following name format: `yydddhhmmss.xxx` format where:

yy
is the year

ddd
is the Julian date

hh
is the hour

mm

is the minutes

xxx

is a three digit number between 101 and 999. After 999, the number rolls over to 101.

Before you can read the contents of a trace log, you must format the log file. You can also merge and format multiple log files into a single output file, sort and display the contents of trace logs in various groupings. For more information about formatting a trace log, see "Formatting an activity or trace log" on page 78.

Hints and tips: The activity log

In most problem determination situations, you need to quickly pinpoint the activity log entries related to the problem that you are investigating. One way to do this is to reduce the activity log to a more manageable size. Here are some ways to reduce the size of the activity log:

- "Setting the size of the activity log" on page 36 "Creating smaller activity logs" on page 36 "Naming formatted logs" on page 36

Setting the size of the activity log

Before starting client or server processes on a host, set the com.ibm.CORBA.activityLogMaxSize runtime property to the desired number of Kbytes. You can use 15K for testing robustness and 50K for long runs.

Note: The activity log wraps when it is full.

For more information about specifying runtime properties, see "Specifying properties for C++ CORBA clients and servers" on page 68.

Creating smaller activity logs

Smaller activity logs may speed up your problem determination process. If the run-time error can be reproduced by rerunning your application, consider performing the completing steps to create a set of small activity logs:

1. Format your last activity log into a file and save it.
2. Delete the activity log. Rerunning the application with a new activity log minimizes the extraneous information in the log.
3. Restart the clients and servers on the host.
4. When the clients and servers have started, run the showlog command to format the activity log to the output file*log1* then delete the activity log.
5. Run your test.
6. After the test, run the showlog command to format the activity log to the new output file *log2*.

You now have a set of small, formatted activity logs. For example, if *log2* shows that a client could not find a factory, *log1* shows you why that factory was not registered.

Naming formatted logs

Consider giving representative file names for the formatted activity logs; for example, you can specify the following showlog command:

```
showlogcpp activitycpp.log -d > serverStartup.980808.firstrun.out
```

An overview of basic CORBA concepts

A CORBA environment is based on client applications finding and using objects that provide a desired function. The objects typically represent something in the real world, for example shopping carts, and are hosted by servers (typically CORBA servers or EJB servers). The type of object is defined by it's interface and the semantics defined for that interface. There can be many instances of an object (with the same interface and semantics), but representing different entities. CORBA provides the Interface Definition Language (IDL) to define object interfaces, and ORBs to provide access to objects through a distributed environment. The binding of an object's interface to a specific implementation is handled in the server environment

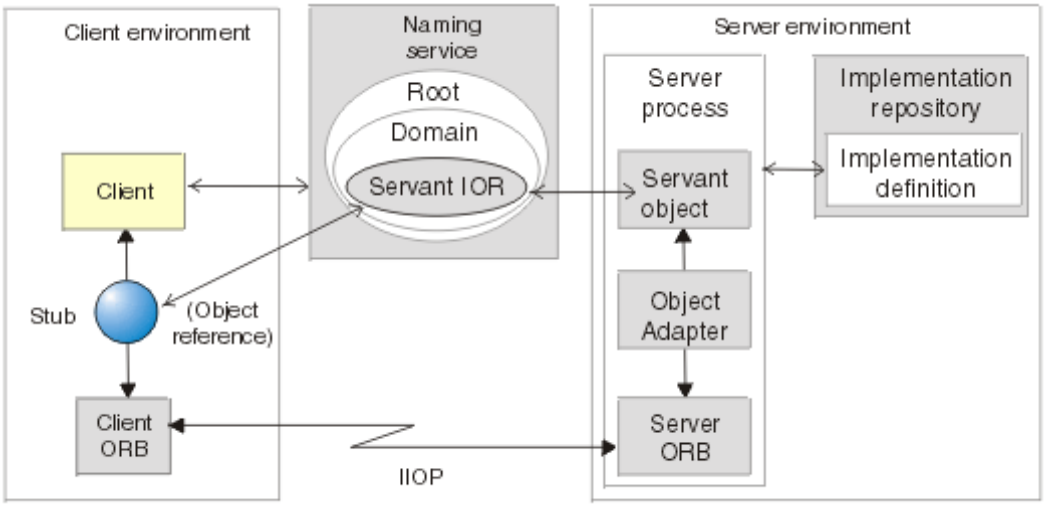

Figure 8 of 8. Conceptual view of a C++ CORBA environment

A CORBA environment comprises the following elements:

Client environment

The client runtime environment enables CORBA client applications to access server implementation objects. The client environment can be any of the following:

- A WebSphere CORBA C++ client or server (written with the CORBA C++ SDK)
- Java code in a WebSphere J2EE Client or Application Server
- A non-IBM-ORB-based CORBA client or server.

Client programming languages

WebSphere CORBA clients can be developed in C++. Also, Enterprise JavaBeans hosted by a WebSphere EJB server, as a CORBA client, can access server implementation objects hosted by CORBA servers. 3rd-party CORBA clients can be developed in C++, ActiveX, Java, or other languages supported by the CORBA client programming model. IDL language-bindings include COBOL.

Object

From the client's point of view, a CORBA object is an entity with an object reference that provides the operations defined in its interface. These operations are always available to the client, from the time the object instance is created until the time it is destroyed.

Client stub

To the client, the object on the server appears as if it resides in the client program. This is accomplished by using a client-side stub object. The stub object has the same interface as the server-side object it represents, but does not directly implement the object's methods. Instead, the stub object translates a method invocation into a format that is communicated over the ORB to the server. The

server then finds the target servant object, which executes the actual method implementation.

When a client wants to call a method on a server object, it creates an object reference, a pointer to the servant object, and stores it in the stub object.

Server implementation (servant) object

The server implementation object (also known as a servant object) is visible only to the server. The servant implementation is the executing CPU and memory resource that performs an object's operation. The client only knows that it invoked an operation defined on the object's IDL interface, and the response came back.

Object reference

An object reference contains information that is used to identify a target object. For example, a client-side stub object (as an object reference) contains information to locate the target server and the target servant object within that server.

An interoperable object reference (IOR) is a distributed object pointer that allows objects to communicate across network boundaries. IORs provide a platform- and vendor-independent object reference. The representation of an IOR depends on where it is used. For instance, it is represented in a wire-level message format when it is being sent between ORBs and in an object format when it is stored in a stub object. A client can convert an IOR into a string, save it to a file, and then terminate. When the client is activated again, the IOR can be read from the file, and then converted back into an object reference.

CORBA-compliant Object Request Broker (ORB)

The CORBA-compliant ORB enables clients to communicate with the application server. The ORB sends local client requests across a network by using the Internet Inter-ORB Protocol (IIOP), which is a TCP/IP-based communications protocol with CORBA-defined message exchanges. Separate ORBs resides at each end of the communication channel.

Object adapter

The object adapter acts as a mediator between the communications framework of the ORB and the servant objects that reside on a server. When the server-side ORB receives a request, the ORB passes the request to the object adapter. The object adapter analyzes each request received by the ORB and dispatches it to the servant object that is the target for the request. The object adapter class provides methods that allow the server application to participate in the exporting and importing of object references and the selection of threads on which remote requests are dispatched.

Server

The server provides the runtime environment in which a servant object can exist. For example, it initializes the ORB and object adapter, creates a servant object, gives it a name in an appropriate naming context in the naming service. This makes it possible for the client to find and use the servant. The server puts itself into an infinite wait loop, during which the ORB can transmit requests to and from the servant object that the server hosts. If the server is shut down, it removes the servant object from the runtime environment and cleans up the resources used to support the servant object.

IIOP

The client and server ORBs communicate using the CORBA Internet Inter-ORB protocol (IIOP), which is a TCP/IP-based protocol with CORBA-defined message exchanges. CORBA uses the General Inter-ORB Protocol (GIOP) to define the format of messages and uses IIOP to map between GIOP messages and TCP/IP

messages. IIOP allows ORBs to communicate with each other and enables them to use the Internet for distributed object communication.

Naming service

The naming service forms the lookup directory for a distributed system. It provides an interface for binding and resolving names to object references. When an object is created, its object reference can be bound to a name in the naming service. Any other object with access to the naming service can use the name to return the associated object reference. The naming service implements the CosNaming service, the standard naming service defined by the CORBA services specification.

A CORBA client can use the Interoperable Name Service (INS) specification to call resolve_initial_references("NameService") on the client ORB to establish a root naming context for the naming service.

Interface Definition Language (IDL)

The CORBA Interface Definition Language (IDL) enables clients and servers to have a platform-independent and language-neutral standard on which to base their communications.

Using IDL, application developers can specify the public interface to a CORBA class or Enterprise JavaBean (as the servant class). For a CORBA server implementation, the application developer typically creates the IDL "by hand". For an Enterprise JavaBean a tool is used to create the IDL from the interface/class file. The IDL definition of a servant is used to generate the client stub. An IDL compiler generates the code necessary to use an interface with a specific programming language.

Serializable objects used in an Enterprise JavaBean's interface are expressed in IDL as CORBA valuetypes. Therefore every Java serializable to be passed by a CORBA client as a parameter or return value for an Enterprise JavaBean must be re-implemented in the language of the client. To simplify the development of CORBA clients of Enterprise JavaBeans, you should try to minimize the range of Java serializables used in the Enterprise JavaBean's interface.

Implementation repository

The implementation repository is a persistent data store of ImplementationDef objects, each representing a logical server that has been registered.

WebSphere Application Server takes care of the communications protocol, ORB, and object references. WebSphere Application Server supports RMI-IIOP. In addition, WebSphere Application Server implements JNDI using a Directory which also supports CORBA CosNaming bindings, making WebSphere Enterprise JavaBeans visible to CORBA clients.

CORBA support task articles

This part contains task topics about the CORBA support provided by WebSphere Application Server 4.0 enterprise services. These topics are intended to describe how you should complete tasks to enable and use the CORBA support.

- “CORBA support concept articles” on page 1
- “CORBA support example articles” on page 84
- “CORBA support reference articles” on page

Developing a C++ CORBA client

Use this task to develop a C++ CORBA client. This task generates and registers the client DLLs and the client-side usage bindings needed by C++ CORBA client programs to access an object class (Enterprise JavaBean or CORBA servant object) hosted by an application server.

To develop a C++ CORBA client, you complete the following steps:

1. Create the interface definition (IDL) files that specifies the public interface to the server implementation object class. If you want the client to access a CORBA server implementation class, you create the IDL file as part of the procedure to define the servant implementation, as described in “Defining the interface for a servant implementation (servant.idl).” on page 50. If you want the client to access an Enterprise JavaBean, you can create the IDL file from the bean class, as described in “Creating IDL files for an Enterprise JavaBean” on page 40 .
2. Use the idlc command to emit the client-side usage bindings from the IDL, specifying the option **-suc:hh**.
   If you want the client to access a CORBA server implementation, you emit the client-side usage bindings when you compile the servant.idl file, as described in “Compiling the servant IDL (using idlc)” on page 52. If you want the client to access an Enterprise JavaBean, you can use the same procedure with the IDL file created from the bean class.

   For example, to emit the client-side bindings from the Hello.idl file, at a command line change to the directory that contains the IDL file, then type the following command:

   ```
   idlc -suc:hh Hello.idl
   ```

   When the specified idl file is compiled successfully, the idlc command creates the binding files and returns a value of zero. For example, for the above example idlc command, the following binding files are created: Hello.hh and Hello_C.cpp
3. Create the main code for the client program, as described in, “Creating the CORBA client main code (client.cpp)” on page 41.

You can next compile and link the C++ client main program, as described in “Compiling a C++ client program” on page 47.

Creating IDL files for an Enterprise JavaBean

Use this task to generate the interface definition language (IDL) files that specify the interface to an Enterprise JavaBean. You can then use the IDL to create client-side usage bindings for CORBA clients to use the Enterprise JavaBean's interface. You need to complete this task only if you are developing a CORBA client that needs to access an Enterprise JavaBean.

To develop the IDL files for an Enterprise JavaBean, complete the following steps:

1. Develop the Enterprise JavaBean. For more information about developing Enterprise JavaBeans, see “4.3: Developing Enterprise JavaBeans” in the WebSphere Application Server Advanced Edition infocenter. .
2. Ensure that the JAR file that contains the Enterprise JavaBean class can be accessed by the rmic command; for example, the JAR file should be in the

system classpath.

3. Use the Java rmic -idl command to generate IDL files from the Enterprise JavaBean's remote and home interfaces.

   For example, to generate IDL files for the Enterprise JavaBean com.ibm.ejb.samples.hello.Hello, you could use the following command:

```
rmic -idl com.ibm.ejb.samples.hello.Hello com.ibm.ejb.samples.hello.HelloHome
```

   This step results in the class.idl and classHome.idl files. For example, for the above rmic command for the Hello Enterprise JavaBean class, created the following idl files: Hello.idl and HelloHome.idl.

You can use the IDL file to create the client-side usage bindings needed by a CORBA client, as described in " Developing a C++ CORBA client" on page 40 .

Creating the CORBA client main code (client.cpp)

Use this task to create the main code for a CORBA client, to locate a servant object hosted by a CORBA server and to call methods on the server object. The client's main method performs the following tasks:

1. Validating user input
2. Initializing the client environment
3. Getting a pointer to the root naming context
4. Accessing the servant object
5. Calling methods on the servant object
6. Stopping the client and releasing resources used

To create the main code for a CORBA client, complete the following steps:

1. Create a source file,*client*.cpp, where *client* is the name of the client program.
2. Edit the client source file, *client*.cpp, to add appropriate code to implement the client. To do this, complete the following steps:
   a. Add include statements and global declarations needed, as described in "Creating CORBA client main code (client.cpp), adding include statements and global declarations" on page 42.
   b. Add the main method, in the form:

```
main(int argc, char *argv[])
{
  int rc;
  ::CORBA::Object_ptr objPtr;
  ::CosNaming::NamingContext_var
rootNameContext = NULL;
  Servant_var liptr = NULL;
  exit( 0 );
}
```

3. Add code to check the input parameters provided on the command used to start the client, as described in "Creating CORBA client main code (client.cpp), adding code to check input parameters" on page 42.
4. Add code to initialize the client environment, as described in "Creating CORBA client main code (client.cpp), adding code to initialize the client environment" on page 43.
5. Add code to get a pointer to the root naming context, as described in "Creating CORBA client main code (client.cpp), adding code to get a pointer to the root naming context" on page 44.

