



# Micro Focus VisiBroker-RT for C++ Version 6.0

Developer's Guide

Micro Focus  
The Lawn  
22-30 Old Bath Road  
Newbury, Berkshire RG14 1QN  
UK  
<http://www.microfocus.com>

© Copyright 2020 Micro Focus or one of its affiliates.

MICRO FOCUS, the Micro Focus logo and VisiBroker are trademarks or registered trademarks of Micro Focus or one of its affiliates.

All other marks are the property of their respective owners.

2020-11-06

# Contents

<b>Preface .....</b>	<b>13</b>
What's new .....	13
What's in this guide? .....	14
Manual conventions .....	16
Contacting Micro Focus .....	17
Further Information and Product Support .....	17
Information We Need .....	17
Contact information .....	18
<b>Introducing VisiBroker-RT for C++ .....</b>	<b>3</b>
What is CORBA? .....	3
What is VisiBroker-RT for C++? .....	4
VisiBroker-RT for C++ Features .....	4
VisiBroker-RT for C++ Smart Agent architecture .....	4
Enhanced object discovery with the Location Service .....	5
Implementation and object activation support .....	5
Robust thread and connection management .....	5
IDL compilers .....	5
Dynamic invocation with DII and DSI .....	6
Interface repositories .....	6
Server-side portability .....	6
Customizing the ORB with interceptors and object wrappers .....	6
VisiBroker-RT for C++ CORBA compliance .....	7
VisiBroker-RT for C++ Development Environment .....	7
Administration tools .....	7
Developer's tools .....	7
VisiBroker-RT for C++ header files .....	7
VisiBroker ORB Libraries .....	7
VisiBroker Sample Applications .....	8
Interoperability with VisiBroker for Java .....	8
Interoperability with other ORB products .....	8
<b>Setting up the Development Environment .....</b>	<b>9</b>
Setting the VBROKERDIR Environment Variable .....	9
Setting VBROKERDIR on a Windows platform .....	9
Setting VBROKERDIR on a UNIX platform .....	9
Setting the Path environment variable .....	9
Updating the PATH on Windows .....	10
Setting the Path on a UNIX platform .....	10
Setting VBROKER_ADM Environment Variable .....	10
Setting VBROKER_ADM on a Windows platform .....	10
Setting VBROKER_ADM on a UNIX platform .....	10
Setting OSAGENT_PORT environment variable .....	10
Setting OSAGENT_PORT on a Windows platform .....	11
Setting OSAGENT_PORT on a UNIX platform .....	11
Logging Output on the Host System .....	11
<b>Developing an Example Application with VisiBroker-RT for C++ .....</b>	<b>13</b>
Development Process .....	13
Step 1: Defining object interfaces .....	14
Writing the account interface in IDL .....	15
Step 2: Generating client stubs and server servants .....	15
Files produced by the idl compiler .....	15
Step 3: Implementing the client .....	16

corba_init.C.....	16
client.C.....	18
Step 4: Implementing the server .....	20
server.C.....	20
Understanding the Account class hierarchy .....	22
Step 5: Building the example .....	22
Step 6: Integrating VisiBroker-RT with Tornado/VxWorks .....	23
The VisiBroker-RT Runtime .....	23
Integrating VisiBroker Libraries with Tornado 2.x.....	28
Integrating VisiBroker Libraries with Tornado 2.2.....	29
Using VisiBroker with VxSim .....	33
Step 7: Starting the Smart Agent (osagent) Service .....	37
Starting the server .....	39
Running the client .....	40

## **Handling Exceptions ..... 41**

Exceptions in the CORBA model .....	41
System exceptions .....	41
Obtaining completion status .....	42
Getting and setting the minor code.....	42
Determining the type of a SystemException.....	42
Catching system exceptions.....	43
Downcasting exceptions to a system exception.....	44
User exceptions .....	45
Defining user exceptions .....	45
The VisiBroker C++ Exception Support .....	47
The Exception Macros .....	47

## **Server basics ..... 51**

Overview .....	51
Initializing the ORB .....	51
Creating the POA .....	51
Obtaining a reference to the root POA.....	52
Creating the child POA.....	52
Implementing servant methods.....	52
Activating the POA .....	54
Activating objects .....	55
Complete example .....	55

## **Using POAs ..... 61**

What is a Portable Object Adapter? .....	61
POA terminology .....	62
Steps for creating and using POAs .....	62
POA policies .....	63
Minimum CORBA and POA Policies .....	63
Thread policy.....	63
Lifespan policy .....	63
Object ID Uniqueness policy .....	64
ID Assignment policy .....	64
Servant Retention policy .....	64
Request Processing policy.....	65
Implicit Activation policy .....	65
Bind Support policy .....	66
Server Engine policy.....	66
Creating POAs .....	66
POA naming convention .....	66
Obtaining the rootPOA .....	67

Setting the POA properties.....	67
Creating and activating the POA.....	67
Activating objects .....	68
Activating objects explicitly .....	68
Activating objects on demand.....	69
Activating objects implicitly .....	69
Activating with the default servant .....	70
Deactivating objects.....	71
Using servants and servant managers .....	71
ServantActivators .....	72
ServantLocators .....	75
Managing POAs with the POA manager .....	77
Getting the current state .....	78
Holding state .....	78
Active state .....	78
Discarding state .....	78
Inactive state.....	79
Adapter activators .....	79
Processing requests .....	80
<b>Using the Tie Mechanism .....</b>	<b>81</b>
How does the tie mechanism work? .....	81
Example program .....	81
Location of an example program using the tie mechanism.....	81
Looking at the tie template .....	81
Changing the server to use the <code>_tie_account</code> class .....	83
Building the tie example .....	84
<b>Client basics .....</b>	<b>87</b>
Initializing the ORB .....	87
Binding to objects .....	87
Invoking operations on an object .....	89
Manipulating object references .....	89
Checking for nil references.....	89
Obtaining a nil reference.....	90
Duplicating an object reference .....	90
Releasing an object reference.....	90
Obtaining the reference count .....	91
Converting a reference to a string .....	91
Obtaining object and interface names.....	92
Determining the type of an object reference.....	92
Determining the location and state of bound objects.....	93
Checking for non-existent objects .....	93
Narrowing object references.....	93
Widening object references .....	94
Using Quality of Service .....	94
Understanding Quality of Service .....	94
QoS interfaces.....	94
<b>Using the VisiBroker-RT for C++ Console .....</b>	<b>103</b>
What is the VisiBroker Console? .....	103
Navigating the VisiBroker Console .....	104
Menu bar.....	105
Toolbar.....	105
Status bar .....	106
Pull down or context menus .....	106
Navigation pane .....	106

Content pane.....	106
Supported ORB Services .....	106
Location Service.....	106
Naming Services .....	107
Interface Repositories .....	107
Implementation Repositories .....	107
Server Manager .....	107
GateKeeper .....	107
Integrated Transaction Services .....	108
Starting the VisiBroker Console .....	108
VisiBroker Console main menu .....	109
Console menu.....	109
View menu .....	109
Help menu .....	109
Setting the VisiBroker Console preferences .....	110
General tab .....	111
Security tab .....	112
State tab .....	112
Tools tab.....	113

## **Setting Properties ..... 115**

Overview .....	115
Setting Properties Through the Property Manager Interface .....	115
Environment variables .....	116
Setting Properties Through the Command-Line .....	117
Setting Properties Through a Property Table .....	117
ORB Default Properties .....	118

## **Using the IDL compiler ..... 121**

Introduction to IDL .....	121
How the IDL compiler generates code .....	121
Example IDL specification.....	121
Looking at code generated for clients.....	121
Methods (stubs) generated by the IDL compiler.....	122
Pointer type <interface name>_ptr definition.....	122
Automatic memory management <interface name>_var class.....	123
Looking at code generated for CORBA server implementations .....	123
The PortableServer_RefCountServantBase class .....	124
The PortableServer_ServantBase class .....	124
Methods (skeletons) generated by the IDL compiler .....	125
Class template generated by the IDL compiler.....	125
Defining interface attributes in the IDL .....	126
Specifying oneway methods with no return value .....	126
Specifying an interface in IDL that inherits from another interface .....	127

## **Using the Smart Agent ..... 129**

What is the Smart Agent? .....	129
Locating Smart Agents.....	129
Locating objects through Agent cooperation.....	129
Starting a Smart Agent (osagent).....	130
Ensuring Agent availability .....	133
Working within ORB domains .....	134
Connecting Smart Agents on different local networks .....	135
Use of the OSAGENT_ADDR_FILE Environment Variable (applicable on Development Host systems only) .....	135
Use of the OSAGENT_ADDR_TABLE By Smart Agents (applicable on VxWorks Target systems only).....	136

How Smart Agents detect each other.....	137
Working with multihomed hosts .....	137
Specifying interface usage for Smart Agents.....	138
Using point-to-point communications .....	140
Specifying a host as a runtime parameter .....	140
Specifying an IP address with an environment variable .....	141
Specifying hosts with the agentaddr table .....	141
Ensuring object availability .....	141
Invoking methods on stateless objects.....	142
Achieving fault-tolerance for objects that maintain state .....	142
Migrating objects between VisiBrokerRT60 Systems .....	142
Migrating objects that maintain state.....	142
Migrating instantiated objects.....	142
Reporting all objects and services .....	143