6. Add code to access the servant object that has already been created by the server, as described in “Creating CORBA client main code (client.cpp), adding code to access the servant object” on page 45.
7. Add code to call methods on the servant object, as described in “Creating CORBA client main code (client.cpp), adding code to call methods on the servant object” on page 46.
8. Add code to shutdown the client and release resources used, as described in “Creating CORBA client main code (client.cpp), adding code to shutdown the client and release resources used” on page 46.

This task is one step of the parent task, “Developing a CORBA client” on page 40 .

Creating CORBA client main code (client.cpp), adding include statements and global declarations

Use this task to add the include statements and global declarations needed to the source file for a CORBA client main code. This task is one step of the parent task to create the CORBA client main code, as described in “Creating a CORBA client main code (client.cpp)” on page 41 .

To add include statements and global declarations to the source file for a CORBA client main code, edit the client source file, *client*.cpp to complete the following steps:

1. Add the following include statements:

```
#include "servant.hh"
#include <CosNaming.hh>
```

Where:

***servant*.hh**

Specifies the name of the client-side usage bindings file for the server implementation class, *servant*. This file is created when the server implementation class IDL is compiled, as described in “Compiling a CORBA server implementation class IDL (using idlc)” on page 52.

CosNaming.hh

Specifies the header file for the COSNaming functions.

2. Add the following global declaration:

```
static ::CORBA::ORB_ptr op;
```

Where:

::CORBA::ORB_ptr op

Declares a pointer to the ORB.

You can also add the code for the functions needed in the client main code, as described in the parent task “Creating a CORBA client main code (client.cpp)” on page 41 .

Creating CORBA client main code (client.cpp), adding code to check input parameters

Use this task to add code to the source file for a CORBA client, to check input parameters. This code is used to check the parameters that a user specifies when starting the CORBA client.

This task is one step of the parent task to create the CORBA client main code, as described in “Creating a CORBA client main code (client.cpp)” on page 41 .

This examples in this task are based on a CORBA client that is started by the following command:

```
client log_file_name iterations
```

Where:

client

is the name of the client program.

log_file_name

is the the full pathname of a log file used to record actions by the client.

iterations

is the number of times that the client code should be run when the client program is started.

The code checks that the command used to start the CORBA client specifies two arguments required.

To add code to check the input parameters to the source file for a CORBA client main code, complete the following steps::

1. Edit the client source file, *client*.cpp, and add the following code:

```
main(int argc, char *argv[])
 {
   int rc;
   ::CORBA::Object_ptr objPtr;
   ::CosNaming::NamingContext_var rootNameContext = NULL;
   servant_var liptr = NULL;
   if ( argc != 3 )
   {
     cerr << "Usage: client <log_file_name> <iterations>" << endl;
     exit( -1 );
   }
   else
   {
     cout << "Entered client with log file name = " << argv[1];
     cout << " and iteration count = " << argv[2] << endl;
   }
   if ( ( rc = perform_initialization( argc, argv ) ) != 0 )
     exit( rc );
```

   Where the client program name and arguments are as specified above.

This task adds code to check input parameters to the main method in the source file for a CORBA client.

You need to add code to the client source file to initialize the client environment, as described in “Creating a CORBA client main code (client.cpp), adding code to initialize the client environment” on page 43.

Creating CORBA client main code (client.cpp), adding code to initialize the client environment

Use this task to add an initialization method to the source file for a CORBA client. This code is used to perform the initialization tasks needed when the client is started.

The aim of the client initialization method is to complete the following tasks to initialize the ORB and object adapter.

This task is one step of the parent task to create the CORBA client main code, as described in “Creating a CORBA client main code (client.cpp)” on page 41 .

To add an initialization method to the source file for a CORBA client main code, edit the client source file, *client*.cpp, and add the following code:

1. Add an initialization method, and add a statement to the main method to call the new method, as shown in the following code extract:

```
int perform_initialization( int argc, char *argv[] )
```

```
{
  // Initialize the ORB.
  op = ::CORBA::ORB_init(argc, argv, "DSOM");
  }
  cout << "Initialized ORB" << endl;
  return( 0 );
}
...
main(int argc, char *argv[])
{
 ...
 if ( ( rc = perform_initialization( argc, argv ) ) != 0 )
    exit( rc );
}
```

Where:

perform_initialization(argc, argv)

Calls the initialization method of the client main code, to initialize the client environment. The method takes as argument the log file name and iteration count specified on the command used to start the client.

This task adds code to initialize the client environment for a CORBA client.

You need to add code to the client source file to enable the client to access naming contexts, as described in “Creating CORBA client main code (client.cpp), adding code to access naming contexts” on page 44.

Creating CORBA client main code (client.cpp), adding code to get a pointer to the root naming context

Use this task to get a pointer to the root naming context. This task adds a "get name context" method (for example, called get_name_context) to the source file for a CORBA client. This method is used to access the naming service and return a pointer to the root naming context. The method performs the following actions after the server environment has been initialized:

1. Getting a pointer to the naming service.
2. Getting a pointer to the root naming context.

This task is one step of the parent task to create the CORBA client main code, as described in “Creating a CORBA client main code (client.cpp)” on page 41 .

To add a get_name_context() method to the source file for a CORBA server main code, edit the server source file, *servant*Server.cpp, and add the following code:

1. Add the get_name_context() method, as shown in the following code extract:

```
// This method accesses the Name Service and then gets
 // the root naming context, which it returns;
 // the WSLogger context.
 ::CosNaming::NamingContext_ptr get_naming_context()
 {
   ::CosNaming::NamingContext_ptr rootNameContext = NULL;
   ::CORBA::Object_ptr objPtr;
   // Get access to the Naming Service.
   try
   {
     objPtr = op->resolve_initial_references( "NameService" );
   }
   // catch exceptions ...
   if ( objPtr == NULL )
   {
     cerr << "ERROR: resolve_initial_references returned NULL" << endl;
     release_resources( op );
     return( NULL );
   }
   else
     cout << "resolve_initial_references returned = " << objPtr << endl;
   // Get the root naming context.
   rootNameContext = ::CosNaming::NamingContext::_narrow(objPtr);
   if ( ::CORBA::is_nil( rootNameContext ) )
   {
     cerr << "ERROR: rootNameContext narrowed to nil" << endl;
     release_resources( op );
     return( NULL );
   }
   else
     cout << "rootNameContext = " << rootNameContext << endl;
```

```
  // Release the temporary pointer.
  ::CORBA::release(objPtr);
  return( rootNameContext );
}
```

Where:

rootNameContext

is the pointer to the root naming context.

This code gets a pointer to the naming service then narrows the pointer object to the appropriate object type and assigns it to the new pointer object called `rootNameContext`, performs some checks, then releases the original pointer object, `objPtr`.

2. **Optional:** Add a statement to the main method to call the new method, as shown in the following code extract:

```
...
  if ( ( rc = perform_initialization( argc, argv ) ) != 0 )
    exit( rc );
  // Get the root naming context.
  rootNameContext = get_naming_context();
  if ( ::CORBA::is_nil( rootNameContext ) )
    exit( -1 );
```

This step returns a pointer object, `rootNameContext`, to the root naming context.

You need to add code to the client source file to access the servant object created by the server, as described in “Creating CORBA client main code (client.cpp), adding code to access the servant object” on page 45.

Creating CORBA client main code (client .cpp), adding code to access the servant object

Use this task to add code to the source file for a CORBA client, to get access to the servant object that has already been created by the CORBA server and bound into the name space. The client code gets access to the servant object by creating a ::CosNaming::Name that specifies the full name of the object from the root naming context.

For the example code in this task, the CORBA server created the servant object called *servant*Object1 in a new context, *servant*Context, that is bound to the domain naming context, which in turn is bound to the root naming context. Therefore the full name for the servant object, from the root naming context, is domain.*servant*Context.*servant*Object1.

This task is one step of the parent task to create the CORBA client main code, as described in “Creating a CORBA client main code (client.cpp)” on page 41 .

To enable the client to get access to the servant object, edit the client source file, *client*.cpp, and add the following code:

1. Add code to the main method to get a new ::CosNaming::Name for the servant object and to look up the servant object with that name in the name space:

```
// Get the root naming context.
 rootNameContext = get_naming_context();
 if ( ::CORBA::is_nil( rootNameContext ) )
   exit( -1 );
 // Find the servant_Impl created by the server. Look up the
 // object using the complex name of domain.servantContext.servantObject1,
 // which is its full name from the root naming context, as created
 // by the server.
 try
 {
   // Create a new ::CosNaming::Name to pass to resolve().
   // Construct it as the full three-part complex name.
   ::CosNaming::Name *servantName = new ::CosNaming::Name;
   servantName->length( 3 );
   (*servantName)[0].id = ::CORBA::string_dup( "domain" );
   (*servantName)[0].kind = "";
   (*servantName)[1].id = ::CORBA::string_dup( "servantContext" );
   (*servantName)[1].kind = ::CORBA::string_dup( "" );
   (*servantName)[2].id = ::CORBA::string_dup( "servantObject1" );
```

```
    (*servantName)[2].kind = ::CORBA::string_dup( "" );
    ::CORBA::Object_ptr objPtr = rootNameContext->resolve( * servantName );
    liptr = servant::_narrow( objPtr);
    cout << "After narrow, liptr = " << liptr << endl;
  }
  // catch exceptions ...
```

This task adds code that enables a CORBA client to find the specified servant object (created by a CORBA server) in the system name space.

You need to add code to the client source file to enable the client to call methods on the servant object, as described in "Creating CORBA client main code (client.cpp), adding code to call methods on the servant object" on page 46.

Creating CORBA client main code (client .cpp), adding code to call methods on the servant object

After a CORBA client has got access to a servant object, the client can call methods on the servant object/ The methods depend entirely on the business functionality of the client, but have the following general syntax:

```
servant_pointer->method_name( arguments );
```

This task is one step of the parent task to create the CORBA client main code, as described in "Creating a CORBA client main code (client.cpp)" on page 41 .

For example,

```
...
liptr = servant::_narrow( objPtr);
...
cout << "Logging to file " << liptr->getFileName() << endl;
liptr->setFileName( argv[1] );
cout << "Now logging to file " << liptr->getFileName() << endl;
```

This code forms the main business functionality of the client; when you have added the method calls needed to your client code, the next stage is to add code to shut down the client and release the resources that it uses, as described in "Creating CORBA client main code (client.cpp), adding code to stop the client and release resources" on page 46.

Creating CORBA client main code (client.cpp), adding code to shutdown the client and release resources used

Use this task to create code for a CORBA client, to shut down the client and release the resources that it used.

This task is one step of the parent task to create the CORBA client main code, as described in "Creating a CORBA client main code (client.cpp)" on page 41 .

To create code to shut down a CORBA client, edit the client source file, *client*.cpp to complete the following steps

1. Add a release_resources method, as shown in the following code extract:

```
// This function deallocates resources used throughout the program.
void release_resources( ::CORBA::BOA_ptr bp, ::CORBA::ORB_ptr op )
{
  // Deallocate the various resources we have allocated.
  ::CORBA::release( bp );
  ::CORBA::release( op );
}
```

   This method is called at the end of the client's main method after the client has finished accessing the servant object. (You add a call to this method in the next step.) The method takes as input pointers to the BOA and the ORB. It releases the resources used by the client.

2. Add code to the main method to call the release_resources method (after the

client has finished accessing the servant objects), as shown in the following code extract:

```
   // Deallocate all resources.
   release_resources( bp, op );
   cout << endl << "Client COMPLETED" << endl;
   cout.flush();
   exit( 0 );
}
```

This task adds code that shuts down a CORBA client and releases the resources that it used.

Building a C++ CORBA client

This topic provides an overview of the task to build the code for a C++ CORBA client. The actual steps that you complete depend on the development environment that you use.

For example, if you are using the Microsoft C++ 6.0 compiler on Windows NT to build a C++ CORBA client, you can use the following commands:

1. At a command line, type the following command:

```
cl /nologo -c -GX /Z7 /c /nologo /MD /Od /X /DLL /Zi -DEXCL_IRTC
 -DSOMCBNOLOCALINCLUDES -D_MS_INC_DIR= msvc60_install\include
 -D_USE_NAMESPACE -DNO_STRICT -I msvc60_install\include
 -Imsvc60_install\include
 -Iwasee_install\include client.cpp
```

   Where

   msvc60_install

   is directory in which the Microsoft C++ 6.0 compiler is installed; for example, `d:\msvc60\vc98`.

   wasee_install

   is the directory into which WebSphere Application Server enterprise services is installed.

   client

   is the name of the C++ client main code file.

2. At a command line, type the following command:

```
 cl /nologo -c -GX /Z7 /c /nologo /MD /Od /X /Zi -DEXCL_IRTC
-DSOMCBNOLOCALINCLUDES
           -D_MS_INC_DIR=msvc60_install  -D_USE_NAMESPACE_USE_NAMESPACE
           -DNO_STRICT -Imsvc60_install\include -Imsvc60_install\include
           -Iwasee_install\include -I.  servant_C.cpp
```

   Where

   servant

   is the name of the server implementation object (servant) that the client is to access.

3. At a command line, type the following command:
   link /nologo /DEBUG /OUT:*client*.exe
   /DEFAULTLIB:\WebSphere\AppServer\Enterprise\lib\somosa1m.lib
   \WebSphere\AppServer\Enterprise\lib\somororm.lib
   \WebSphere\AppServer\Enterprise\lib\somsrvsm.lib *client*.obj *servant*_C.obj

This task is one step of the parent task, “Developing a CORBA client” on page 40 .

For more examples of building C++ CORBA client code (on several platforms) for WebSphere Application Server, see the samples article "Tutorial: Creating a user-defined C++ server and client" at `WAS_HOME/Enterprise/samples/sampeex/sampcppsdk/wsBuildLogger.htm`.

Developing a CORBA server

Use this task to develop a CORBA server to service requests for business functions used in the implementation of client objects. The instructions and code extracts provided in this task are based on the development of the WSLoggerServer sample, for which files are included with WebSphere in the following directory:
`WAS_HOME/Enterprise/samples/sampcppsdk`.

Developing a CORBA server involves developing a server implementation class (known as a servant) and a server, as described in the following steps:

1. Create and edit an IDL file, *servant*.idl, to specify the public interface to the servant object class; where *servant* is the name of the server implementation class.
   For more information about creating and editing an IDL file for the servant object class, see "Defining the interface for a servant implementation (servant.idl)." on page 50

   This steps results in a fully-specified servant.idl file.

2. Compile the servant IDL file,*servant*.idl, to produce the usage binding files needed to implement and use the servant object within a particular programming language.
   For more information about compiling an IDL file, see "Compiling the servant IDL (using idlc)" on page 52.

   This step results in the set of usage binding files required for the servant.idl file.

3. Add declarations for class variables, constructors, and destructors to the servant implementation header (servant.ih).
   For more information about adding declarations to an implementation header, see "Adding declarations to a CORBA servant implementation header (servant.ih)" on page 53.

   This step results in the servant implementation header file, servant.idl, that contains all the declarations for class variables, constructors, and destructors needed by the implementation.

4. Complete the servant implementation *servant*_I.cpp, to add the code that is to implement the servant business logic.
   For more information about completing the servant implementation, see "Adding code to a CORBA servant implementation (servant_I.cpp)" on page 54.

   This step results in the server implementation file, servant_I.cpp, that contains the code needed by the implementation to support the business logic.

5. Create the server main, *server*.cpp, to write the code for the methods that the server implements (for example, to perform initialization tasks and create servant objects).
   For more information about completing the servant implementation, see "Creating the server main code (server.cpp)" on page 55.

   This step results in the server main source file, server.cpp, that contains the main method and associated code needed to implement the server.

6. Build the server code, as described in "Building a C++ CORBA server" on page 67.

Storing a logical definition for a CORBA server in the system implementation repository

Use this task to register a CORBA server in the implementation repository. To register a CORBA server in the implementation repository, you need to identify the server's alias, the server application program, and the server object implementation class (servant) that the server implements. The information registered is used to activate the server process when the server is started, and thereafter to help clients to find the server that supports servants that they want to use.

To register a CORBA server in the implementation repository, you use the regimpl utility, as follows:

1. To register the server, *server_alias*, type the following command:

```
regimpl -A -i server_alias -p server_application -t SOMD_TCPIP
```

Where

server_alias

is the server alias by which the server is known

server_application

is the name of the application program that implements the server.

[Windows] The program name has the form `program.exe`.

[Unix] The program name has the form `program`.

This task results in the implementation repository containing an entry for the server and the server object implementation class that it supports.

For example, for a server object implemetation class called WSLogger, supported by the server application WSLoggerServer, you would use the following regimpl commands:

```
regimpl -A -i WSLoggerServer
```

This task is one step of the parent task, “Developing a CORBA server” on page 49 .

Defining the interface for a CORBA servant class

Use this task to define the public interface of a CORBA servant class that is to provide the business logic to be used by clients. This defines the information that a client must know to call and use servant objects of that class, and forms one stage of the tasks to develop a CORBA server or client.

To specify the public interface for a CORBA servant class, you create an IDL (interface definition language) file that contains an interface declaration:

1. Create an IDL file, *servant*.idl, where *servant* is the name of the server implementation class.

   This steps results in a fully-specified servant.idl file.

2. Edit the *servant*.idl file to add an interface definition.
   The interface definition declares the interface name (and optionally its base interface names), and the methods (operations), and any constants, type definitions, and exception structures that the interface exports.

   The following information is an overview of the format of an interface declaration, and provides links to the reference topics that provide details about parts of the IDL declaration and IDL syntax. For reference information about IDL interface declarations, and the component declarations that they can

contain, see “IDL interface declarations” on page .

An interface declaration has the following syntax:

```
interface  interface-name
 [: base-interface1, base-interface2, ...]
  {
   [constant declarations]
   [type declarations]
   [exception declarations]
   [attribute declarations]
   [operation declarations]
 };
```

interface-name

The name of the public interface for the servant object, this should match the servant class name.

[: *base-interface1*, *base-interface2*, ...]

The base-interface names for one or more parent interfaces from which this interface, *interface-name* , is derived.

You need to specify base-interface names only if this interface is derived from one or more parent interfaces. Each base interface is specified in the form `: interface_name` and can be named only once in the interface statement header. If you specify a base-interface name, you must also add an include statement for the base-interface IDL file to the top of the servant.idl file.

[constant declarations] and [type declarations]

An interface declaration can include constant declarations and type declarations as in C and C++, with some restrictions and extensions. For more information about these declaration types, see “IDL type and constant declarations” on page .

[exception declarations]

An interface declaration can include exception declarations, which define data structures to be returned when an exception occurs during the execution of an operation. Each exception declaration specifies a name and, optionally, a struct-like data structure for holding error information. For more information about these declaration types, see “IDL exception declarations” on page .

[attribute declarations]

Declaring an attribute as part of an interface is equivalent to declaring one or two accessor operations: one to retrieve the value of the attribute (a get or read operation) and (unless the attribute specifies readonly) one to set the value of the attribute (a set or write operation). For more information about these declaration types, see “IDL attribute declarations” on page .

[operation declarations]

Operation declarations define the interface of each operation introduced by the interface. (An IDL operation is typically implemented by a method in the implementation programming language. Hence, the terms operation and method are often used interchangeably.). For more information about these declaration types, see “IDL operation declarations” on page .

The order in which these declarations are specified is usually optional, and declarations of different kinds can be intermixed. Although all of the

declarations are listed above as optional, in some cases using one declaration can mandate another. For example, if an operation raises an exception, the exception structure must be defined beforehand. In general, types, constants, and exceptions, as well as interface declarations, must be defined before they are referenced, as in C or C++.

This task results in a fully-specified IDL file, *servant*.idl, that contains a declaration for the public interface to a servant class, *servant*.

For example, for a servant class called WSLogger, the IDL file, WSLogger.idl, was created and edited to add the following interface definition:

```
interface WSLogger
{
  void setFileName(in string newFileName);
  string getFileName();
  void setMethodName( in string newMethodName );
  string getMethodName();
  short openLogFile();
  short closeLogFile();
  short writeLogMessage(in string newMessage, in short newSeverity);
  enum mdyFormat { DMY_DATE_FORMAT,
                   MDY_DATE_FORMAT };
  void setDateFormat(in unsigned short newDateFormat);
  unsigned short getDateFormat();
};
```

You can next compile the servant.idl to create the usage bindings and other files needed to complete the implementation, as described in “Compiling the servant IDL (using idlc)” on page 52.

This task is one step of the parent task, “Developing a CORBA server” on page 49 .

Compiling a CORBA server implementation class IDL (using idlc)

Use this task to compile the IDL file, *servant*.idl, that defines the public interface for a CORBA server implementation class. You can also use this task to compile the IDL file (also referred to in this task as *servant*.idl) for an Enterprise JavaBean.

Note: If your servant.idl file references other IDL files, ensure that all those other IDL files can be accessed by the idlc program.

To compile the IDL file, *servant*.idl, you can use the idlc command to complete the following steps:

1. At a command line change to the directory that contains the IDL file,*servant*.idl, where *servant* is the name of the server implementation class.
2. Type the following command:

   ```
   idlc -ehh:ih:ic:uc:sc  servant.idl
   ```

   The options specified, and the files created are summarised in “Options for the idlc command” on page . The names of the generated output files are derived from the file name of the specified IDL file. For example, for the IDL file, *servant*.idl, the emitter option -ehh outputs the file *servant*.hh.

   This produces the files *servant*.hh, *servant*.ih, *servant*_I.cpp, *servant*_C.cpp, and *servant*_S.cpp.

This task creates the usage binding files needed to implement and use the servant object within a particular programming language.

For example, for a server object implemetation class called WSLogger, the IDL file, WSLogger.idl, was created and edited to add its interface definition. To compile the IDL file, the following command was used:

```
idlc -ehh:ih:ic:uc:sc WSLogger.idl
```

This created the following files: WSLogger.hh, WSLogger.ih, WSLogger_I.cpp, WSLogger_C.cpp, and WSLogger_S.cpp.

You can next add declarations for class variables, constructors, and destructors to the servant class definition, *servant*.ih, as described in "Adding declarations to the servant class definition" on page 53.

This task is one step of the parent task, "Developing a CORBA server" on page 49 . It can also be used to create the client-side bindings files needed to develop a CORBA client that is to access an Enterprise JavaBean, as described in "Developing a CORBA client" on page 40 .

Adding declarations to a CORBA servant class definition (servant.ih)

Use this task to add declarations for class variables, constructors, and destructors for a CORBA servant class to its skeleton implementation header file, *servant*.ih. This defines any private variables for the implementation code in the associated *servant*_I.cpp file.

This task follows on from the task to compile the *servant*.idl file that defines the public interface for the server implementation class. For more information about compiling the IDL file, which creates the *servant*.ih file, see "Compiling the servant IDL (using idlc)" on page 52 .

To add declarations for class variables, constructors, and destructors to an implementation header file,*servant*.ih, complete the following steps:

1. At a command line change to the directory that contains the *servant*.ih file, where *servant* is the name of the servant class.
2. Edit the implementation header file, *servant*.ih, to add appropriate declarations for class variables, constructors, and destructors. For more information about the types of declarations that you can add to an implementation header file, see "IDL type declarations" on page .

   For example, the idlc command `idlc -ehh:ih:ic:uc:sc -mdllname=WSLogger WSLogger.idl` converted the following interface declaration to the class declaration in the WSLogger.ih file. The WSLogger.ih file was then edited to add the extra declarations shown in bold.

Table: 1. Example: WSLogger interface and declarations added to the skeleton implementation header

Interface declaration in WSLogger.idl	Implementation header in WSLogger.ih
`interface WSLogger {` `  void setFileName(in string newFileName);` `  string getFileName();` `  void setMethodName( in string newMethodName );` `  string getMethodName();` `  short openLogFile();` `  short closeLogFile();` `  short writeLogMessage(in string newMessage, in short newSeverity);` `  const short DMY_DATE_FORMAT = 1;` `  const short MDY_DATE_FORMAT = 2;` `  void setDateFormat(in unsigned short newDateFormat);` `  unsigned short getDateFormat();` `};`	`class WSLogger_Impl : public virtual ::WSLogger_Skeleton {` `public:` `::CORBA::Void setFileName (const char* newFileName);` `char*  getFileName ();` `::CORBA::Void setMethodName (const char* newMethodName);` `char*  getMethodName ();`

Interface declaration in WSLogger.idl	Implementation header in WSLogger.ih
	`` ::CORBA::Short openLogFile (); ::CORBA::Short closeLogFile (); ::CORBA::Short writeLogMessage (const char* newMessage, ::CORBA::Short newSeverity); ::CORBA::Void setDateFormat (::CORBA::UShort newDateFormat); ::CORBA::UShort getDateFormat (); private: char * fileName; char * methodName; ::CORBA::UShort dateFormat; ofstream logFile; ::CORBA::UShort logFileOpen; public: WSLogger_Impl(char * newFileName); virtual ~WSLogger_Impl(); }; ```

You can next add code to skeleton implementation definition, *servant*_I.cpp, to implement the business logic, as described in “Completing the server implementation (server_I.cpp)” on page 54.

This task is one step of the parent task, “Developing a CORBA server” on page 49 .

Adding code to a CORBA servant implementation (servant_I.cpp)

Use this task to add code for a CORBA server implementation class to its skeleton implementation file, *servant*_I.cpp. The code defines the methods that implement the business logic for the server implementation class, *servant*.

This task follows on from the task to add declarations for class variables, constructors, and destructors to the servant implementation header file,*servant*.ih. For more information about adding declarations to an implementation header, see “Adding declarations to a CORBA servant implementation header (servant.ih)” on page 53 .

To add code to an implementation file,*servant*_I.cpp, to add the business logic that the servant is to provide, complete the following steps:

1. At a command line change to the directory that contains the *servant*_I.cpp file, where *servant* is the name of the server implementation class.
2. Edit the implementation file, *servant*_I.cpp, to add appropriate code to implement the business logic methods.

   For example, the idlc command `idlc -ehh:ih:ic:uc:sc -mdllname=WSLogger WSLogger.idl` converted the following interface declaration to the skeleton methods in the implementation file, WSLogger_I.cpp. The WSLogger_I.cpp file was then edited to add the code to implement the methods, with the code added for the `WSLogger_Impl::writeLogMessage` method shown in bold.

```
::CORBA::Void WSLogger_Impl::setFileName (const char* newFileName)
{
}
char*  WSLogger_Impl::getFileName ()
{
}
::CORBA::Void WSLogger_Impl::setMethodName (const char* newMethodName)
{
}
char*  WSLogger_Impl::getMethodName ()
{
}
::CORBA::Short WSLogger_Impl::openLogFile ()
{
}
::CORBA::Short WSLogger_Impl::closeLogFile ()
{
}
// This method writes one line of message text to the log file. The line
// prefaced with the current date and time in the currently specified
```

```
// format, the current method name (if any), the severity level, and
// the message text.
::CORBA::Short WSLogger_Impl::writeLogMessage (const char* newMessage,
::CORBA::Short newSeverity)
{
  ::CORBA::String_var timeString;
  if ( logFileOpen == FALSE )
    return( -1 );
  // Get the date and time string.
  time_t tp;
  time_t tp2;
  if ( ( tp = time(&tp2) ) != -1 )
  {
    struct tm *x = gmtime( &tp2 );
    timeString = ::CORBA::string_dup( ctime( &tp2 ) );
  }
  // Determine the day and month.
  ::CORBA::String_var day = ::CORBA::string_alloc( 3 );
  ::CORBA::String_var month = ::CORBA::string_alloc( 4 );
  day[0] = timeString[8];
  day[1] = timeString[9];
  day[2] = 0;
  month[0] = timeString[4];
  month[1] = timeString[5];
  month[2] = timeString[6];
  month[3] = 0;
  // Copy the time and year.
  ::CORBA::String_var time = ::CORBA::string_alloc( 14 );
  strncpy( time, (const char *) &timeString[11], 13 );
  time[13] = 0;
  // Output the time of the log message.
  if ( dateFormat == DMY_DATE_FORMAT )
    logFile << day << " " << month;
  else if ( dateFormat == MDY_DATE_FORMAT )
    logFile << month << " " << day;
  logFile << " " << time << "  ";
  if ( getMethodName() != NULL )
    logFile << getMethodName() << ", ";
  logFile << "severity " << newSeverity << ": ";
  // Output the log message.
  logFile << newMessage << endl;
  return 0;
}
::CORBA::Void WSLogger_Impl::setDateFormat (::CORBA::UShort newDateFormat)
{
}
::CORBA::UShort WSLogger_Impl::getDateFormat ()
{
}
```

You can next create the server main code (server.cpp), to implement the server, as described in “Creating a CORBA server main code (server.cpp)” on page 55.