## **Using the Location Service ..... 145**

What is the Location Service? .....	145
Location Service components .....	146
What is the Location Service agent? .....	146
What is a trigger? .....	149
Querying an agent .....	150
Finding all instances of an interface.....	150
Finding everything known to Smart Agents.....	151
Writing and registering a trigger handler .....	153
Implementing and registering a trigger handler .....	153

## **Using the Naming Service ..... 157**

Overview .....	157
Understanding the namespace .....	158
Naming contexts .....	158
Names and NameComponent .....	159
Name resolution .....	159
Running the Naming Service .....	160
Integrating the Naming Service into your application .....	161
Compiling and linking programs.....	161
Sample programs .....	162
Starting the Naming Service.....	162
Bootstrapping a Naming Service .....	164
Calling resolve_initial_references .....	164
Using -ORBInitRef.....	165
-ORBDefaultInitRef .....	167
NamingContext .....	167
NamingContextExt .....	168
Default naming contexts .....	168
Obtaining the default context .....	168
Binding a name in C++ .....	169
Resolving a name in C++ .....	170

## **Using the Event Service ..... 173**

Overview .....	173
Proxy consumers and suppliers.....	174
OMG common object services specification .....	174
Communication models .....	175
Push model.....	175
Pull model .....	176
Using event channels .....	177
Example push supplier and consumer .....	178

Deriving a PushSupplier class .....	178
Deriving a PushConsumer class .....	181
Implementing the PushConsumer .....	181
Starting the Event Service .....	183
Installing the Event Service .....	183
Integrating the Naming Service into your application .....	184
Setting the queue length.....	184
Compiling and linking programs .....	185
Interface reference .....	185
EventChannel .....	185
ConsumerAdmin .....	185
SupplierAdmin .....	186
ProxyPullConsumer.....	186
ProxyPushConsumer .....	186
ProxyPullSupplier .....	187
ProxyPushSupplier.....	187
PullConsumer .....	187
PushConsumer.....	187
PullSupplier.....	188
PullSupplier methods.....	188
PushSupplier .....	189

## **Real-Time CORBA Extensions ..... 199**

Overview .....	199
Using the Real-Time CORBA Extensions .....	200
Real-Time CORBA ORB .....	200
Real-Time Object Adapters .....	202
Real-Time CORBA Priority .....	203
Priority Mappings .....	203
Priority Mapping Types.....	204
Rules for Priority Mappings .....	205
Default Priority Mapping.....	206
Replacing the Default Priority Mapping .....	207
Using Native Priorities in VisiBroker Application Code .....	208
Threadpools .....	208
Threadpool API .....	209
Threadpool Creation and Configuration.....	209
Association of an Object Adapter with a Threadpool .....	210
The General Threadpool .....	211
Threadpool Destruction .....	212
Real-Time CORBA Current .....	212
Real-Time CORBA Priority Models .....	213
Client Model Backwards Compatability with VisiBroker 3.2.2.....	215
Setting Priority at the Object Level .....	215
Real-Time CORBA Mutex API .....	216
Control of Internal ORB Thread Priorities .....	216
Limiting the Internal ORB Thread Priority Range .....	217
Configuring Individual Internal ORB Thread Priorities .....	218
Protocol Configuration Policies .....	219
ServerProtocolPolicy .....	219
ClientProtocolPolicy .....	221

## **Listening and Dispatch Configuration ..... 223**

Overview .....	223
When to Configure Listening and Dispatching .....	223
Listening and Dispatch Architecture .....	223
Interaction of an SCM and Threadpool during Dispatch .....	224

Server Engines and SCM Configuration .....	228
Required Server Engine and SCM Properties .....	228
Optional Server Engine Properties .....	229
Optional SCM Properties .....	229
Server Engine and SCM Creation .....	230
Associating a POA with Server Engines .....	230
Default Server Engines .....	231
Restriction on POA/Server Engine Relationship .....	231
Code Example .....	231
<b>Connection Management .....</b>	<b>235</b>
VisiBroker Default Connection Behavior of VisiBrokerRT60 .....	235
Overriding the Default Behavior with _clone() .....	236
Limiting the Number of Connections .....	236
Limiting Connections on the Server-Side .....	236
Limiting Connections on the Client-Side .....	237
<b>Bidirectional Communication .....</b>	<b>239</b>
Using bidirectional IIOP .....	239
Bidirectional ORB properties .....	239
About the examples .....	240
Enabling bidirectional IIOP for existing applications .....	241
Security considerations .....	241
<b>VisiBroker Pluggable Transport Interface .....</b>	<b>243</b>
Pluggable Transport Interface Files .....	243
Transport Layer Requirements .....	243
User-Provided Code Required for a Protocol Plugin .....	244
Unique Profile ID Tag .....	245
Example Code .....	245
Implementing a New Transport .....	246
Connection Class .....	246
Connection Factory Class .....	248
Listener Class .....	249
Listener Factory Class .....	250
Profile Class .....	251
Profile Factory Class .....	252
Classes Provided by the Interface .....	253
Transport Bridge Class .....	253
Transport Registrar Class .....	254
Creating a Loadable Library .....	254
<b>Using Portable Interceptors .....</b>	<b>255</b>
Overview .....	255
Portable Interceptor and Information interfaces .....	256
Request Interceptor .....	256
IOR Interceptor .....	259
Portable Interceptor Current .....	260
Codec .....	260
CodecFactory .....	260
Creating a Portable Interceptor .....	261
Registering Portable Interceptors .....	261
Registering an ORBInitializer .....	262
VisiBroker Edition Extensions to Portable Interceptors .....	263
Limitations of VisiBroker Edition Portable Interceptors Implementation .....	263
Examples .....	264
Example Code .....	264

Example: client_server .....	264
------------------------------	-----

## **Using VisiBroker Interceptors ..... 281**

Overview .....	281
Interceptor interfaces and managers .....	281
Client interceptors .....	282
Server interceptors.....	283
Registering interceptors with the VisiBroker-RT for C++ ORB .....	284
Creating interceptor objects.....	285
Loading interceptors .....	285
Example interceptors .....	286
Example code .....	286
Code listings.....	288
Passing information between your interceptors .....	293
Using both Portable Interceptors and Interceptors simultaneously .....	293
Order of invocation of interception points .....	293
Client side interceptors .....	293
Server side Interceptors.....	293
Order of ORB events during POA creation .....	294
Order of ORB events during object reference creation .....	294

## **Using Object Wrappers ..... 295**

Overview .....	295
Typed and un-typed object wrappers .....	295
Special idl2cpp requirements .....	296
Example applications .....	296
Un-typed object wrappers .....	296
Using multiple, un-typed object wrappers .....	297
Order of pre_method invocation .....	297
Order of post_method invocation.....	298
Using un-typed object wrappers .....	298
Implementing an un-typed object wrapper factory .....	298
Implementing an un-typed object wrapper .....	299
Creating and registering un-typed object wrapper factories.....	300
Removing un-typed object wrappers .....	302
Typed object wrappers .....	302
Using multiple, typed object wrappers.....	303
Order of invocation.....	304
Typed object wrappers with co-located client and servers .....	304
Using typed object wrappers .....	304
Implementing typed object wrappers .....	305
Registering typed object wrappers for a client .....	305
Registering typed object wrappers for a server .....	306
Removing typed object wrappers.....	307
Combined use of un-typed and typed object wrappers .....	308
Command-line arguments for typed wrappers .....	308
Initializer for typed wrappers .....	308
Command-line arguments for un-typed wrappers .....	309
Initializers for un-typed wrappers .....	310
Executing the sample applications .....	311

## **Using Valuetypes ..... 313**

Understanding valuetypes .....	313
Concrete valuetypes .....	313
Abstract valuetypes .....	314
Implementing valuetypes .....	314
Defining your valuetypes.....	314

Compiling your IDL file .....	315
Inheriting the valuetype base class .....	315
Implementing the Factory class .....	315
Registering your Factory with the ORB .....	316
Implementing factories .....	316
Factories and valuetypes.....	316
Registering valuetypes .....	316
Boxed valuetypes .....	317
Abstract interfaces .....	317
Custom valuetypes .....	318
Truncatable valuetypes .....	318
<b>VisiBroker Logging .....</b>	<b>319</b>
Logging Overview .....	319
The Logger Manager .....	319
Configuring ORB Logging .....	320
ORB Log Levels .....	320
ORB Logging Components.....	320
Controlling the Level of ORB Logging .....	321
Library liblog_message_catalog.o and Formatted ORB Log Messages .....	322
Controlling the Priority of ORB Logging .....	322
Enabling Forwarding of ORB Logging .....	323
Controlling the Destination of ORB Logging .....	323
Application Logging .....	323
Creating or Obtaining a Reference to a Logger .....	323
Setting the Forwarder Thread Priority of a Logger.....	324
Enabling Message Forwarding.....	325
Logging a Message to a Logger.....	326
Adding and Removing Logger Forwarders.....	327
Implementing a Logger Forwarder.....	328
The Default Logger Forwarder .....	330
<b>Using Interface Repositories .....</b>	<b>335</b>
What is an interface repository? .....	335
What does an interface repository contain? .....	336
How many interface repositories can you have? .....	336
Creating and viewing an interface repository with irep .....	336
Creating an interface repository with irep.....	337
Viewing the contents of the interface repository .....	338
Updating an interface repository with idl2ir .....	338
Understanding the structure of the interface repository .....	338
Identifying objects in the interface repository .....	339
Types of objects that can be stored in the interface repository .....	339
Inherited interfaces.....	340
Accessing an interface repository .....	341
Example programs .....	341
<b>Using the Dynamic Invocation Interface .....</b>	<b>347</b>
What is the Dynamic Invocation Interface? .....	347
Introducing the main DII concepts .....	348
Steps for invoking object operations dynamically .....	350
Location of example programs for using the DII .....	351
Obtaining a generic object reference .....	351
Creating and initializing a request .....	351
Request class.....	351
Ways to create and initialize a DII request .....	352
Using the create_request method .....	352

Using the _request method .....	352
Example of creating a Request object.....	353
Setting the context for the request .....	353
Setting arguments for the request.....	354
Passing type safely with the Any class.....	355
Representing argument or attribute types with the TypeCode class .....	355
Sending DII requests and receiving results .....	357
Invoking a request .....	357
Sending a deferred DII request with the send_deferred() method.....	358
Sending an asynchronous DII request with the send_oneway method .....	359
Sending multiple requests .....	359
Receiving multiple requests .....	360
Using the interface repository with the DII .....	361

## **Using the Dynamic Skeleton Interface ..... 365**

What is the Dynamic Skeleton Interface? .....	365
Steps for creating object implementations dynamically .....	366
Location of an example program for using the DSI.....	366
Extending the DynamicImplementation class .....	366
Example of designing objects for dynamic requests.....	367
Specifying repository ids .....	369
Looking at the ServerRequest class .....	369
Implementing the Account object .....	370
Implementing the AccountManager object .....	370
Server implementation .....	371

## **Using the Dynamically Managed Types ..... 375**

Overview .....	375
DynAny types .....	375
Usage restrictions .....	376
Creating a DynAny .....	376
Initializing and accessing the value in a DynAny .....	376
Constructed data types .....	377
DynEnum.....	377
DynStruct .....	377
DynUnion.....	378
DynSequence and DynArray .....	378
Example IDL .....	378
Example client application .....	378
Example server application .....	380

## **Using the BOA in VisiBroker-RT for C++ 6.0 ..... 389**

Compiling your BOA code with VisiBroker-RT for C++ 6.0 .....	389
Supporting BOA options .....	389
Using object activators .....	389
Naming Objects under the BOA .....	389
Object names .....	389

## **Migrating VisiBroker Code ..... 391**

Migrating BOA to POA .....	391
Looking at an example.....	391
Mapping BOA types to POA policies.....	393
Migrating interceptors .....	393

<b>CORBA exceptions .....</b>	<b>397</b>
<b>Glossary .....</b>	<b>405</b>



# Preface

VisiBroker-RT for C++ allows you to develop and deploy distributed object-based applications, as defined in the Common Object Request Broker Architecture (CORBA) specification.

The VisiBroker-RT for C++ *Developer's Guide* provides you information on how to get started with the VisiBroker-RT for C++ fundamentals, use the VisiBroker-RT for C++ Console to simplify certain functions, and work with the more advanced features. It is written for C++ programmers who are familiar with object-oriented development.

This chapter highlights the latest features, and identifies typographical and platform conventions used throughout the manual. It also tells you where to find additional information about Common Object Request Broker Architecture (CORBA) and the remaining VisiBroker-RT for C++ documentation set, as well as how to contact Borland Software developer support.

## What's new

This manual has been updated to reflect the latest VisiBroker-RT for C++ release. The new features and enhancements include:

- **CORBA 2.5 compliance:** VisiBroker-RT for C++ is fully compliant with the CORBA specification (version 2.5) from the Object Management Group (OMG). For more details, refer to the CORBA specification located at <http://www.omg.org>.
- **Minimum CORBA 1.0 compliance.** VisiBroker-RT for C++ is fully compliant with the Minimum CORBA specification (version 1.0) from the Object Management Group (OMG). For more details, refer to the Minimum CORBA specification located at <ftp://ftp.omg.org/pub/docs/orbos/98-08-04.pdf>

***Is this "Minimum spec" now called something else? Could Dev give me an up to date link, please?***

- **Real-Time CORBA 1.0 compliance.** VisiBroker-RT for C++ is fully compliant with the Real-Time CORBA specification (version 1.0) from the Object Management Group (OMG). For more details, refer to the Real-Time CORBA specification located at <https://www.omg.org/spec/RT/1.2/PDF>.
- **Naming Service:** The new VisiBroker-RT for C++ Naming Service. Provides support for the OMG Interoperable Naming Service specification. The `corbaloc` and `corbaname` functionality supports stringified object references which can be used in an Internet environment. This allows you to refer to objects by a URL. See "Using the Naming Service" for a description of how to use the Naming Service.
- **Portable Object Adaptor (POA):** The POA offers portability on the server side. This feature replaces the Basic Object Adapter (BOA). Although BOA is being deprecated, VisiBroker-RT for C++ 6.0 will still support BOA functionality. See "Using POAs" for an explanation of how to use the POA.
- **Objects by Value (OBV) or Value types:** Previous versions of CORBA allowed you to pass objects between clients and servers by reference. However, CORBA 2.3 allows you to pass objects by value between clients and servers using VisiBroker-RT for C++. OBV is

interoperable with other 2.3-compliant ORBs. See [“Using Valuetypes”](#) for more information on this feature.

- **Property Management:** This feature provides you with a way to centralize management of properties. Using Property Management, you can get/set the value of configurable properties of VisiBroker. See [“Setting Properties”](#) for more information on Property Management.
- **Quality of Service (QoS):** This feature, which implements the CORBA 2.3 Messaging Specification, allows you to define policies that influence how connections are made. You perform client-side policy management by setting properties that are associated with connections or client/server pairs. See [“Using Quality of Service”](#) for a description of the VisiBroker-RT for C++ 6.0 QoS features.
- **Interceptors and Object Wrappers:** The ORB provides a set of APIs known as interceptors which provide a way to plug-in additional ORB behavior such as support for transactions and security, which may be defined on either the client or server side. One of the main difference in this release is that now the interceptors have scope. See [“Using Portable Interceptors”](#) for more information on how to use the VisiBroker-RT for C++ 6.0 interceptor.
- **Pluggable Transport Interface:** This feature provides support for the use of transport protocols besides TCP for the transmission of CORBA invocations. The Interface supports the ‘plugging-in’ of multiple transport protocols simultaneously and is designed to provide a common interface that is suitable for use with a wide variety of transport types. The interface uses CORBA standard classes wherever possible, but is itself VisiBroker proprietary.
- **VisiBroker Logging:** This feature allows applications to log messages and have them directed, via configurable logging forwarders, to an appropriate destination or destinations. The ORB itself uses this mechanism for the output of any error, warning or informational messages. The application can choose to log its and the ORB’s messages to the same destination, producing a single message log for the entire system, or to log messages from different sources to independent destinations.

Other updates include new sample code snippets to reflect the new VisiBroker-RT for C++ features. New interfaces and methods are covered in the VisiBroker-RT for C++ *Programmer’s Reference Guide*.

## What’s in this guide?

This programmer’s guide provides detailed information on developing distributed object-based applications using VisiBroker-RT for C++. It contains the following sections:

- [Part 1, “Basic Concepts”](#)

This part presents an introduction to VisiBroker-RT for C++. It also includes an overview of the CORBA model and a quick start example designed to introduce you to the VisiBroker-RT for C++ development principles and the handling of exceptions.

- [Part 2, “Server Concepts”](#)

This part describes how to develop a VisiBroker-RT for C++ server, use the Portable Object Adapter (POA) and the tie mechanism.

- [Part 3, “Client Concepts”](#)

This part describes how to develop a VisiBroker-RT for C++ client.

- **Part 4, “Configuration and Management”**

This part is designed to familiarize you with the configuration and management of the VisiBroker-RT for C++ ORB and its CORBA services, using the Console and its associated browsers. This allows you to perform many of your configuration tasks in one location that previously were performed on the command line. From the Console, you can access browsers for the ORB services, repositories and the Server Managers. From this central location, you can easily view, monitor, and manage VisiBroker-RT for C++ Services, Servers and Objects. The configuration of VisiBroker-RT for C++ using properties files is also described.

- **Part 5, “Tools and Services”**

This part describes the IDL compiler, the Smart Agent, the Location, Event, and Naming services.

- **Part 6, “Advanced Concepts”**

This part describes advanced concepts such as the Real Time CORBA Extensions, configuration of the Listening and Dispatching of CORBA invocations, Connection Management, Pluggable Transports, Portable Interceptors, Object Wrappers, Value Types and the VisiBroker Logging Service.

- **Part 7, “Dynamic CORBA Concepts”**

This part describes the Dynamic features of CORBA. These concepts are the Interface Repository, the Dynamic Invocation Interface, the Dynamic Skeleton Interface and the DynAny class.

- **Part 8, “Backward Compatibility”**

This part describes compatibility issues between previous releases of VisiBroker-RT for C++ and the current one.

- **“CORBA exceptions”**

This Appendix contains additional information about CORBA exceptions that can be thrown by the VisiBroker ORB, and explains possible causes for VisiBroker-RT for C++ to throw them.

- **“Glossary”**

Provides a glossary of commonly used terms.

# Manual conventions

This section identifies the VisiBroker-RT for C++ *Programmer's Reference Guide's* typographical and platform conventions.