This task is one step of the parent task, “Developing a CORBA server” on page 49 .

Creating the CORBA server main code (server.cpp)

Use this task to create a CORBA server that hosts a servant object. The server performs the following tasks

1. Validating user input
2. Initializing the server environment
3. Accessing naming contexts
4. Naming, creating, and binding a servant object
5. Creating a server shutdown object
6. Going into a wait loop
7. Servicing requests

This task follows on from adding code for the business logic methods in the servant implementation file,*servant*_I.cpp. For more information about adding code to a servant implementation file, see “ Completing the servant implementation (servant_I.cpp)” on page 54 .

To create the main code for a CORBA server, complete the following steps:

1. Create a source file,*servant*Server.cpp, where *servant* is the name of the implementation class for which the server is to service requests.

2. Edit the server source file, *servant*Server.cpp, to add appropriate code to implement the server methods. To do this, complete the following steps:

   a. Add include statements and global declarations needed, as described in “Creating CORBA server main code (server.cpp), adding include statements and global declarations” on page 56.

   b. Add the main method, in the form:

```
void main( int argc, char *argv[] )
 {
   ::CORBA::Object_ptr objPtr;
   ::CORBA::Status stat;
   int rc = 0;
 }
```

3. Edit the server source file, *servant*Server.cpp, to add appropriate code to check the input parameters provided on the command used to start the server, as described in “Creating CORBA server main code (server.cpp), adding code to check input parameters” on page 57.
4. Edit the server source file, *servant*Server.cpp, to add appropriate code to initialize the server environment, as described in “Creating CORBA server main code (server.cpp), adding code to initialize the server environment” on page 58.
5. Edit the server source file, *servant*Server.cpp, to add appropriate code to access naming contexts, as described in “Creating CORBA server main code (server.cpp), adding code to access naming contexts” on page 60.

   At this point initialization has been accomplished and a naming context created for servant objects.

6. Edit the server source file, *servant*Server.cpp, to add appropriate code to name, create, and bind servant objects, as described in “Creating CORBA server main code (server.cpp), adding code to name, create, and bind servant objects” on page 63.
7. Edit the server source file, *servant*Server.cpp, to add code to create a server shutdown object, as described in “Creating CORBA server main code (server.cpp), adding code to create a server shutdown object” on page 64.
8. Edit the server source file, *servant*Server.cpp, to add code to put the server into an infinite loop (to service any ORB requests received), as described in “Creating CORBA server main code (server.cpp), adding code to put the server into an infinite loop” on page 65.
9. Edit the server source file, *servant*Server.cpp, to add code to shutdown the server and release resources used, as described in “Creating CORBA server main code (server.cpp), adding code to shutdown the server and release resources used” on page 65.

This task is one step of the parent task, “Developing a CORBA server” on page 49 .

Creating CORBA server main code (server.cpp), adding include statements and global declarations

Use this task to add the include statements and global declarations needed to the source file for a CORBA server main code. This task is one step of the parent task to create the CORBA server main code, as described in “Creating a CORBA server main code (server.cpp)” on page 55 .

To add include statements and global statements to the source file for a CORBA server main code, edit the server source file, *servant*Server.cpp, and add appropriate statements:

1. Add appropriate include statements; for example:

```
#include "servant.ih"
#include "servershutdown.h"
#include <CosNaming.hh>
```

Where:

***servant*.ih**

Specifies the name of the implementation header file for the servant class, *servant*, to be hosted by the server.

***servershutdown*.h**

Specifies the name of the header file for the class used to shutdown the CORBA server.

CosNaming.hh

Specifies the header file for the COSNaming functions.

2. Add appropriate global declarations; for example:

```
// Global declarations:
static ::CORBA::ORB_ptr op;
static ::CORBA::BOA_ptr bp;
::CORBA::ImplementationDef_ptr imp ;
servant_var object_Impl;
```

Where:

::CORBA::ORB_ptr op

Declares a pointer to the ORB.

::CORBA::BOA_ptr bp

Declares a pointer to the BOA.

::CORBA::ImplementationDef_ptr imp

Declares a pointer to the ImplementationDef associated with the server alias.

***servant*_var *object*_Impl**

Declares the servant object to be created later in the server main code.

You can add the code for the functions needed in the server main code, as described in the parent task “Creating a CORBA server main code (server.cpp)” on page 55 .

Creating CORBA server main code (server.cpp), adding code to check input parameters

Use this task to add code to check input parameters to the source file for a CORBA server. This code is used to check the parameters that a user specifies when starting the CORBA server.

This task assumes that the CORBA server is started by the following command:

```
servantServer server_alias
```

Where:

servant

is the name of the server implementation class that the server supports.

server_alias

is the server alias (defined in the Implementation Respository).

The code checks that the command used to start the CORBA server specifies a string, *server_alias*, the server alias. During the subsequent server initialization function called when the server starts, the server alias is used to retrieve the server's ImplementationDef; therefore, the string specified must match the server alias predefined in the system Implementation Repository.

This task is one step of the parent task to create the CORBA server main code, as described in “Creating a CORBA server main code (server.cpp)” on page 55 .

To add code to check the input parameters to the source file for a CORBA server main code, complete the following steps::

1. Edit the server source file, *servant*Server.cpp, and add the following code:

```
void main( int argc, char *argv[] )
{
  ::CORBA::Object_ptr objPtr;
  ::CORBA::Status stat;
  int rc = 0;
  // Validate the input parameters.
  if ( argc != 2 )
  {
    cerr << "Usage:  servant <server_alias>" << endl;
    exit( -1 );
  }
  if ( ( rc = perform_initialization( argc, argv ) ) != 0 )
    exit( rc );
  ...
}
```

Where:

server_alias

Specifies the server alias predefined in the system Implementation Repository.

perform_initialization(argc, argv)

Calls the initialization function of the server main code, to check that the server alias is defined in the system Implementation Repository (and to perform other tasks to initialize the server environment).

This task adds code to check input parameters to the main method in the source file for a CORBA server.

You need to add code to the server source file for the server initialization function, as described in “Creating CORBA server main code (server.cpp), adding code to initialize the server environment” on page 58.

Creating CORBA server main code (server.cpp), adding code to initialize the server environment

Use this task to add a server initialization method to the source file for a CORBA server. This code is used to perform the initialization tasks needed when the server is started.

The aim of the server initialization method is to complete the following tasks to initialize the server environment:

1. Getting a pointer to the Implementation Repository
2. Getting a pointer to the ImplementationDef associated with the server alias.
3. Initializing the communications protocol.
4. Initializing the ORB and object adapter.
5. Registering the server application as a CORBA server.

This task is one step of the parent task to create the CORBA server main code, as described in “Creating a CORBA server main code (server.cpp)” on page 55 .

To add a server initialization method to the source file for a CORBA server main code, edit the server source file, *servant*Server.cpp, and add the following code:

1. Add an initialization method, and add a statement to the main method to call the

new method, as shown in the following code extract:

```
// This function performs general initializtion, including retrieval
// of the appropriate ImplementationDef, setting the communications
// protocol, and initalization of the ORB and BOA.
int perform_initialization( int argc, char *argv[] )
{
  return( 0 );
}
void main( int argc, char *argv[] )
{
  ::CORBA::Object_ptr objPtr;
  ::CORBA::Status stat;
  int rc = 0;
  // Validate the input parameters.
  if ( argc != 2 )
  {
    cerr << "Usage: WSLoggerServer <server_alias>" << endl;
    exit( -1 );
  }
  if ( ( rc = perform_initialization( argc, argv ) ) != 0 )
    exit( rc );
...
}
```

Where:

server_alias

Specifies the server alias predefined in the system Implementation Repository.

perform_initialization(argc, argv)

Calls the initialization method of the server main code, to perform tasks to initialize the server environment. The method takes as argument the string (server alias) specified on the command used to start the server.

2. Edit the initialization method, to add code to initialize the server's implementation repository and retrieve the appropriate implementation definition, as shown in the following code extract:

```
int perform_initialization( int argc, char *argv[] )
{
  // Initialize the server's Implementation Repository.
  ::CORBA::ImplRepository_ptr implrep = new ::CORBA::ImplRepository();
  // Retrieve the appropriate ImplementationDef by using the server alias.
  try
  {
    imp = implrep->find_impldef_by_alias( argv[1] );
  }
  // catch exceptions ...
  cout << "Retrieved ImplementationDef" << endl;
...
}
```

3. Edit the initialization method, to add code to initialize the server's communications protocol, as shown in the following code extract:

```
int perform_initialization( int argc, char *argv[] )
{
...
  cout << "Retrieved ImplementationDef" << endl;
  // Set the server's communication protocol.
  imp->set_protocols("SOMD_TCPIP");
  cout << "Set communication protocol" << endl;
...
}
```

4. Edit the initialization method, to add code to initialize the ORB and object adapter, as shown in the following code extract:

```
int perform_initialization( int argc, char *argv[] )
{
...
  cout << "Set communication protocol" << endl;
...
  // Initialize the ORB.
  op = ::CORBA::ORB_init(argc, argv, "DSOM");
  // Initialize the BOA.
  try
  {
    bp = op->BOA_init(argc, argv, "DSOM_BOA");
  }
  // catch exceptions ...
  cout << "Initialized ORB" << endl;
...
}
```

5. Edit the initialization method, to add code to register the server, as shown in the following code extract:

```
int perform_initialization( int argc, char *argv[] )
{
...
  cout << "Initialized ORB" << endl;
  // Initialize this application as a server, allow it to accept
  // incoming request messages, and register it with the somorbd
  // daemon.
  try
  {
    bp->impl_is_ready( imp, 0 );
  }
  // catch exceptions ...
  cout << "Finished initialization of implementation" << endl;
  return( 0 );
}
```

   This enables the server to accept incoming request messages, and registers it with the somorbd daemon.

This task adds code to initialize the server environment for a CORBA server.

You need to add code to the server source file to enable the server to access naming contexts, as described in “Adding code to access naming contexts to a CORBA server main code (server.cpp)” on page 60.

Creating CORBA server main code (server.cpp), adding code to access naming contexts

Use this task to add a "get name context" method (for example, called get_name_context) to the source file for a CORBA server, to access naming contexts. This method is used to create a new naming context within which the CORBA server can create and make available servant objects. The method performs the following actions after the server environment has been initialized:

1. Getting a pointer to the naming service.
2. Getting a pointer to the root naming context.
3. (Optional) Creating a ::CosNaming::Name for the domain and getting a pointer to the domain naming context.
4. Getting a new servant naming context for servant objects.

This task is one step of the parent task to create the CORBA server main code, as described in “Creating a CORBA server main code (server.cpp)” on page 55 .

To add a get_name_context() method to the source file for a CORBA server main code, edit the server source file, *servant*Server.cpp, and add the following code:

1. Add a get_name_context() method, and add a statement to the main method to call the new method, as shown in the following code extract:

```
// This function accesses the Name Service and then gets or creates
// the desired naming contexts. It returns the naming context for
// the servant context.
::CosNaming::NamingContext_var get_naming_context()
{
  ::CosNaming::NamingContext_var rootNameContext = NULL;
  ::CosNaming::NamingContext_var domainNameContext = NULL;
  ::CosNaming::NamingContext_var  servantNameContext = NULL;
  ::CORBA::Object_ptr objPtr;
  return( servantNameContext );
}
...
void main( int argc, char *argv[] )
{
...
  if ( ( rc = perform_initialization( argc, argv ) ) != 0 )
    exit( rc );
  // Get the various naming contexts.
  ::CosNaming::NamingContext_var  servantNameContext = NULL;
  servantNameContext = get_naming_context();
  if ( ::CORBA::is_nil( servantNameContext ) )
    exit( -1 );
...
}
```

Where:

rootNameContext

is the pointer to the root naming context.

domainNameContext

is the pointer to the (optional) domain naming context.

***servant*NameContext**

is the pointer to the naming context for servant objects.

2. Edit the get_name_context() method, to add code to get a pointer to the naming service, as shown in the following code extract:

```
// This function accesses the Name Service and then gets or creates
// the desired naming contexts. It returns the naming context for
// the servant context.
::CosNaming::NamingContext_var get_naming_context()
{
  ::CosNaming::NamingContext_var rootNameContext = NULL;
  ::CosNaming::NamingContext_var domainNameContext = NULL;
  ::CosNaming::NamingContext_var  servant NameContext = NULL;
  ::CORBA::Object_ptr objPtr;
  // Get access to the Naming Service.
  try
  {
    objPtr = op->resolve_initial_references( "NameService" );
  }
  // catch exceptions ...
...
  return( servant NameContext );
}
```

This step returns a pointer object, `objPtr`, to the naming service.