## Typographic conventions

This manual uses the following conventions:

Convention	Used for
<b>Boldface</b>	Bold type indicates that syntax should be typed exactly as shown. For UNIX, used to indicate database names, file names, and similar terms.
<i>italics</i>	Italics indicates information that the user or application provides, such as variables in syntax diagrams. It is also used to introduce new terms.
<code>computer</code>	Computer typeface is used for sample command lines and code.
<b>bold computer</b>	In code examples, important statements appear in boldface
UPPERCASE	Uppercase letters indicate Windows file names.
[ ]	Brackets indicate optional items.
...	An ellipsis indicates the continuation of previous lines of code or that the previous argument can be repeated.
	A vertical bar separates two mutually exclusive choices.

## Platform conventions

This manual uses the following conventions—where necessary—to indicate that information is platform-specific:

Convention	Used for
<b>Windows</b>	All Windows (Windows NT, Windows 2000, Windows XP) development hosts
<b>WinNT</b>	Windows NT development host platform
<b>Win2000</b>	Windows 2000/XP development host only
<b>UNIX</b>	All UNIX development host platforms including Solaris
<b>Solaris</b>	Solaris development host only
<b>Tornado</b>	VisiBroker-RT for C++ for Tornado only
<b>C++</b>	VisiBroker-RT for C++

## VisiBroker Library conventions

This manual uses the following conventions—where necessary—to indicate that information is VisiBroker library specific or to indicate that VisiBroker interfaces are not supported in certain versions of the VisiBroker libraries.



Not Supported in the VisiBroker-RT Minimum Corba Library

## Where to find additional information

For more information about VisiBroker-RT for C++, refer to these information sources:

- VisiBroker-RT for C++ *Release Notes* contain late-breaking information about the current release of VisiBroker-RT for C++.

- VisiBroker-RT for C++ for Tornado. This guide contains the instructions for installing VisiBroker-RT for C++ on Windows and UNIX host systems as well as information for deploying distributed applications built using VisiBroker-RT for C++.
- VisiBroker-RT for C++ *Programmer's Reference Guide*. This manual contains information on the VisiBroker-RT for C++ C++ Application Programmers Interfaces (API).
- For more information about the CORBA specification, refer to *The Common Object Request Broker: Architecture and Specification*. This document is available from the Object Management Group and describes the architectural details of CORBA.

You can access the CORBA specification at the OMG web site:  
<https://www.omg.org/>.

## Contacting Micro Focus

***Old version has been replaced with the standard MF section.***

Our Web site gives up-to-date details of contact numbers and addresses.

### Further Information and Product Support

Additional technical information or advice is available from several sources.

The product support pages contain a considerable amount of additional information, such as:

- The *Product Updates* section of the Micro Focus SupportLine Web site, where you can download fixes and documentation updates.
- The *Examples and Utilities* section of the Micro Focus SupportLine Web site, including demos and additional product documentation.

To connect, enter <http://www.microfocus.com> in your browser to go to the Micro Focus home page, then click *Support*.

#### Note:

Some information may be available only to customers who have maintenance agreements.

If you obtained this product directly from Micro Focus, contact us as described on the Micro Focus Web site, <http://www.microfocus.com>. If you obtained the product from another source, such as an authorized distributor, contact them for help first. If they are unable to help, contact us.

Also, visit:

- The Micro Focus Community Web site, where you can browse the Knowledge Base, read articles and blogs, find demonstration programs and examples, and discuss this product with other users and Micro Focus specialists.
- The Micro Focus YouTube channel for videos related to your product.

### Information We Need

However you contact us, please try to include the information below, if you have it. The more information you can give, the better Micro Focus SupportLine can help you. But if you don't know all the answers, or you

think some are irrelevant to your problem, please give whatever information you have.

- The name and version number of all products that you think might be causing a problem.
- Your computer make and model.
- Your operating system version number and details of any networking software you are using.
- The amount of memory in your computer.
- The relevant page reference or section in the documentation.
- Your serial number. To find out these numbers, look in the subject line and body of your Electronic Product Delivery Notice email that you received from Micro Focus.

## Contact information

Our Web site gives up-to-date details of contact numbers and addresses. Additional technical information or advice is available from several sources.

The product support pages contain considerable additional information, including the *Product Updates* section of the Micro Focus SupportLine Web site, where you can download fixes and documentation updates. To connect, enter <http://www.microfocus.com> in your browser to go to the Micro Focus home page, then click *Support*.

If you are a Micro Focus SupportLine customer, please see your SupportLine Handbook for contact information. You can download it from our Web site or order it in printed form from your sales representative. Support from Micro Focus may be available only to customers who have maintenance agreements.

You may want to check these URLs in particular:

- <https://www.microfocus.com/products/corba/visibroker/> (VisiBroker trial software)
- <https://supportline.microfocus.com/login.aspx> (Micro Focus support login)
- <https://supportline.microfocus.com/productdoc.aspx> (documentation updates and PDFs)

To subscribe to Micro Focus electronic newsletters, use the online form at: <https://software.microfocus.com/en-us/select/email-subscription>

# Part 1

## Basic Concepts

### **In this part**

This part contains the following chapters:

<a href="#">Introducing VisiBroker-RT for C++</a>	<a href="#">page 3</a>
<a href="#">Setting up the Development Environment</a>	<a href="#">page 9</a>
<a href="#">Developing an Example Application with VisiBroker-RT for C++</a>	<a href="#">page 13</a>
<a href="#">Handling Exceptions</a>	<a href="#">page 41</a>



# Introducing VisiBroker-RT for C++

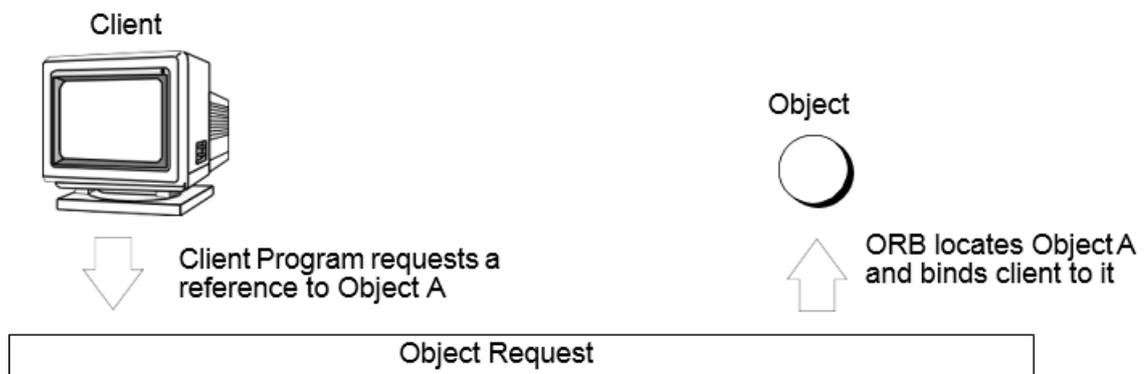
*This chapter introduces VisiBroker-RT for C++, a complete implementation of the CORBA 2.5 specification. This chapter describes VisiBroker-RT for C++ features and components.*

## What is CORBA?

The Common Object Request Broker Architecture (CORBA) allows distributed applications to interoperate (application to application communication), regardless of what language they are written in or where these applications reside.

The CORBA specification was adopted by the Object Management Group to address the complexity and high cost of developing distributed object applications. CORBA uses an object-oriented approach for creating software components that can be reused and shared between applications. Each object encapsulates the details of its inner workings and presents a well defined interface, which reduces application complexity. The cost of developing applications is reduced, because once an object is implemented and tested, it can be used over and over again.

The Object Request Broker (ORB) in [Figure 1](#) connects a client application with the objects it wants to use. The client program does not need to know whether the object implementation it is in communication with resides on the same computer or is located on a remote computer somewhere on the network. The client program only needs to know the object's name and understand how to use the object's interface. The ORB takes care of the details of locating the object, routing the request, and returning the result.



**Figure 1** Client program acting on an object

### Note

The ORB itself is not a separate process/thread. It is a collection of libraries and network resources that integrates within end-user applications, and allows your client applications to locate and use objects.

# What is VisiBroker-RT for C++?

VisiBroker-RT for C++ provides a complete CORBA 2.3 ORB runtime and supporting development environment for building, deploying, and managing distributed C++ applications that are open, flexible, and inter-operable. Objects built with VisiBroker-RT for C++ are easily accessed by Web-based applications that communicate using OMG's Internet Inter-ORB Protocol (IIOP) standard for communication between distributed objects through the Internet or through local intranets. VisiBroker-RT for C++ has a built-in implementation of IIOP that ensures high-performance and inter-operability.