3. Edit the get_name_context() method, to add code to get a pointer to the root naming context, as shown in the following code extract:

```
::CosNaming::NamingContext_var get_naming_context()
{
...
  else
    cout << "resolve_initial_references returned = " << objPtr << endl;
  // Get a root naming context.
  rootNameContext = ::CosNaming::NamingContext::_narrow(objPtr);
  if ( ::CORBA::is_nil( rootNameContext ) )
  {
    cerr << "ERROR: rootNameContext narrowed to nil" << endl;
    release_resources( bp, imp, op );
    return( NULL );
  }
  else
    cout << "rootNameContext = " << rootNameContext << endl;
  // Release the temporary pointer.
  ::CORBA::release(objPtr);
...
}
```

This code narrows the pointer object to the appropriate object type and assigns it to the new pointer object called `rootNameContext`, performs some checks, then releases the original pointer object, `objPtr`.

This step returns a pointer object, `rootNameContext`, to the root naming context.

4. Edit the get_name_context() method, to add code to create a ::CosNaming::Name for the domain and get a pointer to the domain naming context, as shown in the following code extract:

```
::CosNaming::NamingContext_var get_naming_context()
{
...
  // Get a root naming context.
...
  // Release the temporary pointer.
  ::CORBA::release(objPtr);
  // Create a ::CosNaming::Name for the domain.
  ::CosNaming::NameComponent nc;
  nc.kind = CORBA::string_dup("");
  nc.id = CORBA::string_dup("domain");
  ::CosNaming::Name_var name = new ::CosNaming::Name( 1, 1, &nc, 0 );
  // Get the domain naming context.
  try
  {
```

```
    objPtr = rootNameContext->resolve( name );
    cout << "objPtr from nameContext resolve = " << objPtr << endl;
  }
  // catch exceptions ...
  cout << "Resolved domain in root name context" << endl;
  domainNameContext = ::CosNaming::NamingContext::_narrow(objPtr);
  if ( ::CORBA::is_nil( domainNameContext ) )
  {
    cerr << "ERROR: domainNameContext narrowed to null" << endl;
    release_resources( bp, imp, op );
    return( NULL );
  }
  cout << "domainNameContext = " << domainNameContext << endl;
  // Release the temporary pointer.
  ::CORBA::release( objPtr );
...
}
```

The ::CosNaming::Name is initialized with the string "domain", which is the name of the domain naming context in which a new servant naming context is to be created. The domain naming context is located within the root naming context, so the root naming context is asked to resolve the name "domain". After resolving the name, the code narrows the result to get the domain naming context, assigns it to the new pointer object called `domainNameContext`, performs some checks, then releases the original pointer object, `objPtr`.

This step returns a pointer object, `domainNameContext`, to the domain naming context.

5. Edit the get_name_context() method, to add code to create a ::CosNaming::Name that specifies the name of the new context, "*servant*Context" (into which servant object are placed) and get a pointer to the naming context, as shown in the following code extract:

```
::CosNaming::NamingContext_var get_naming_context()
{
...
  // Create a ::CosNaming::Name for the domain.
...
  // Release the temporary pointer.
  ::CORBA::release( objPtr );
  // Get a new naming context for our objects.
  ::CosNaming::NameComponent nc2;
  nc2.kind = CORBA::string_dup("");
  nc2.id = CORBA::string_dup(" servantContext");
  ::CosNaming::Name_var name2 = new ::CosNaming::Name( 1, 1, &nc2, 0 );
  try
  {
    servantNameContext = domainNameContext->bind_new_context( name2 );
    cout << "bind_new_context,  servantNameContext = " <<  objectNameContext
      << endl;
  }
  // catch exceptions ...
  catch( ::CosNaming::NamingContext::AlreadyBound e )
  {
    cerr << "ERROR: bind_new_context threw AlreadyBound" << endl;
    cout << "Trying to resolve the context" << endl;
    try
    {
      ::CosNaming::Name *servantName = new ::CosNaming::Name;
      servantName->length( 1 );
      (*servantName)[0].id = ::CORBA::string_dup( " servantContext" );
      (*servantName)[0].kind = ::CORBA::string_dup( "" );
      ::CORBA::Object_ptr objPtr = domainNameContext->resolve( *  servantName
);
cout << "Before  objectNameContext = " <<  servantNameContext << endl;
      servantNameContext = ::CosNaming::NamingContext::_narrow( objPtr );
cout << "After  servantNameContext = " <<  servantNameContext << endl;
    }
    // catch exceptions ...
  }
  return( servantNameContext );
}
...
}
```

Where:

servant

is a string related to the name of the servant object class; for example, for the servant object class WSLogger you might create a naming context pointer called loggerNameContext.

This code narrows the pointer object to the appropriate object type and assigns it to the new pointer object called , *servant*NameContext performs some checks, then releases the original pointer object, objPtr.

This step returns a pointer object, *servant*NameContext, to the naming context for servant objects.

This task creates and returns a pointer object, *servant* NameContext, to the naming context for servant objects.

You need to add code to the server source file to name, create, and bind servant objects, as described in "Creating CORBA server main code (server.cpp), adding code to name, create, and bind servant objects" on page 60 .

Creating CORBA server main code (server.cpp), adding code to name, create, and bind servant objects

Use this task to add code to the source file for a CORBA server, to get a new ::CosNaming::Name for servant objects, create servant objects, and bind them into the appropriate naming context. This makes it possible for clients to find and use servant objects.

This task is one step of the parent task to create the CORBA server main code, as described in "Creating a CORBA server main code (server.cpp)" on page 55 .

To add code to name, create, and bind servant objects, edit the server source file, *servant*Server.cpp, and add the following code:

1. Add code to the main method to get a new ::CosNaming::Name for servant objects and to call a method to create and bind servant objects, as shown in the following code extract:

```
int create_and_bind( ::CosNaming::Name *nc, ::CosNaming::NamingContext_var
  servantNameContext )
{
  return( 0 );
}
void main( int argc, char *argv[] )
{
...
  // Get the various naming contexts.
  ::CosNaming::NamingContext_var  servantNameContext = NULL;
  servantNameContext = get_naming_context();
  if ( ::CORBA::is_nil( servantNameContext ) )
    exit( -1 );
  // Get a new ::CosNaming::Name for our  servant_Impl object.
  // This is done here rather than in create_and_bind() so that the
  // name can be reused later, when terminating the server.
  ::CosNaming::Name *nc = new ::CosNaming::Name;
  nc->length( 1 );
  (*nc)[0].id = ::CORBA::string_dup( " servantObject1" );
  (*nc)[0].kind = ::CORBA::string_dup( "" );
  // Create a new servant_Impl object and bind it to the
servantNameContext.
  if ( ( rc = create_and_bind( nc,  servantNameContext ) ) != 0 )
    exit( -1 );
...
}
```

Where:

***servant*NameContext**

is the pointer to the naming context for servant objects.

***servant*Object1**

is the name under which we choose to bind this servant object in the *servant*NameContext This name uniquely defines the *servant*_Impl object in the *servant*NameContext.

create_and_bind method

is the name of the method that the server calls to create servant objects

and bind them into the naming context.

The ::CosNaming::Name is obtained outside the create_and_bind() method so that the name can be reused later, when terminating the server.

2. Add a create_and_bind method, and add a statement to the main method to call the new method, as shown in the following code extract:

```
int create_and_bind( ::CosNaming::Name *nc, ::CosNaming::NamingContext_var
  servantNameContext )
{
  // Create a servant object.
  servantImpl = new servantImpl( "defaultlog" );
  // Bind the object to this name in the servant naming context.
        try
        {
          servantNameContext->bind( *nc, objectPtr );
          cout << "bind of servant
NameContext succeeded" << endl;
        }
        // catch exceptions ...
        return( 0 );
      }
```

This method takes as input the ::CosNaming::Name obtained before the method is called and the pointer to the naming context for servant objects. The code creates a servant object then binds the object to the ::CosNaming::Name in the servant naming context and performs a variety of binding checks.

Note: A string is passed to the servant object's initializer, to specify a default name for the log file. This name is not used by client programs, because clients change the default name to a user-specified value by invoking the method setFileName().

This task adds code that enables a CORBA server to name, create, and bind servant objects.

You need to add code to the server source file to enable the server to create a server shutdown object that can be used to help shutdown the server, as described in “Creating CORBA server main code (server.cpp), adding code to create a server shutdown object” on page 64.

Creating CORBA server main code (server.cpp), adding code to create a WSServerShutdown object

Use this task to add code to the source file for a CORBA server, to create a WSServerShutdown object, which enables the server to be shut down whenever needed.

This task is one step of the parent task to create the CORBA server main code, as described in “Creating a CORBA server main code (server.cpp)” on page 55 .

To add code to create a WSServerShutdown object, complete the following steps:

1. Edit the main method in the server source file, *servant*Server.cpp and add code to create a WSServerShutdown object, as shown in the following code extract:

```
void main( int argc, char *argv[] )
{
...
  // Create a new servant_Impl object and bind it to the objectNameContext.
  if ( ( rc = create_and_bind( nc,  servantNameContext ) ) != 0 )
    exit( -1 );
...
  // Create a WSServerShutdown object that can break the server out of the
  // method execute_request_loop() when we are ready to terminate
  // the server. The WSStopServer command will make the subsequent
  // invocation of execute_request_loop() return to the server.
  WSServerShutdown *shutdownObj = new WSServerShutdown( argv[1], bp );
  cout << "Created WSServerShutdown object" << endl;
  cout << endl;
  cout << "server listening...." << endl << endl;
  cout.flush();
  // Go into an infinite loop, servicing ORB requests as they are received.
```

```
...
}
```

When created, the WSServerShutdown object is initialized with the server alias and the object adapter pointer.

This task adds code to create a WSServerShutdown object. After the server has initialized itself, it creates the WSServerShutdown object, which waits for a message informing it that the server is to be shutdown. That message can be sent by the StopServer command line program provided with WebSphere Application Server enterprise services.

To continue developing the server main code, you need to add code to put the server into an infinite wait loop, during which the ORB can transmit requests to and from the servant object hosted by the server, as described in “Creating CORBA server main code (server.cpp), adding code to put the server into an infinite loop” on page 65.

Creating CORBA server main code (server.cpp), adding code to put the server into a loop to service requests

Use this task to add code to the source file for a CORBA server, to put the server into an infinite loop during which the ORB can transmit requests to and from the servant object hosted by the server.

This task is one step of the parent task to create the CORBA server main code, as described in “Creating a CORBA server main code (server.cpp)” on page 55 .

To add code to put the server into an infinite loop, edit the server source file, *servant*Server.cpp, and add the following code:

1. Add code to the main method to call a method to start the loop. There are several ways to do this, including calling the execute_request_loop() execute_next_request methods.This method should be called only after CORBA::BOA::impl_is_ready has been called successfully. The following example code extract shows the use of the execute_request_loop() method; in this example, a block wait is specified by CORBA::BOA::SOMD_WAIT:

```
void main( int argc, char *argv[] )
{
...
  // Initialize this application as a server, ...
  try
  {
    bp->impl_is_ready( imp, 0 );
  }
...
  cout << "Created ServerShutdown object" << endl;
  cout << endl;
  cout << "server listening...." << endl << endl;
  cout.flush();
  // Go into an infinite loop, servicing ORB requests as they are
  // received. execute_request_loop() will return when an external command,
  // StopServer, is executed.
  stat = bp-<;gt;execute_request_loop( ::CORBA::BOA::SOMD_WAIT );
  cout << "execute_request_loop has returned!" << endl;
  // Terminate the server.
...
}
```

This task adds code that puts a CORBA server into a loop during which it can service requests for the servant object that it hosts.

You need to add code to the server source file to enable the server to complete the server shutdown when requested, as described in “Creating CORBA server main code (server.cpp), adding code to shutdown the server and release resources used” on page 65.

Creating CORBA server main code (server.cpp), adding code to shutdown the server and release resources used

Use this task to create code for a CORBA server, to shut down the server and release the resources that it used.

This task is one step of the parent task to create the CORBA server main code, as

described in “Creating a CORBA server main code (server.cpp)” on page 55 .

To create code to shut down a CORBA server, complete the following steps

1. Edit the server source file, *servant*Server.cpp, and add a release_resources method, as shown in the following code extract:

```
// This function releases resources used throughout the program.
void release_resources( ::CORBA::BOA_ptr bp, ::CORBA::ImplementationDef_ptr
  imp, ::CORBA::ORB_ptr op )
{
  // Release the various resources we have allocated.
  bp->deactivate_impl( imp );
  ::CORBA::release( bp );
  ::CORBA::release( op );
  ::CORBA::release( imp );
}
```

   This method is called at the end of the server's main method after other shutdown processing has been completed. (You add a call to this method in the next step.) The method takes as input pointers to the object adapter, the server's implementation repository entry, and the ORB. It deactivates the implementation repository entry then releases the resources used by the server.

2. Edit the server source file, *servant*Server.cpp, and add code to the main method to respond to the server being shutdown (when the execute_request_loop is forced to return), as shown in the following code extract:

```
void main( int argc, char *argv[] )
{
...
  // Go into an infinite loop, servicing ORB requests as they are
  // received. execute_request_loop() will return when an external command,
  // WSStopServer, is executed.
  stat = bp->execute_request_loop( ::CORBA::BOA::SOMD_WAIT );
  cout << "execute_request_loop has returned!" << endl;
  // Terminate the server.
  // Unbind the servant object from the object naming context.
  cout << "Unbinding the  servant  object" << endl;
  try
  {
    object NameContext->unbind( *nc );
  }
  catch( ::CORBA::SystemException &ex; )
  {
    cerr << "ERROR: SystemException minor = " << ex.minor() <<
      " and id = " << ex.id();
    cerr << " was received when calling unbind()" << endl;
  }
  // Remove the logger naming context.
 try
  {
    object NameContext->destroy();
  }
  catch( ::CosNaming::NamingContext::NotEmpty e )
  {
    cerr << "ERROR: destroy threw NotEmpty" << endl;
  }
  release_resources( bp, imp, op );
  cout << "Exiting  servant Server..." << endl;
  cout.flush();
}
```

   This code completes the following actions:

   1. Unbinds the *servant* object from the *object* naming context.
   2. Removes the *object* naming context.
   3. Calls a release_resources method to releases resources used by the server.

This task adds code that shutdowns a CORBA server and releases the resources that it used, when the server's execute_request_loop() is forced to return. The loop returns when a shutdown request has been made by a separate server shutdown program.

Building a C++ CORBA server

This topic provides an overview of the task to build the code for a C++ CORBA server. The actual steps that you complete depend on the development environment that you use.

For example, if you are using the Microsoft C++ 6.0 compiler on Windows NT to build a C++ CORBA server, you can use the following commands:

1. Compile the server.cpp, *servant*_I.cpp, and *servant*_S.cpp files.

   At a command line, type the following command for each file:

   ```
   -DSOMCBNOLOCALINCLUDES -D_MS_INC_DIR= msvc60_install\include
   -D_USE_NAMESPACE -DNO_STRICT -I msvc60_install\include
   -Imsvc60_install\include
   -Iwasee_install\include -I. filename
   ```

   Where

   msvc60_install

   is directory in which the Microsoft C++ 6.0 compiler is installed; for example, `d:\msvc60\vc98`.

   wasee_install

   is the directory into which WebSphere Application Server enterprise services is installed.

   filename

   is the name of the file to be compiled (server.cpp, *servant*_I.cpp, or *servant*_S.cpp).

2. ```
   link /nologo /DEBUG /OUT:WSLoggerServer.exe
   /DEFAULTLIB:\WebSphere\AppServer\Enterprise\lib\somosa1m.lib
   \WebSphere\AppServer\Enterprise\lib\somororm.lib
   \WebSphere\AppServer\Enterprise\lib\somsrvsm.lib
   servantServer.obj servant_I.obj servant.obj
   ```

This task is one step of the parent task, “Developing a CORBA client” on page 40 .

For more examples of building C++ CORBA server code (on several platforms) for WebSphere Application Server, see the samples article "Tutorial: Creating a user-defined C++ server and client" at `WAS_HOME/Enterprise/samples/sampeex/sampcppsdk/wsBuildLogger.htm`.
   ```

Specifying runtime properties for C++ CORBA clients and servers

This topic provides an overview of the task to specify the runtime properties for C++ clients and server. You do this by defining the properties in a properties file, and specifying the full path, including the file name, of the properties file on the SOMCBPROPS environment variable. If you want clients and servers to use different properties, you must ensure that the SOMCBPROPS environment variable is set within the local environment of the client or server.

To specify the runtime properties of a C++ client or server, complete the following steps:

1. Create a properties file; for example, *client*.props:
2. Edit the properties file to specify appropriate runtime properties.
   The properties and values that you choose depend on your use of the clients and servers, and are selected from the properties listed in the reference topic “Runtime properties for CORBA clients and servers” on page .

   Example properties files (called WSClient.props, WSServer.props, and WSEJBClient.props), which show use appropriate subsets of the runtime properties for CORBA clients and servers, are provided with the WSlogger sample. If you have installed the enterprise services' samples, the properties files are located in `WAS_HOME/Enterprise/samples/sampcppsdk`

   The values that you specify for the runtime properties must match the equivalent settings used to configure WebSphere Application Server, where applicable.
3. Specify the name of the properties file on the SOMCBPROPS environment variable.

Creating your own C++ valuetypes

To aid application development, WebSphere Application Server provides a valuetype library that contains C++ valuetype implementations for some commonly used Java classes in the java.lang, java.io, java.util, and javax.ejb packages. For example, Integer, Float, Vector, Exception, OutputStream, and so on. However, you may want to create your own C++ valuetypes.

You can create your own C++ valuetypes by completing the following steps, which use java.util.Hashtable as an example:

1. Generate the IDL file of your Java class using the following command:

   rmic -idl java.util.Hashtable.java

2. Generate the .hh and _C.cpp files, using the following command:

   idlc -mcpponly -mnohhguards -mdllname=*vtlib_name* -shh:uc -I*include-path* java.util.Hashtable.idl

   This outputs the files Hashtable.hh and Hashtable_C.cpp. These two files are generated files and should not be edited. The Hashtable.hh file contains the super class definition and the definition of a default implementation class in the OBV_java::util namespace (if the original java is an abstract class or an interface, there is no default implementation class). The Hashtable_C.cpp file contains the default implementation of the java::util::Hashtable and OBV_java::util::Hashtable that are defined in Hashtable.hh.

3. Generate the .ih and _I.cpp files, using the following command

   idlc -mcpponly -mnohhguards -mdllname=*vtlib_name* -eih:ic -I*include-path* java.util.Hashtable.idl

   This outputs the files Hashtable.ih and Hashtable_I.cpp. These files contain initial templates for the definitions of the corresponding concrete subclasses that implement the abstract super class java::util::Hashtable. The name of a concrete implemention subclass is the valuetype name prefixed with the Java package name and suffixed with _Impl. In the case of Hashtable, it is java_util_Hashtable_Impl, which inherits from the default implementation class OBV_java::util::Hashtable. These files are only generated once. The files are next edited to add the implementation details.

4. To add the implementation details to the .ih and _I.cpp files, complete the following steps:

   a. Add the implementation of all the public methods defined in the super class; in this case, java::util::Hashtable. New instance variables and methods can be added to the implementation class, java_util_Hashtable_Impl.
   Instance variables of a java class are mapped into the C++ conterparts in the default implementation class in the OBV_* namespace. ORB marshaling/demarshaling uses these instance variables. Therefore, in the *_Impl class, you need to use the getters and setters of the instance variables defined in the default implementation class in the OBV_*

namespace.

b. Add the implementation of the creation methods defined in the corresponding factory class. These methods correspond to the constructors in the Java class.

c. Create a factory object for the valuetype and register it with the ORB by using the following method:
`orb->register_value_factory(...);`
This enables the ORB to get the factory of the valuetype to create instances for the valuetype during marshaling and demarshaling.

For an example of the use of C++ valuetypes, see the sample vtlib_vb.ih and vtlib_I_vb.cpp files in the directory `WAS_HOME\samples\interop\ejb\java\ws4.0\cpp\visibroker4.0\client`.

Writing a WebSphere Enterprise JavaBean as a client of a 3rd-party CORBA ORB

An enterprise bean hosted by WebSphere Application Server can act as a client to a CORBA server on a third-party ORB. The enterprise bean itself is written like any other enterprise bean. It must implement the required methods in the usual home and remote interfaces so that its clients can contact it. The remote interface defines the business methods of the bean, and all of the work related to using the CORBA server occurs in the implementation of those methods. The use of a CORBA server does not change the usual programming tasks associated with enterprise beans, but the code in the remote methods must include code for communicating with the server. This includes code for the following:

- Getting a reference to the client-side ORB, which is needed for communicating with the servant objects
- Looking up the servant objects
- Calling the methods in the CORBA IDL interface betwen the enterprise bean and the server

In the sample application, the enterprise bean that acts as a client of the CORBA server is written as a stateless session bean. The writing such an enterprise bean can be divided logically into two pieces:

- Writing the usual parts of an enterprise bean:
    - The home interface: In the enterprise bean described here, this interface consists of a single create method.
    - The remote interface: In the bean described here, this interface consists of methods that initiate the interoperability tests. These methods are called by the client of the enterprise bean to run a suite of interoperability tests. Of primary interest are the following methods:
        - setOrbProperties: This method obtains a reference to the client-side ORB and sets necessary properties.
        - testNameService: This method attempts to locate the servant objects either by contacting the name service directoy and looking up the server, by re-creating the name-service IOR from the file that the server writes and looking up the servant objects, or by re-creating the servant-object IORs from the files that the server writes and contacting the objects directly. (The specific test that is run is determined by a mode variable.)
        - testPrimitive: This method connects to the servant object, starts a loop, and runs through the methods in the Primitive interface.
        - testComplex: This method connects to the servant object, starts a loop, and runs through the methods in the Complex interface. (This method is structurally identical to the testPrimitive method and is not discussed in detail.)

      These methods make use of other methods designed to support the interaction with the CORBA server.
    - The bean class: This is the class in which the methods required by the home and remote interfaces are implemented. It can also contain methods used internally by the home and remote methods.
- Writing the code to support the interaction with the CORBA server. This code is called

from the methods in the remote interface and is the primary object of this discussion. In general, the types of things this code must do include:

- Getting a reference to the client-side ORB. See “Contacting the client-side ORB” on page 72for more information.
- Establishing connections to servant objects. See “Locating servant objects” on page 73for more information.
- Issuing remote invocations to the server. See “Invoking servant objects” on page 74for more information.

Writing a WebSphere Enterprise JavaBean as a CORBA client, contacting the client-side ORB

To use a CORBA server running on a third-party ORB, an enterprise bean in the WebSphere environment must explicitly make contact with the client-side ORB so that it can issue remote method invocations to the server.

The enterprise bean described in this example makes use of a class called ClientOrb, which provides most of the logic needed for using the client-side ORB. This class includes the appropriate auxiliary files and methods The methods include the getOrb method, which returns a reference to the client-side ORB, and the related getOrbProperties method, which is used to read the values used in initializing the ORB from a properties file specified at startup. The ClientOrb class does the following:

1. Includes the resources in the source code. For the ClientOrb class, these include standard Java resources, CORBA naming resources, and some application-specific utilities.

```
package com.ibm.orb.interop.samples;
 // Java
 import java.io.FileReader;
 import java.io.BufferedReader;
 import java.io.FileInputStream;
 import java.io.IOException;
 import java.io.FileNotFoundException;
 import java.util.Properties;
 // CORBA
 import org.omg.CORBA.ORB;
 import org.omg.CosNaming.NameComponent;
 import org.omg.CosNaming.NamingContext;
 import org.omg.CosNaming.NamingContextHelper;
 // Application-specific
 import com.ibm.orb.interop.samples.Defs;
 import com.ibm.orb.interop.samples.util.Logger;
 import com.ibm.orb.interop.samples.util.Report;
 import com.ibm.orb.interop.samples.util.UserReport;
```

2. Provides the getOrb method, which obtains a reference to the client-side ORB if one is not already available. If the method obtains a new reference, it also intializes the ORB by passing a set of properties. The client-side ORB is the IBM ORB, not the third-party ORB.

```
public ORB getORB()
 {
    if (orb==null)
    {
       ...
       com.ibm.ejs.oa.EJSORB.getORBInstance();
       if (orb==null)
          orb = ORB.init((String[]) null,
 getOrbProperties());
    }
    return orb;
 }
```

Writing a WebSphere Enterprise JavaBean as a CORBA client, locating servant objects

Before a CORBA client can call methods on a servant object, it must establish a connection to that object. The techniques available for establishing such connections depend in part on the ORB for which the server is written and on the design of the application.

The server described in this example puts binding information into the name server and also writes string versions of that information out into files. The client therefore has several choices in the way it looks up the servant objects. It can do any of the following:

- Go directly to the name service and look up the desired servant object.
- Read the name-service binding information from the file created by the server, reconstruct the IOR from this information, use the IOR to reach the name service, and look up the desired servant object.
- Read the servant-object binding information from the file created by the server, reconstruct the IOR, and contact the object directly.

The code that accomplishes these tasks is spread across a variety of methods. The code shown below reflects the logical sequence of events, not explicit code extracts. The getOrb method called in these code fragments is described in “Contacting the client-side ORB” on page 72 ; this method returns a reference to the client-side ORB.

Note: When an enterprise bean declares CORBA naming-context objects, it must explicitly type them as org.omg.CORBA.Object types to distinguish them from java.lang.Object types.

The sample client is designed to test several techniques for establishing connectivity to the servant objects, so it provides the following code to support these techniques:

1. To get a reference to the name service, do one of the following:
   a. Get an initial reference to the name server by calling the resolve_initial_references method on the ORB reference, and then narrow the reference.

```
org.omg.CORBA.Object
nameServiceObject = null;
 NamingContext        nc = null;
 nameServiceObject =
getORB().resolve_initial_references("NameService");
 nc =
NamingContextHelper.narrow(nameServiceObject);
```

   b. Read the context from the file written by the server, reconstruct the IOR, and narrow the reference.

```
org.omg.CORBA.Object
nameServiceObject = null;
 NamingContext        nc = null;
 String
nameServiceIOR;
 nameServiceIOR =
readIOR("nameservice.ior",
"NameService");
 nameServiceObject =
getORB().string_to_object(nameServiceIOR);
 nc =
NamingContextHelper.narrow(nameServiceObject);
```

2. To get a reference to a servant object, do one of the following:
   a. By using the naming context obtained in the previous step, look up the servant object by name. The resulting object, of the type org.omg.CORBA.Object, must be narrowed before it can be used. For example, if the client

looks up the Primitive servant object, where the value of the name argument used below is "primitive", the PrimitiveHelper.narrow method must be used.

```
org.omg.CORBA.Object servantObject
= null;
 NamingContext        primitive =
null;
 // Name service...
 nc =
NamingContextHelper.narrow(nameServiceObject);
 NameComponent nameSeq[] = new
NameComponent[1];
 nameSeq[0] = new
NameComponent(name, "");
 servantObject =
nc.resolve(nameSeq);
 // Later...
 primitive =
PrimitiveHelper.narrow(servantObject);
```

b Read the context from the file written by the server, reconstruct the IOR, and narrow the reference. This does not require a naming-service context.

```
org.omg.CORBA.Object
primitiveObject = null;
 NamingContext        primitive =
null;
 String
primitiveIOR;
 primitiveIOR =
readIOR("primitive.ior",
"primitive");
 primitiveObject =
getORB().string_to_object(primitiveIOR);
 primitive =
PrimitiveHelper.narrow(primitiveObject);
```

Writing a WebSphere Enterprise JavaBean as a CORBA client, invoking a servant object

An enterprise bean acting as a client to a CORBA server uses the servant objects in the same way as any other CORBA client: it obtains a reference to the object and invokes the methods. This work is typically done within the implementation of the methods in the enterprise bean's remote interface.