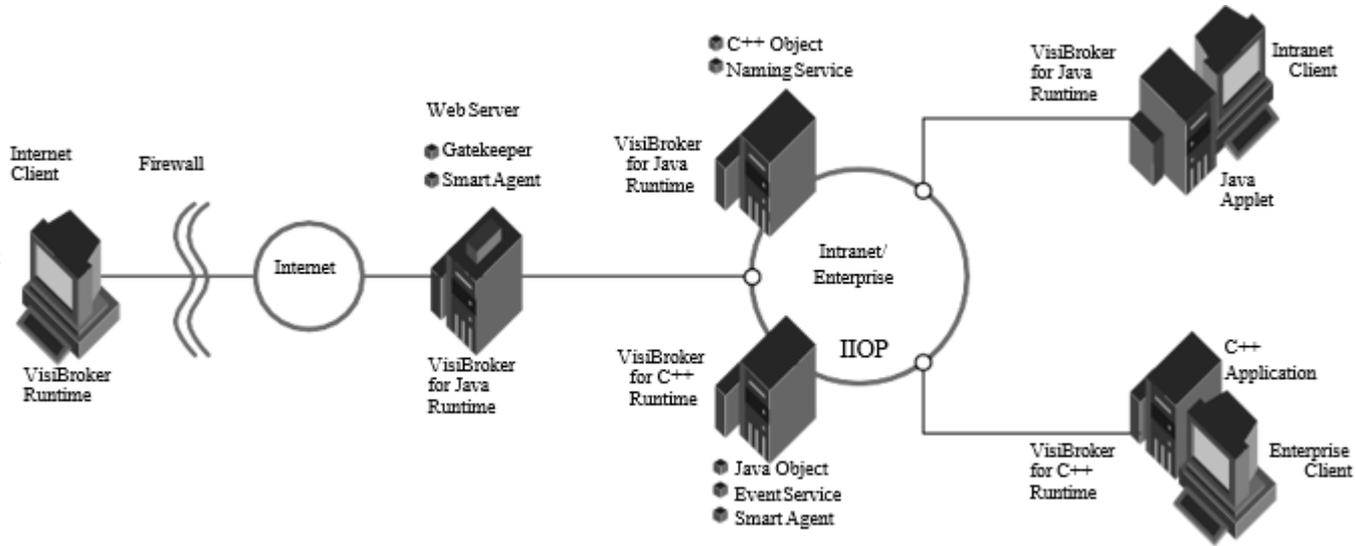


Figure 2 VisiBroker-RT for C++ architecture

## VisiBroker-RT for C++ Features

VisiBroker-RT for C++ has several key features as described in the following sections.

### VisiBroker-RT for C++ Smart Agent architecture

VisiBroker-RT for C++'s Smart Agent (`osagent`) is a dynamic, distributed directory service that provides facilities for both client applications and object implementations. Multiple Smart Agents on a network cooperate to provide load balancing and high availability for client access to server objects. The Smart Agent keeps track of objects that are available on a network, and locates objects for client applications at invocation time. VisiBroker-RT for C++ can determine if the connection between your client application and a server object has been lost, due to an error such as a server crash or a network failure. When a failure is detected, an attempt is automatically made to connect your client to another server on a different node, if it is so configured. For details on the Smart Agent, see ["Using the Smart Agent"](#) and ["Using Quality of Service"](#).

## Enhanced object discovery with the Location Service

VisiBroker-RT for C++ provides a powerful Location Service—an extension to the CORBA specification—that enables you to access the information from multiple Smart Agents.

Working with the Smart Agents on a network, the Location Service can see all the available instances of an object to which a client can bind. Using *triggers*, a callback mechanism, client applications can be instantly notified of changes to an object’s availability. Used in combination with *interceptors*, the Location Service is useful for developing enhanced load balancing of client requests to server objects. See the chapter [“Using the Location Service”](#) for more information.

## Implementation and object activation support

VisiBroker-RT for C++ provides functionality that enables you to defer object activation until a client request is received. You can defer activation for a particular object or an entire class of objects. See the chapter [“Using POAs”](#) for more information on *servant managers*.

## Robust thread and connection management

VisiBroker-RT for C++ provides native support for multithreading thread management. With VisiBroker-RT for C++’s thread pooling model, threads are allocated based on the amount of request traffic to the server object. This means that a highly active client will be serviced by multiple threads—ensuring that the requests are quickly executed—while less active clients can share a single thread, and still have their requests immediately serviced.

VisiBroker-RT for C++’s connection management minimizes the number of client connections to the server. All client requests for objects residing on the same server are multiplexed over the same connection, even if they originate from different threads.

Additionally, released client connections are recycled for subsequent reconnects to the same server, eliminating the need for clients to incur the overhead of new connections to the same server.

All thread and connection behavior is fully configurable. See the chapter [“Connection Management”](#) for details on how VisiBroker-RT for C++ manages connections.

***Originally this referred to a “Thread and connection management” chapter - which doesn’t exist. Is this the best place to point readers to?***

## IDL compilers

VisiBroker-RT for C++ comes with two IDL compilers that make object development easier:

- `idl2cpp`—The `idl2cpp` compiler takes IDL files as input and produces the necessary client stubs and server skeletons (in C++).
- `idl2ir`—The `idl2ir` compiler takes an IDL file and populates an interface repository with its contents.

(The Interface Repository is available only on the Development Host - that is, Solaris/Windows.)

See [“Using the IDL compiler”](#) and [“Using Interface Repositories”](#) for details on these compilers.

## Dynamic invocation with DII and DSI

For dynamic invocation, VisiBroker-RT for C++ provides implementations of both the Dynamic Invocation Interface (DII) and the Dynamic Skeleton Interface (DSI). The DII allows client applications to dynamically create requests for objects that were not defined at compile time. The DII is covered in the chapter [“Using the Dynamic Invocation Interface”](#). The DSI allows servers to dispatch client operation requests to objects that were not defined at compile time. See [“Using the Dynamic Skeleton Interface”](#) for complete details.

## Interface repositories

The Interface Repository (IR) is an online database of meta information about ORB objects. Meta information stored for objects includes information about modules, interfaces, operations, attributes, and exceptions. The chapter [“Using Interface Repositories”](#) covers how to start an instance of the Interface Repository, add information to an interface repository from an IDL file, and extract information from an interface repository.

### Note

The Interface Repository is available only as a Development Host utility (i.e. Solaris/ WindowsNT)

## Server-side portability

VisiBroker-RT for C++ supports the CORBA Portable Object Adapter (POA), which is a replacement to the Basic Object Adapter (BOA). The POA shares some of the same functionality as the BOA, such as activating objects, support for transient or persistent objects, and so forth. The POA also has new features, such as the POA Manager and Servant Manager which creates and manages instances of your objects. See [“Using POAs”](#) for more information.

## Customizing the ORB with interceptors and object wrappers

VisiBroker-RT for C++’s interceptors enable developers to view under-the-cover communications between clients and servers. Interceptors can be used to extend the ORB with customized client and server code that enables load balancing, monitoring, or security to meet specialized needs of distributed applications. See the chapter [“Using Portable Interceptors”](#) for information.

VisiBroker-RT for C++’s object wrappers allow you to define methods that are called when a client application invokes a method on a bound object or when a server application receives an operation request. See [“Using Object Wrappers”](#) for information.

# VisiBroker-RT for C++ CORBA compliance

VisiBroker-RT for C++ is fully compliant with the CORBA specification (version 2.3) from the Object Management Group (OMG). For more details, refer to the CORBA specification located at <http://www.omg.org/>.

## VisiBroker-RT for C++ Development Environment

VisiBroker-RT for C++ is used in both the development and deployment phases. The VisiBroker-RT for C++ development environment includes the following components:

- Administration and Development tools
- C++ header files
- VisiBroker ORB libraries (including the VisiBroker Smart Agent)
- Sample applications

### Administration tools

The following tools are used to administer the VisiBroker-RT for C++ ORB during development:

Tool	Purpose
osagent	Used to manage the Smart Agent. See <a href="#">“Using the Smart Agent”</a> .
osfind	Reports on objects running on a given network.
irep	Used to manage the Interface Repository. See <a href="#">“Using Interface Repositories”</a> .

### Developer’s tools

The following tools are used during the development phase:

Tool	Purpose
idl2ir	This tool allows you to populate an interface repository with interfaces defined in an IDL file.
idl2cpp	This tool generates C++ stubs and skeletons from an IDL file.

### VisiBroker-RT for C++ header files

The VisiBroker-RT for C++ for Tornado header files have been installed under `<install location>/VisiBrokerRT60/include`. Please refer to the section [“Development Process”](#) for a description of how to develop VisiBroker-RT for C++ for Tornado applications.

### VisiBroker ORB Libraries

The VisiBroker-RT for C++ ORB libraries enable client and server applications to use and provide distributed objects. The runtime support services is included with the VisiBroker product.

VisiBroker-RT for C++ version 6.0 provides a set of libraries for each supported CPU variant (that is for SPARC, PPC, etc). See [“Step 6: Integrating VisiBroker-RT with Tornado/VxWorks”](#) for details on how to use the VisiBroker run-time libraries.

## VisiBroker Sample Applications

VisiBroker-RT for C++ provides a set of sample applications as a starting point for the user. These sample applications can be found in the `<install location>/VisiBrokerRT60/examples` directory.

## Interoperability with VisiBroker for Java

Applications created with VisiBroker-RT for C++ can communicate with object implementations developed with VisiBroker for Java, which is sold separately. Simply use the same IDL you used to develop your C++ application as input to the VisiBroker for Java IDL compiler, supplied with VisiBroker for Java. You may then use the resulting Java skeletons to develop the object implementation.

Also, object implementations written with VisiBroker-RT for C++ will work with clients written in VisiBroker for Java. In fact, a server written with VisiBroker-RT for C++ will work with *any* CORBA-compliant client; a client written with VisiBroker-RT for C++ will work with *any* CORBA-compliant server.

## Interoperability with other ORB products

CORBA-compliant software objects communicate using the Internet Inter-ORB Protocol (IIOP) and are fully interoperable, even when they are developed by different vendors who have no knowledge of each other's implementations. VisiBroker-RT for C++'s use of IIOP allows client and server applications you develop with VisiBroker-RT for C++ to interoperate with a variety of ORB products from other vendors.

# Setting up the Development Environment

*VisiBroker-RT for C++ requires very little development host environment configuration. The following section specifies what environment variables VisiBroker uses. There are three mandatory environment variables which must be set and/or modified:*

- `VBROKERDIR`
- `PATH`
- `VBROKER_ADM`

## Setting the VBROKERDIR Environment Variable

The `VBROKERDIR` environment variable defines the directory where the VisiBroker-RT for C++ distribution was installed.