In the example on which this task is based, the methods in the remote interface are implemented by calling methods in several supporting classes. For example, the remote interface provides calls to test the CORBA Primitive and Complex interfaces. Methods in the Primitive interface are called by the client-side TestPrimitive class, which is used in the implementation of the remote interface. (The structurally similar TestComplex class performs the same tasks for the Complex interface.) The TestPrimitive class provides the following:

- Constructors
- Methods for testing each data type in the Primitive interface, singly and iteratively

From an organizational perspective, the class is structured as follows:

```
package com.ibm.orb.interop.samples;
 // Java
 import java.util.Properties;
 // CORBA
 import org.omg.CORBA.ORB;
 import org.omg.CosNaming.NameComponent;
 import org.omg.CosNaming.NamingContext;
 import org.omg.CosNaming.NamingContextHelper;
```

```
// Application-specific
import com.ibm.orb.interop.samples.Defs;
import com.ibm.orb.interop.samples.idl.*;
import com.ibm.orb.interop.samples.util.Report;
public class TestPrimitive
{
   private boolean    showData = false;
   private ClientOrb  clientOrb = null;
   private Report     testReport = null;
   private Primitive  primitive = null;   // Target server reference
   // Constructors
   public TestPrimitive(ClientOrb clientOrb, Primitive primitive)
   { ... }
   public TestPrimitive(ClientOrb clientOrb,
                        org.omg.CORBA.Object primitive)
   { ... }
   /// Methods to exercise the Primitive interface
   public void testOctet(String testName) { ... }
   public void testBoolean(String testName) { ... }
   public void testShort(String testName) { ... }
   ...
}
```

The TestPrimitve class provides the following kinds of code:

1. Two constructors, which are used to obtain a reference to the client-side ORB and to narrow a CORBA Object reference to a Primitive servant object.

```
public TestPrimitive(ClientOrb clientOrb, Primitive
primitive)
 {
    super();
    this.clientOrb = clientOrb;
    this.primitive = primitive;
    ...
 }
 public TestPrimitive(ClientOrb clientOrb,
                      org.omg.CORBA.Object primitive)
 {
    this(clientOrb, PrimitiveHelper.narrow(primitive));
 }
```

2. Methods for testing each of the methods in the Primitive interface. A typical TestPrimitive method calls the three Primitive methods for a particular data type. In the Primitive interface, each data type has three methods, each designed to exercise a different way of returning values: as the value of the method, as an `out` argument, and as an `inout` argument. The typical TestPrimitive methods follows this pattern: compute the expected return value, call the remote method that returns the result as the value of the method, compare the actual and expected results, log the results, call the remote method that returns the result as an `out`, compare the result to the expected value, log the results, call the remote method that returns the result as an `inout`, compare and log the results. For illustration, the code fragment shows the relevant parts of the TestPrimitive.testShort method, which calls the methods in the Primitive interface that work with short integers.

```
 short initial = 32000;
 short expected = PrimitiveOps.processShort(initial);
 org.omg.CORBA.ShortHolder holder = new
org.omg.CORBA.ShortHolder();
 try
 {
    // Test the first of three methods.
    short result = primitive.testShortIn(initial);
    ...
    testReport.log("Initial: " + initial
        + ",  Expected: " + expected
        + ",  Result: " + result );
 }
 catch (Exception e)
 { ... }
 try
 {
    // Test the second of three methods.
    primitive.testShortOut(initial, holder);
    short result = holder.value;
```

```
   ...
   testReport.log("Initial: " + initial
       + ",  Expected: " + expected
       + ",  Result: " + result );
}
catch (Exception e)
{ ... }
try
{
   // Test the third of three methods.
   holder.value = initial;
   primitive.testShortInOut(holder);
   short result = holder.value;
   ...
   testReport.log("Initial: " + initial
       + ",  Expected: " + expected
       + ",  Result: " + result );
}
catch (Exception e)
{ ... }
```

Writing a WebSphere Enterprise JavaBean as a CORBA client, building the Enterprise JavaBean

After the client is coded, it must be compiled. For a basic enterprise bean, this work is done as part of the deployment, in which the code needed by the container for managing the bean is generated. For a bean that also acts as a CORBA client, the CORBA-related code must be separately compiled. This code is outside the domain of the deployment process. The usual approach is as follows:

- Use the Websphere IDL compiler to compile the CORBA IDL interfaces; this is done to generate the client-side code.
- Compile the Java code that supports the CORBA calls to the server.
- Compile the enterprise-bean code.
- Package and deploy the bean for the WebSphere environment.
- Start the bean.

The packaging, deployment, and starting of the bean are done in the usual manner for enterprise beans running in the WebSphere environment. They are not discussed here.

The enterprise bean described in this example makes use of two CORBA IDL interfaces, Primitive and Complex. These are compiled for Java with IDL compiler provided by WebSphere Application Server. The source code for the client consists of the IDL-generated files, the code called by the bean's business methods to invoke the CORBA server, and the actual enterprise-bean code, consisting of the home and remote interfaces and the bean class. These are compiled with Java's **javac** compiler. The example application is built as follows:

1. Set the environment variables for the build.

```
set
PATH=%JDKROOT%\bin;%JDKROOT%\jre\bin;%JDKROOT%\jre\bin\classic;%PATH%
 set WAS_CP=.
 set WAS_CP=%WAS_CP%;%WAS_HOME%\lib\j2ee.jar
 set WAS_CP=%WAS_CP%;%WAS_HOME%\lib\ejbcontainer.jar
 set CLASSPATH=%WAS_CP%
```

2. Compile the IDL files for the client. To generate Java code from the IDL files, use the idlj compiler provided with WebSphere Application Server. The -v option requests verbose output; the -fall option directs the compiler to generate all bindings.

```
% idlj -v -fall Primitive.idl
 % idlj -v -fall Complex.idl
```

3. Compile the CORBA-related source code and the IDL-generated interface code into class files.

```
% javac com\ibm\orb\interop\samples\idl\*.java
 % javac com\ibm\orb\interop\samples\util\*.java
 % javac com\ibm\orb\interop\samples\Defs.java
 % javac com\ibm\orb\interop\samples\ClientOrb.java
 % javac com\ibm\orb\interop\samples\PrimitiveOps.java
 % javac com\ibm\orb\interop\samples\ComplexOps.java
 % javac com\ibm\orb\interop\samples\TestNameService.java
 % javac com\ibm\orb\interop\samples\TestPrimitive.java
 % javac com\ibm\orb\interop\samples\TestComplex.java
```

4. Compile the enterprise-bean source code into class files.

```
% javac com\ibm\orb\interop\samples\ejb\Client.java
 % javac com\ibm\orb\interop\samples\ejb\ClientHome.java
 % javac com\ibm\orb\interop\samples\ejb\ClientBean.java
```

5. Package the bean for deployment by using the WebSphere application assembly tool (AAT) to create an enterprise application (EAR) file.
6. Deploy the bean into the container by using the WebSphere administrative console.
7. Start the bean by using the WebSphere administrative console.

Tasks for problem determination

You can use the following tasks to help identify the cause of a runtime problem with WebSphere Application Server enterprise services:

- To set the directories and maximum sizes for activity and trace logs, and to turn on various component traces, you can specify appropriate runtime properties as described in “Specifying runtime properties for C++ CORBA clients and servers” on page 68
- To format one activity log or trace log, you can use the showlog utility, as described in “Formatting an activity or trace log” on page 78
- To examine the information output by formatting an activity log, see“Reading a formatted activity log” on page 79
- To format several trace logs at the same time and merge the output into one file, you can use the combtrace utility, as described in “Formatting and merging multiple trace files” on page 81.
- To simplify the task of finding appropriate information in a formatted trace log, you can filter the information as described in“Filtering the information in a formatted trace file” on page 81
- “Identifying and resolving CORBA interoperability issues” on page 83

Formatting an activity or trace log

The WebSphere Application Server enterprise services' activity logs and trace logs must be formatted before you can read their contents. You can use the showlog utility to format an activity log or trace log, and pipe the formatted output to a file that you can view using the Log Analyzer or any text editor:

Note: Run the showlog utility on the host where the log was created (where events in the log occurred) to get the optimum substitution values in the output file.

If you want to format several trace logs at the same time and merge the output into one file, you can use the combtrace utility, as described in “Formatting and merging multiple trace files” on page 81.

To use the showlog utility to format an activity or trace log file, complete the following steps

1. Open a command-line window on the host where the log file is stored.
2. On the command line, type the following command

```
showlog -cpp [-debug|-d|-d1|-d2|-d3] [-?] filename [> outfilename ]
```

The command line options may be specified in any order.

Where:

-cpp

Indicates that you want to format an enterprise services' log.

-debug, -d, -d1

These alternative options are all equivalent and display log messages in level 1 debug mode. Under this mode, showlog formats each activity log entry with full debug information, but ORB communication traces are not formatted.

-d2

This option is used when formatting ORB communication trace logs. It

displays log messages in debug level 2 mode which is equivalent to debug level 1 mode plus the formatting of ORB communication trace messages in brief mode for enhanced readability. You get additional GIOP header data following the raw data dump.

-d3

This option is used when formatting ORB Communication trace logs. It displays log messages in debug level 3 mode which is equivalent to debug level 2 mode plus the verbose expansion of communication traces. This includes formatting of service context and tagged profile information. You get additional GIOP header data and other information on the service context, object key, principal, and so on.

-?

displays the help information for the showlog command.

filename

is the input log file, for example, `activitycpp.log`. The showlogcpp utility reads the file and formats it for reading. The filename does not have to be the first option specified.

> outfilename

This option is used to pipe the formatted output to a new file.

Note: If the log output is piped to a file and an editor is used to display the information, turn on word wrap in the editor to view the full contents of a line.

Note: If you do not specify -debug, -d, or -d1, then only PrimaryMessage, ExtendedMessage, and RawDataLen information is included for each entry in the log.

The showlogcpp utility formats the information in the input log file, and optionally pipes the output to a new file that can be displayed in a text editor (or some other utility).

For example, the following steps format the activity log file c:\WebSphere\AppServer\services\activitycpp.log and pipe the formatted output to the file showlogcpp.out:

1. Open a command line window on the host where the activity log is stored
2. At the command line, type the following command:

```
showlogcpp c:\WebSphere\AppServer\services\activitycpp.log > showlogcpp.out
```

After formatting a trace file, you can use the protrace utility to filter the output in the *outfilename* file, as described in “Filtering the information in a formatted trace file” on page 81 .

Reading a formatted activity log

The WebSphere Application Server enterprise services' activity logs and trace logs must be formatted before you can read their contents. You can use the showlog utility to format an activity log or trace log, and pipe the formatted output to a file as described in “Formatting an activity or trace log” on page 78 .

To read a formatted activity log, you can use you can use a text editor

It is easier to locate the cause of a problem in smaller activity logs. Therefore, consider reducing the size of the activity log before attempting to read it. For more information about, see “Hints and tips: activity log” on page 36 .

When reading a formatted activity log, you need to identify the group of entries that are related to the problem or error that you want to resolve. A group of entries forms a bracket, as follows:

The start of the bracket

Initial failure, which is a single entry in the log

Results of the initial failure

A number of entries in the log

The end of the bracket

Last result of the failure, which is a single entry in the log

In general, when you are reading the activity log, you start with its last entry and then work backwards, reading the previous entry, then the one before that, and so on.

To find the bracket of entries for a problem that you are diagnosing, complete the following steps:

1. Identify the end of the bracket.
   The first step in reading the activity log is to identify the last entry that reported the problem that you want to diagnose (that is, the end of bracket entry). This is essential for identifying the cause of the problem. You want to start with the latest entry in the activity log and search backwards for the entry that reports the problem.

   Note: Sometimes, the last entries of the log do not relate to the problem that you are interested in. For example, there may be entries made by the ORB or by requests not associated with the problem

   If you do not know the UOW for the problem that you want diagnose, you can examine the entries by alternative groupings, such as TimeStamp. You can look for the end of bracket by searching in the log starting with its last entry with the TimeStamp.

   When the entry related to the failure has been identified, you have found the end of the bracket. Remember the unit of work (UOW) and record IDs (Rec_nnnn) for the end of bracket entry. In the next step, you look at entries within that unit of work.

   Sometimes you may not have any UOW identification for the end of bracket entry. In such situations, you must look at entries that do not have a UOW identification.

2. Find relevant entries.
   Examine each entry with the unit of work identified for the end of bracket entry, to see if the entry is related to the problem. Examine the entries with the same unit of work identifier in reverse timestamp sequence, starting with the entry before the end of bracket, then the entry before that, and so on, to try to identify all the relevant entries until you find the first entry for the initial cause of the problem.

   Some activity log entries are reraised exceptions received from lower level calls. The fact that these reraised exceptions occur suggests that these entries are not the source of the problem. Often, you are not interested in reraised exception entries in the log. Therefore, you may want to read the first few entries before the end of bracket and then quickly skim over the ones that have reraised exception

Also, sometimes the runtime remaps the exception it receives from a lower call to another exception which is defined on its interface.

3. Find the initial failure
When you have found the first entry for the initial cause of the problem, you can take action to resolve the problem. Depending on the situation, you may also want to read a couple of entries before the initial failure's entry, just in case there is some additional data to help you diagnose the problem.

Formatting and merging multiple trace files

If you turn on component trace, WebSphere Application Server enterprise services stores trace data in one or more trace logs, which must be formatted before you can read their contents.

If you want to format several trace logs at the same time and merge the output into one file, you can use the combtrace utility, as described in this topic.

If you want to format a single trace log, you can use the showlog utility as described in"Formatting an activity or trace log" on page 78 .

To use the combtrace utility to format multiple trace log files, complete the following steps

1. Open a command-line window on the host where the trace logs are stored.
2. On the command line, type the following command

```
Combtrace outfilename
```

The combtrace command calls showlog -debug to format all the yydddhhmmss.xxx trace files in the current directory, and sequentially combines the results in the specified outputfile. It ignores all file names containing a character other than a number or a period (.).

You can use the protrace utility to filter the output in the *outfilename* file, as described in "Filtering the information in a formatted trace file" on page 81.

Filtering the information in a formatted trace file

If you turn on component trace, WebSphere Application Server enterprise services stores trace data in one or more trace logs, which must be formatted before you can read their contents.