### Note

This environment variable *must* be set in order for the VisiBroker development host tools to work correctly.

## Setting VBROKERDIR on a Windows platform

Assuming that the VisiBroker distribution is installed in the directory `c:\VisiBrokerRT60` the following commands can be used for setting the `VBROKERDIR` environment variable in a DOS window:

### Windows

```
prompt> set VBROKERDIR=C:\VisiBrokerRT60
```

## Setting VBROKERDIR on a UNIX platform

If you are using `csh` and you installed the VisiBroker distribution in the default location the following commands can be used for setting the `VBROKERDIR` environment variable.

### Solaris

```
prompt> setenv VBROKERDIR $HOME/VisiBrokerRT60
```

If you are using Bourne (or BASH) shell and you installed the VisiBroker distribution in the default location the following commands can be used for setting the `VBROKERDIR` environment variable:

### Solaris

```
prompt> VBROKERDIR=$HOME/VisiBrokerRT60
prompt> export VBROKERDIR
```

## Setting the Path environment variable

The `PATH` environment variable should be set to include the bin directory which contains the VisiBroker-RT for C++ distribution. The bin directory is where the VisiBroker-RT for C++ tools/utilities for developers and users are located.

If you choose to explicitly set the `PATH` environment variable, the following sections explain how to do so.

## Updating the PATH on Windows

Assuming that the VisiBroker distribution was installed in the C: directory (C:\VisiBrokerRT60) you can set your PATH with the following DOS command.

### Windows

```
prompt> set PATH=C:\VisiBrokerRT60\bin;%PATH%
```

## Setting the Path on a UNIX platform

If you are using `cs`h and you installed the VisiBroker distribution in the default location the following commands can be used for updating the PATH environment variable.

### Solaris

```
prompt> setenv PATH $HOME/VisiBrokerRT60/bin:$PATH
```

If you are using Bourne (or BASH) shell and you installed the VisiBroker distribution in the default location the following commands can be used for updating the PATH environment variable:

### Solaris

```
prompt> PATH=$PATH:$HOME/VisiBrokerRT60/bin
```

## Setting VBROKER\_ADM Environment Variable

The `VBROKER_ADM` environment variable defines the administration directory where important configuration information for development host environment tools such as VisiBroker's interface repository and Smart Agent are stored.

## Setting VBROKER\_ADM on a Windows platform

Assuming that the VisiBroker distribution is installed in the C: directory (C:\VisiBrokerRT60) the following commands can be used for setting the `VBROKER_ADM` environment variable. You can set your `VBROKER_ADM` environment variable with the following DOS command.

### Windows

```
prompt> set VBROKER_ADM=C:\VisiBrokerRT60\adm
```

## Setting VBROKER\_ADM on a UNIX platform

If you are using `cs`h and you installed the VisiBroker distribution in the default location the following commands can be used for setting the `VBROKER_ADM` environment variable.

### Solaris

```
prompt> setenv VBROKER_ADM $HOME/VisiBrokerRT60/adm
```

If you are using Bourne (or BASH) shell and you installed the VisiBroker distribution in the default location the following commands can be used for setting the `VBROKER_ADM` environment variable:

### Solaris

```
prompt> VBROKER_ADM=$HOME/VisiBrokerRT60/adm prompt> export  
VBROKER_ADM
```

## Setting OSAGENT\_PORT environment variable

The `OSAGENT_PORT` environment variable defines the port number under which the Smart Agent will listen. By default, the Smart Agent will listen on port number 14000.

It is often desirable to have two or more separate Osagent domains running at the same time. One domain might consist of the production versions of client programs and object implementations while another domain might be made up of test versions of the same clients and objects that have not yet been released for general use. If several developers are working on the same local network, each may want to establish their own ORB domain so that their testing efforts do not conflict with one another. For details on establishing multiple Osagent domains see [“Working within ORB domains”](#).

## Setting OSAGENT\_PORT on a Windows platform

The following commands can be used for setting the `OSAGENT_PORT` environment variable. You can set your `OSAGENT_PORT` environment variable with the following DOS command.

### Windows

```
prompt> set OSAGENT_PORT=10000
```

## Setting OSAGENT\_PORT on a UNIX platform

If you are using `csh` and you want the Smart Agent to listen on port number 10000, set the `OSAGENT_PORT` environment variable as follows:

### Solaris

```
prompt> setenv OSAGENT_PORT 10000
```

If you are using Bourne (or BASH) shell and you want the Smart Agent to listen on port number 10000, set the `OSAGENT_PORT` environment variable as follows:

### Solaris

```
prompt> OSAGENT_PORT=10000 prompt> export OSAGENT_PORT
```

## Logging Output on the Host System

Many VisiBroker tools offer a *verbose* mode that displays information about the tool as it executes. In addition, any application that is linked with the VisiBroker library may also produce output. On UNIX systems, this output is written to the console, or the corresponding shell if invoking commands from a shell. On Windows systems, this output is written to one of several log files.

The following table summarizes the names of the various log files that may be produced on Windows Host systems when using the Windows “Host Side” tools.

**Table 1** Summary of log file names produced on Windows in verbose mode.

File Name	Description
<code>osagent.log</code>	Produced by the Smart Agent when started with the <code>-v</code> flag.
<code>visout.log</code>	Contains any output to <code>cout</code> that is produced by a client or server.
<code>vislog.log</code>	Contains any output to <code>clog</code> that is produced by a client or server.
<code>viserr.log</code>	Contains any output to <code>cerr</code> that is produced by a client or server.

The location of these log files is determined by the following rules:

- 1 An attempt will be made to write the file to the log directory within the directory pointed to by the `VBROKER_ADM` variable. Please refer to [“Setting VBROKER\\_ADM Environment Variable”](#) for details on how to set the `VBROKER_ADM` environment variable.
- 2 If step 1 fails, an attempt will then be made to write the file to the current directory.

# Developing an Example Application with VisiBroker-RT for C++

*This chapter uses an example application to describe the development process for creating distributed, object-based applications.*

*The code for this example application is provided in the <install\_location>/examples/basic/bank\_account directory where the VisiBroker-RT for C++ distribution was installed. If you do not know the location of the VisiBroker-RT for C++ distribution, see your system administrator.*

## Development Process

When you develop distributed applications with VisiBroker-RT for C++, you must first identify the objects required by the application. You will then usually follow these steps:

- 1 Write a specification for each object using the Interface Definition Language (IDL).

IDL is the language that an implementer uses to specify the operations that an object will provide and how they should be invoked. In this example, we define, in IDL, the `Account` interface with a `balance()` method and the `AccountManager` interface with an `open()` method.

- 2 Use the IDL compiler to generate the client stub code and server POA servant skeleton code.

Using the `idl2cpp` compiler, we'll produce client-side stubs (which provide the interface to the `Account` and the `AccountManager` objects' methods) and server-side classes (which provides classes for the implementation of the remote objects).

- 3 Write the client program code.

To complete the implementation of the client program, initialize the ORB, bind to the `Account` and the `AccountManager` objects, invoke the methods on these objects, and print out the balance.

- 4 Write the server object code.

To complete the implementation of the server object code, we must derive from the `POA_Account` and `POA_AccountManager` classes, provide implementations of the interfaces' methods, and implement the server's "main/start" routine.

- 5 Compile the client and server code.

To create the client program, compile and link the client program code with the client stub. To create the `Account` server, compile and link the server object code with the server skeleton.

- 6 Integrate the VisiBroker libraries needed into VxWorks

- 7 Initialize the ORB for the Server processor and start the server.

- 8 Initialize the ORB for the Client processor and run the client program.

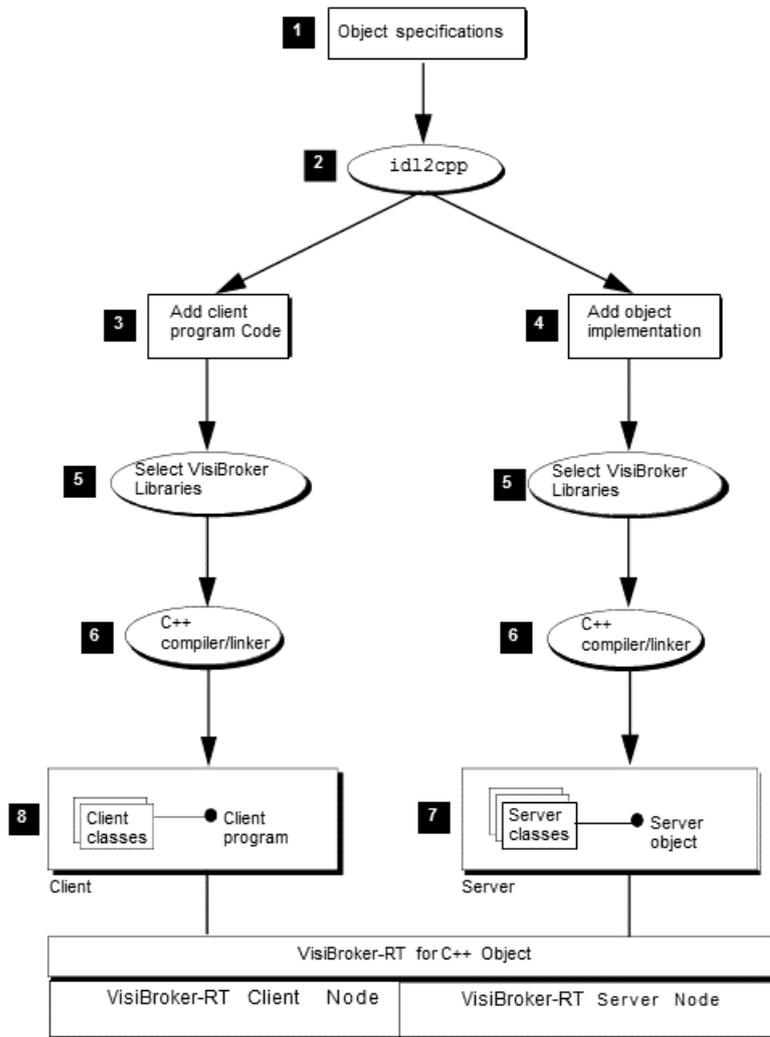


Figure 3 Developing the sample bank application

## Step 1: Defining object interfaces

The first step to creating an application with VisiBroker-RT for C++ is to specify all of your objects and their interfaces using the OMG's Interface Definition Language (IDL). The IDL can be mapped to a variety of programming languages. The IDL mapping for C++ is summarized in the VisiBroker-RT for C++ *Reference Guide*.

You then use the `idl2cpp` compiler to generate stub routines and servant skeleton code from the IDL specification. The stub routines are used by your client program to invoke operations on an object. You use the servant code, along with code you write, to create a server that implements the object. The code for the client and object, once completed, is used as input to your C++ compiler to produce a client application and an object server.

## Writing the account interface in IDL

IDL has a syntax similar to C++ and can be used to define modules, interfaces, data structures, and more.

[IDL sample 1](#) shows the contents of the `Bank.idl` file for the `bank_account` example. The `Account` interface provides a single member function for obtaining the current balance. The `AccountManager` interface creates an account for the user if one does not already exist.

**IDL sample 1** Bank.idl file provides the Account and Account Manager interface definition

```
module Bank{
    interface Account {
        float balance();
    };
    interface AccountManager {
        Account open(in string name);
    };
};
```

## Step 2: Generating client stubs and server servants

The interface specification you create in IDL is used by VisiBroker-RT for C++'s `idl2cpp` compiler to generate C++ stub routines for the client program, and skeleton code for the object implementation. The stub routines are used by the client program for all member function invocations. You use the skeleton code, along with code you write, to create the server that implements the objects.

The code for the client program and server object, once completed, is used as input to your C++ compiler and linker to produce the client and server. These steps are shown in [Figure 3](#).

Because the `bank.idl` file requires no special handling, it can be compiled with the following command.

```
prompt> idl2cpp -source_ext cpp bank.idl
```

For more information on the command-line options for the `idl2cpp` compiler, see ["Using the IDL compiler"](#).

## Files produced by the idl compiler

The `idl2cpp` compiler generates four files from the `bank.idl` file,

- `bank_c.hh`—Contains the definitions for the `Account` and `AccountManager` classes.
- `bank_c.cc`—Contains internal stub routines used by the client.
- `bank_s.hh`—Contains the definitions for the `POA_Account` and `POA_AccountManager` servant classes.
- `bank_s.cc`—Contains the internal routines used by the server.

You will use the `bank_c.hh` and `bank_c.cc` files to build the client application. The `bank_s.hh` and `bank_s.cc` files are for building the server object. All generated files have either a `.cc` or `.hh` suffix. (The suffix may be controlled by the `-source_ext` option on the `idl2cpp` command line.)

## Caution

You should never modify the contents of files generated by the `idl2cpp` compiler.

## Step 3: Implementing the client

Many of the classes used in implementing the bank client are contained in the code generated by the `idl2cpp` compiler. The file named `client.cpp`, part of the `bank_account` example, contains the implementation of the client program. Normally you would create this file.

Because your program uses the `Account` as well as the `AccountManager` IDL interfaces, it must include the `bank_c.hh` file.

In order for a client and/or server application to be able to use the ORB, the ORB object must be initialized. The file `corba_init.C` contains the ORB initialization code for both the server and client objects. The function `start_corba` can be called from the WindShell after loading the `corba_init` program. Please refer to the `bank_account.html` file for a detailed description of how to load (where applicable) and execute the `bank_account` client example on your VxWorks target.

The files `corba_init.C` and `client.C` implement the sequence of steps required to run the `start_account_client` program. These are:

- Initialize the ORB (`corba_init.C`)
- Bind to an `AccountManager` object (`client.C`)
- Obtain a `Account` object by invoking `open()` on the `AccountManager` object (`client.C`)
- Obtain the balance by invoking `balance()` on the `Account` object (`client.C`)

## corba\_init.C

The first task that your client application needs to do is initialize the ORB object, as shown in [Code example 1](#).

### Code example 1 Initializing the ORB

```
#include <vxWorks.h>
#include "corba.h"
#include <taskLib.h>
#include "vutil.h"

#define OSAGENT_PORT "14000"

/*-----*/
/* Forward Declarations.                               */
/*-----*/

extern "C" void start_corba(char * ORB_options_string);
static void do_corba(Char * ORB_options_string);

/*-----*/
/* Global Variable Declarations                         */
/*-----*/
CORBA::ORB_var orb;

/*-----*/
/* function ==> start_corba*/
/* This function will spawn a vxWork task @*/
/* priority 100, which will perform the necessary*/
/* initialization for the ORB (i.e. ORB_init,...)*/
/*-----*/
```

```

void start_corba(char * ORB_options_string)
{
    char *          taskName = "DO_CORBA";
    int             prio     = 100;
    int             option   = VX_FP_TASK;
    int             stackSize = 20000;

    /*-----*/
    /* Spawn do_corba task. */
    /*-----*/
    taskSpawn(taskName,
              prio,
              option,
              stackSize,
              (FUNCPTR)do_corba,
              (int)ORB_options_string,0,0,0,0,0,0,0,0,0);
}

/*-----*/
/* function ==>do_corba */
/* This function will perform the necessary */
/* initialization for the ORB (i.e. ORB_init,...) */
/*-----*/

void do_corba(char * ORB_options_string)
{
    /*-----*/
    /* ORB init options can be specified in two ways. */
    /* 1) By calling start_corba and specifying the */
    /*    ORB initialization string */
    /*    (e.g. start_corba("-ORBagentport 19000")) */
    /* 2) Programatically by specifying the */
    /*    ORB initialization options in the */
    /*    default_argc and default_argv variables below. */
    /*-----*/
    /* PLEASE NOTE THAT THE OPTIONS PASSED IN VIA start_corba*/
    /* OVERRIDE THE OPTIONS THAT ARE SET PROGRAMATICALLY. */
    /*-----*/

    int default_argc = 2;
    char *default_argv[] = {"-ORBagentport", OSAGENT_PORT};
    char **new_argv;
    int new_argc = VISUtil::stringToArgv(&new_argv, default_argv,
    default_argc, ORB_options_string);

    /*-----*/
    /* Call ORB_init */
    /*-----*/
    VISTRY
    {
        // Initialize the ORB
        orb = CORBA::ORB_init(new_argc, new_argv);

        VISUtil::freeArgv(new_argc, new_argv);
    }
    VISCATCH(CORBA::Exception, e)
    {
        cerr << e << endl;
        taskSuspend(0);
    }
    VISEND_CATCH

    return;
}

```

## client.C

The `start_bank_client` program implements the client application which obtains the current balance of a bank account. The client programs performs the following steps:

- 1 Bind to an `AccountManager` object (`client.C`)
- 2 Obtain a `Account` object by invoking `open()` on the `AccountManager` object (`client.C`)
- 3 Obtain the balance by invoking `balance()` on the `Account` object (`client.C`)

### Code example 2 Client side program

```
//bank_account client

#include <vxWorks.h>
#include "corba.h"
#include <vport.h>
#include "bank_c.hh"

/*-----*/
/* Forward Declarations */
/*-----*/

extern "C" void start_bank_client(const char* name);
static void bank_client(const char* name);

/*-----*/
/* Global Variable Declarations */
/*-----*/

extern CORBA::ORB_var orb;

void start_bank_client(const char* name)
{
    char *    taskName = "BANK_CLINT";
    int      Prio = 100;
    int      option = VX_FP_TASK;
    int      stackSize = 20000;

    taskSpawn(taskName,
              Prio,
              option,
              stackSize,
              (FUNCPTR)bank_client,
              (int)name,0,0,0,0,0,0,0,0,0);
}

void bank_client(const char* name)
{
    // The client uses the "_bind" method by default which locates
    // the Server Object via the OSAgent. There is also a provision
    // for the client to use the Server's stringified IOR
    // (eg. cases where using the OsAgent may not be supported). To
    // use the IOR method, copy the stringified IOR in place of the
    // NULL value below. This stringified IOR is typically displayed
    // on the server console after the server has been activated.
    char * IOR = NULL ;

    VISTRY {

        // Locate an account manager. Give the full POA name and the
        // servant ID.
        Bank::AccountManager_var manager;

        if ( IOR!=NULL ) {
            // convert the stringified IOR to an object reference

            CORBA::Object_var object = orb->string_to_object(IOR);
        }
    }
}
```

```

        VISIFNOT_EXCEP
            manager = Bank::AccountManager::_narrow(object);
        VISEND_IFNOT_EXCEP
    }
    else {
        PortableServer::ObjectId_var managerId =
PortableServer::string_to_ObjectId("BankManager");

        VISIFNOT_EXCEP
            manager = Bank::AccountManager::_bind("bank_account_poa",
(CORBA_OctetSequence &)managerId);
        VISEND_IFNOT_EXCEP
    }

    Bank::Account_var account;

    // Set the account name
    if (name==NULL) {
        name = "Jack B. Quick";
    }

    VISIFNOT_EXCEP
        account = manager->open(name);
    VISEND_IFNOT_EXCEP

    // Get the balance of the account.
    CORBA::Float balance;

    VISIFNOT_EXCEP
        balance = account->balance();
    VISEND_IFNOT_EXCEP

    // Print out the balance.
    VISIFNOT_EXCEP
        cout << "The balance in " << name << "'s account is $"
        << balance << endl;
    VISEND_IFNOT_EXCEP
    }
    VISCATCH(CORBA::Exception, e) {
        cerr << e << endl;
    }
    VISEND_CATCH

    return;
}

```

## Binding to the AccountManager object

Before your client program can invoke the `open(String name)` member function, it must first use the `_bind()` member function to establish a connection to the server that implements the `AccountManager` object. The implementation of the `_bind()` member function is generated automatically by the `idl2cpp` compiler. The `_bind()` member function requests the ORB to locate and establish a connection to the CORBA server object. If the server object is successfully located and a connection is established, a proxy object is created to represent the server's `POA_AccountManager` object. A pointer to this proxy `AccountManager` object is returned to your client program.