After you have formatted a trace log, as described in related tasks, you can use the protrace utility to filter the formatted trace information. You can use the protrace utility to filter the formatted trace information, to perform one or more of the following actions:

- Filtering: each entry in the trace is reduced to a single line (with the exception of raw data).
- Filtering: entry and exit trace points are used to control indenting of the output. A separate indent is maintained for each thread in the trace.
- Formatting: parenthesis are added to the output to allow editors such as vi to match entry and exit trace points.
- Filtering: the utility is configurable. For example, fields (for example, PID) can be switched off.
- Sorting: the output is sorted into date and time sequence.

To use the protrace utility to filter a formatted trace log file, complete the following steps

1. Open a command-line window on the host where the formatted trace log is stored.
2. On the command line, type the following command

```
protrace inputfile [outputfile] [-t thread] [-i maxindent]
[-r] [-m]
 [-s(b,e,d,n,p,t,r,i)] [-inc stringlist] [-exc stringlist]
[-v]
```

Where:

inputfile

is the file output from the showlog or combtrace command.

outputfile

(if specified) the output is sent to this file rather than sent to standard output.

-t *thread*

displays entries for the specified thread.

-i *maxindent*

specifies the maximum indent level. Set it to 0 (zero) to disable indenting.

-r

includes any raw data in the output.

-m

displays the primary message.

-s(b,e,d,n,p,t,r,i)

suppresses output specified by one or more of the following characters appended to the -s:

b

disables braces. ({})

e

disables extended message.

d

disables date/time.

n

disables entry numbering.

p

disables process Id.

t

disables thread Id.

r

disables the thread indent report generated after each run.

i

disables the raw data indicator.

For example, the option `-sedp` means that extended message, date/time, and processId information is suppressed.

-inc *stringlist*

specifies a set of inclusion strings. The primary or extended message must contain at least one of these strings to be output. Separate each entry in the list with white space.

-exc *stringlist*

specifies a set of exclusion strings. The primary or extended message *must not* contain any of these strings to be output. Separate each entry in the list with white space.

-v

Notes:

1. By default, open and close braces ({...}) are added to entry and exit trace points. This allows bracket matching in editors such as vi.
2. The list separator in the inclusion and exclusion options is a space character. Inclusion or exclusion strings that contain a space character must be delimited with quote (") symbols. For example: `-exc "was entered"`.
3. The R field in the output is the raw data indicator field. This field is only displayed when the raw data display is disabled and is used to highlight entries that have raw data associated with them. Such entries have a dash (-) in this field. To remove the indicator, and append the raw data to the output, specify the -r flag on the command line.
4. The protrace utility warns you if any invalid entries are found in the input file. A trace entry is invalid if the function name is missing or the date/time field is missing or incorrectly formatted.

The output in the *outfilename* file can be displayed in a text editor (or some other utility).

Identifying and resolving CORBA interoperability issues

By embracing the CORBA open architecture, WebSphere can work with new and old applications from different vendors. Since complete compatibility between CORBA ORBs is not yet available, this section describes issues that you may encounter when working with different ORBs, and provides some strategies to resolve these issues.

Compliance statements by ORB vendors do help to identify a set of features that can be expected to function appropriately within a distributed environment involving multiple instances of the same ORB. However, compliance does not assure interoperability between different ORBs.

When trying to resolve an interoperability problem, first review the fundamentals: the communication link (GIOP/IIOP). Ensure that both the client and server ORBs support the expected features. Next, begin at your point of failure in the steps below. Review the information provided in the linked topics, and if needed consult the CORBA specification. The CORBA specification also includes comments on differences between different levels of the GIOP specification.

Failure to interoperate between a CORBA client and a servant object is, by its very nature, manifest in the external communication protocols between the two. If interoperability fails, check the following in sequence:

1. Check the communication links: “CORBA communication protocols (GIOP/IIOP)” on page 8 .
2. Check the use of the naming service: “C++ CORBA client, locating the root naming context (bootstrapping)” on page 16 and “C++ CORBA client, locating a servant object” on page 18 .
3. Check for unsupported CORBA valuetypes and data types: “CORBA value type considerations ” on page 8 and “Resolving unsupported CORBA data types” on page 10 .

CORBA support example articles

This part contains example topics about the CORBA support provided by WebSphere Application Server 4.0 enterprise services. These topics provide an overview of, and links to, the samples provided with WebSphere Application Server enterprise services.

- “CORBA support concept articles” on page 1
- “CORBA support task articles” on page 39
- “CORBA support reference articles” on page

Sample: C++ CORBA client of a C++ servant object

Samples are provided to demonstrate typical use of the CORBA client and server programming models to develop a C++ CORBA client of a C++ CORBA server within a WebSphere Application Server environment. The client and server use standard CORBA and IBM CORBA extension methods.

The sample code demonstrates tasks to create a client that accesses a server object as follows:

- Define the server on WebSphere Application Server.
- Create a C++ server, named WSLoggerServer, that hosts a C++ implementation object (a servant in CORBA 2.3 terminology).
- Define and implement a C++ servant, named WSLogger_Impl.
- Create a C++ client, named WSLoggerClient, to use the server and its servant.
- Build the client and server components.
- Test the client and server.

Sample files are provided for Windows, AIX, and Solaris platforms.

For more information about the samples, see *Tutorial: Creating a user-defined C++ server and client* (wsBuildLogger.htm) in the WSLoggerServer sample. If you have installed the samples option for WebSphere Application Server enterprise services, the tutorial and associated files are installed in the following directory: see the samples article "Tutorial: Creating a user-defined C++ server and client" at `WAS_HOME/Enterprise/samples/sampcppsdk`.

Sample: C++ CORBA client of an Enterprise JavaBean

Samples are provided to demonstrate typical use of the CORBA client programming model to develop a C++ CORBA client of the Hello Enterprise JavaBean within a WebSphere Application Server environment. The client uses standard CORBA methods.

The sample code demonstrates tasks to create a client that accesses a server object as follows:

- Performing general setup.
- Generating C++ bindings from the EJB source files and Java source files.
- Creating a C++ client, named WSEJBClient, which uses the Hello Enterprise JavaBean supplied with WebSphere Application Server.
- Building the client.
- Testing the client.

Sample files are provided for Windows, AIX, and Solaris platforms.

For more information about the samples, see *Tutorial: Creating a user-defined C++ client that uses an Enterprise JavaBean* (wsBuildEJBClient.htm) in the WSLogger sample. If you have installed the samples option for WebSphere Application Server enterprise services, the tutorial and associated files are installed in the following directory: `WAS_HOME/Enterprise/samples/sampcppsdk`.

CORBA interoperation samples

Samples are provided to demonstrate typical use of the CORBA and EJB programming models for interoperation between WebSphere and 3rd-party ORBs. Where necessary, the strategies described in the previous sections are applied. The sample code covers the following scenarios:

Client	Server
3rd-party ORB, C++ language bindings	WebSphere Enterprise JavaBean
WebSphere EJB server (as a CORBA client)	3rd-party Java CORBA object
3rd-party C++ CORBA object	
WebSphere EJB server (as a CORBA client) coexistent with a 3rd-party Java ORB	3rd-party C++ CORBA object

For the latest list of interoperation samples, see **Samples and Tutorials > Websphere - 3rd Party ORB interoperation** in the navigation pane of the WebSphere Application Server enterprise services infocenter.

The Enterprise JavaBeans used in the sample code, for both client and server, are session beans.

The program techniques vary from one scenario to another, and are described in detail in the documentation for the scenario.

Each set of sample code explores server object access, primitive data types, and complex data types and operations between a WebSphere ORB and a specific vendor ORB. The sample code *does not* attempt to verify compliance, nor does it attempt to verify consistent behavior of an ORB. For example, not all possible ways of exchanging data types are explored: simple data types can be passed to the server as CORBA in parameters, and returned in the return-value, where CORBA out and inout parameters are not used. Some features work or do not work for a specific ORB. The information accompanying each sample states whether or not a feature can be used successfully.

The sample code demonstrates access to a server object as follows:

- Name Service IOR: A utility writes the IOR of the server's root naming context to a file, which is read by a client. The client looks-up the server object in that naming context.
- Object IOR: A utility writes the server object's IOR to a file, which is read by client.

For CORBA server objects, the sample code demonstrates methods that exchange simple IDL data types as input parameters (CORBA in parameters) and return types. The sample code also demonstrates methods that exchange complex IDL data types and operations. This includes the following data types and operations:

- octet
- boolean
- short
- long
- long long
- float
- double
- char

- wchar
- Object
- string
- wstring
- any
- compound data types (struct, union, array, sequence)

For CORBA access to Enterprise JavaBeans hosted by a WebSphere EJB server, the sample code demonstrates use of Java data types. The sample code also demonstrates the exceptions that are generated by an Enterprise JavaBean and caught by the client. This includes the following following Java data types and exceptions:

- Java data types (primitive data types):
    - byte
    - boolean
    - short
    - lint
    - long
    - float
    - double
    - char
    - org.omg.CORBA.Object
- Exceptions generated by Enterprise JavaBeans:
    - java.lang.IndexOutOfBoundsException (RuntimeException)
    - java.lang.UnknownError java.rmi.remoteException
    - java.rmi.MarshalException
    - java.rmi.NoSuchObjectException
    - java.rmi.AccessException
    - java.rmi.RemoteException
    - javax.ejb.createException
    - javax.transaction.TransactionRequiredException
    - javax.transaction.TransactionRolledBackException
    - javax.transaction.InvalidTransactionException

C++ value type library, examples

The following examples are provided to illustrate use of the valuetype library methods in a distributed environment.

Example: A client program that uses a remote object to call methods of ::java::util::Vector

```
//First obtain the stringified ior of an EJB deployed on an AE server
using namespace com::ibm::ws;
CORBA::Object_var vector_obj;
//init the orb
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "DSOM");
//Get the stringified ior from a file, then use it to obtain a valid object reference
    ifstream in;
    int fileIndex = 0;
    char* iorfile = "VectorSession.ior"; // the stringified ior of a session bean that
                                         // uses instances of the java serializable
                                         // java.util.Vector
    in.open(iorfile);
    if (in.fail()) {
       std::cerr << "Cannot open file " << iorfile << std::endl;
       return 1;
    }
    char iorstr[2048];
    //read ior from file
    in >> iorstr;
    in.close();
    std::cout << "using ior:'" << iorstr << "'" << std::endl;
    //get stringified ior
    vector_obj = orb->string_to_object(iorstr);
    if (CORBA::is_nil(vector_obj))
    {
        std::cerr << "string_to_object failed"<< std::endl;
        return 1;
    }
// Now, call the createVector method of the stub class ejbPackage::VectorSession to access
// an EJB method that returns an instance of the java serializable, java.util.Vector. The
// stub method then returns a pointer to a java::util::Vector.
::CORBA::Short initialElement = 999;
java::util::Vector *vPrt;
    try
    {
       vPrt = vector_obj ->createVector(initialElement);
       if (vPrt == 0)
       {
          VtlUtil::debug("In testVector: vector_obj ->createVector(arg) returned a null pointer\n");
       }
    } catch (...)
       {
          VtlUtil::debug("In testVector: vector_obj ->createVector(arg) has thrown an exception\n");
       }
//Next use the remote object to a method of  ::java::util::Vector.
/***********************
 * Create and populate a
 * java::lang::Object
 ***********************/
short inValue = 999;
::CORBA::Long incrementValue = 1;
java::lang::Object obj;
obj <<= inValue;   //rvalue is a ptr
/*********************************************************
 * Call the addElement method using the pointer obtaned remotely
 * via the createVector method. Add "numberToAdd" elements.
 *  Verify that the correct size is returned
 *********************************************************/
::CORBA::Long numberToAdd = 5;
    for (int i = 0; i < numberToAdd; i ++) {
       try {
          vPrt->addElement (obj);
          } catch (...)
        {
VtlUtil::debug("In testVector: In catch after pwPtr->addElement()\n");
        }
    }
```

Example: A client program that uses a remote object to call methods of ::java::lang::Boolean

```
using namespace com::ibm::ws;
const char *factoryName = "java::lang::Boolean_init";
// Use the utility method, com::ibm::ws::VtlUtil::getBooleanFactory to get a pointer to the registered
// java:lang::Boolean_init factory object.
java::lang::Boolean_init* fact = VtlUtil::getBooleanFactory();
if(fact == 0)
{
  VtlUtil::debug("VtlUtil::getFactory returned a null value ");
}
else
{
  VtlUtil::debug("VtlUtil::getFactory returned a valid value ");
}
//call create__boolean  to create a pointer to a java::lang::Boolean that contains true
java::lang::Boolean* booleanPtr = fact->create__boolean(1);
if(booleanPtr == 0)
{
        VtlUtil::debug("booleanPtr == 0");
        return failed;
}
```

```
else
{
        VtlUtil::debug("create__boolean returned a valid value: test succeeded");
}
CORBA::Object_var boolean _obj;
 ::CORBA::Boolean trueBooleanValue = boolean_obj->callBooleanValue(booleanPtr);
 int tempTrueBooleanValue = trueBooleanValue;
 if (tempTrueBooleanValue == 1)
 {
   VtlUtil::debug("tempTrueBooleanValue == 1");
}
```

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing IBM Corporation North Castle DriveArmonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:
IBM World Trade Asia Corporation Licensing 2-31Roppongi 3-chome, Minato-ku Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS DOCUMENT "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the document. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:
IBM Corporation Department LZKS 11400 Burnet Road Austin, TX 78758 U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM International Program License Agreement or

any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks and service marks

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States, other countries, or both:

Advanced Peer-to-Peer Networking AFS AIX APPN AS/400 CICS CICS OS/2 CICS/400 CICS/6000 CICS/ESA CICS/MVS CICS/VSE CICSPlex DB2 DB2 Universal Database DCE Encina Lightweight Client DFS Encina IBM IBM System Application Architecture IMS IMS/ESA Language Environment	***	MQSeries MVS/ESA NetView Open Class OS/2 OS/390 OS/400 Parallel Sysplex PowerPC RACF RAMAO RMF RISC System/6000 RS/6000 S/390 SAA SecureWay TeamConnection Transarc TXSeries VSE/ESA VTAM VisualAge WebSphere