## Obtaining an Account object

Next your client program needs to call the `open()` member function on the `AccountManager` object to get a pointer to the `Account` object for the specified customer name.

## Obtaining the balance

Once your client program has established a connection with an `Account` object, the `balance()` member function can be used to obtain the balance. The `balance()` member function on the client side is actually a stub generated by the `idl2cpp` compiler that gathers all the data required for the request and sends it to the server object.

## Other member functions

Several other member functions are provided that allow your client program to manipulate an `AccountManager` object reference. Many of these are not used in the example client application, but they are described in detail in the VisiBroker-RT for C++ *Reference Guide*.

## Step 4: Implementing the server

Just as with the client, many of the classes used in implementing the bank server are contained in the header files generated by the `idl2cpp` compiler. The `server.C` file is a server implementation included for the purposes of illustrating this example. Normally you, the programmer, would create this file.

Note, just as with the client, the server program requires the ORB to have already been initialized. The file `corba_init.C` contains the ORB initialization code for the server objects. Please refer to the *bank\_account.html* file for a detailed description of how to load and execute the `bank_account` example on your VxWorks target.

## server.C

This file implements the `Server` class for the server side of our `bank_account` example. The server program does the following:

- 1 Initialize the ORB (`corba_init.C`)
- 2 Creates a Portable Object Adapter with the required policies. (`server.C`)
- 3 Creates the account manager servant object. (`server.C`)
- 4 Activates the servant object. (`server.C`)
- 5 Activates the POA manager (and the POA). (`server.C`)

### Code example 3 Initializing the ORB

```
//bank_account server
#include <vxWorks.h>
#include "corba.h"
#include "bankImpl.h"

/*-----*/
/* Forward Declarations. */
/*-----*/

extern "C" void start_bank_server(void);
static void bank_server(void);

extern CORBA::ORB_var orb;

// Declare global objects
AccountRegistry AccountManagerImpl::_accounts;

void start_bank_server(void)
{
    char * taskName = "BANK_SRVR";
```

```

int          Prio = 100;
int          option = VX_FP_TASK;
int          stackSize = 20000;

taskSpawn(taskName,
          Prio,
          option,
          stackSize,
          (FUNCPTR)bank_server,
          0,0,0,0,0,0,0,0,0,0);
}

void bank_server()
{
    PortableServer::POA_var rootPOA;

    VISTRY {

        //get a reference to the root POA
        CORBA::Object_var obj =
        orb->resolve_initial_references("RootPOA");

        VISIFNOT_EXCEP
            rootPOA = PortableServer::POA::_narrow(obj);
        VISEND_IFNOT_EXCEP

        CORBA::PolicyList policies;
        policies.length(1);

        VISIFNOT_EXCEP
            policies[(CORBA::ULong)0] =
            rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);
        VISEND_IFNOT_EXCEP

        // get the POA Manager
        PortableServer::POAManager_var poa_manager;

        VISIFNOT_EXCEP
            poa_manager = rootPOA->the_POAManager();
        VISEND_IFNOT_EXCEP

        // Create myPOA with the right policies
        PortableServer::POA_var myPOA;

        VISIFNOT_EXCEP
            myPOA = rootPOA->create_POA("bank_account_poa",
            poa_manager, policies);
        VISEND_IFNOT_EXCEP

        // Create the servant
        AccountManagerImpl *managerServant = new AccountManagerImpl;

        // Create the object ID
        PortableServer::ObjectId_var managerId;

        VISIFNOT_EXCEP
            managerId =
            PortableServer::string_to_ObjectId("BankManager");
        VISEND_IFNOT_EXCEP

        // Activate the servant with the ID on myPOA
        VISIFNOT_EXCEP
            myPOA->activate_object_with_id((CORBA_OctetSequence
            &)managerId,managerServant);
        VISEND_IFNOT_EXCEP

        // Activate the POA Manager
        VISIFNOT_EXCEP
            poa_manager->activate();
        VISEND_IFNOT_EXCEP

        CORBA::Object_var ref;
    }
}

```

```

VISIFNOT_EXCEP
    ref = myPOA->servant_to_reference(managerServant);
VISEND_IFNOT_EXCEP

CORBA::String_var string_ref;

VISIFNOT_EXCEP
    string_ref = orb->object_to_string(ref.in());
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
    cout << endl << "CORBA Object ==> " << endl << endl;
    cout << ref << endl;
    cout << string_ref << endl << endl;
    cout << " is ready" << endl << endl;
VISEND_IFNOT_EXCEP

}
VISCATCH(CORBA::Exception, e) {
    cerr << e << endl;
    taskSuspend(0);
}
VISEND_CATCH
return;
}

```

## Understanding the Account class hierarchy

The `Account` class that you implement is derived from the `POA_Bank::Account` class that was generated by the `idl2cpp` compiler. Look closely at the `POA_Bank::Account` class definition that is defined in the `bank_s.hh` file. Figure 4 shows the class hierarchy.

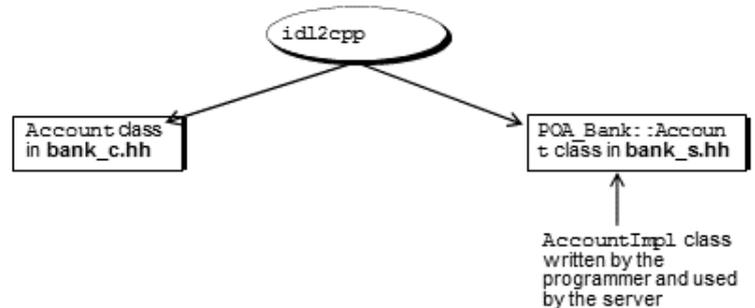


Figure 4 Class hierarchy for the AccountImpl interface

## Step 5: Building the example

There are basically three types of VxWorks programs/modules which are produced with each example:

- ORB Initializer (***corba\_init***)
 

The server skeleton (i.e. `bank_s.o`) and the client stub (i.e. `bank_c.o`) are compiled and linked in as part of this program, to support the use of the Tornado linking loader via the "ld" command from the WindShell.
- Server implementation (***i.e server***)
 

Created from the `server.C` file.
- Client program (***i.e. client***)

Created from the client.C file.

The corba\_init, server and client programs/modules are all dependant on the VisiBroker-RT for C++ ORB libraries (i.e. liborb\_dyn.o or liborb\_min.o and the libagentsupport.o or libagentsupport\_min.o, depending on whether you intend to use the Osagent location service). Please refer to [“Step 6: Integrating VisiBroker-RT with Tornado/VxWorks”](#) for more information on the delivered VisiBroker libraries.

Each example directory contains a **html** file detailing, in addition to a description of the example, the procedure for building that specific example. The top level of the examples directory (i.e. <install\_location/VisiBrokerRT60/examples) also contains a **README.html** which contains links to all the individual example html files.

## Step 6: Integrating VisiBroker-RT with Tornado/VxWorks

### The VisiBroker-RT Runtime

The VisiBroker-RT for C++ run-time is composed of several libraries. Each library supports a particular feature set of VisiBroker-RT. Which VisiBroker-RT feature the VxWorks application requires will dictate which VisiBroker-RT library is needed on the VxWorks target system.

VisiBroker-RT libraries are delivered in the following formats:

***The old manual says “delivered in four formats” - and then lists three! So I’ve changed the wording***

- Relocatable object modules (e.g. liborb\_min.o)  
This format is provided to support linking the VisiBroker-RT library with the VxWorks kernel to make a bootable VxWorks image when building a VxWorks image from the command line. (e.g. "make vxWorks", from the VxWorks Board Support Package directory)
- "munched" relocatable object modules (e.g. liborb\_min\_munched.o)  
VisiBroker-RT provides "munched" libraries as "ease-of-use" libraries to allow dynamic loading when using the Tornado WindShell or CrossWind debugger. (e.g. from the Tornado WindShell ->"ld < liborb\_min\_munched.o")
- VisiBroker-RT Tornado 2.2 Components  
This format is provided to support building a VisiBroker-RT enabled "bootable VxWorks image (custom configured)".

## VisiBroker-RT runtime libraries

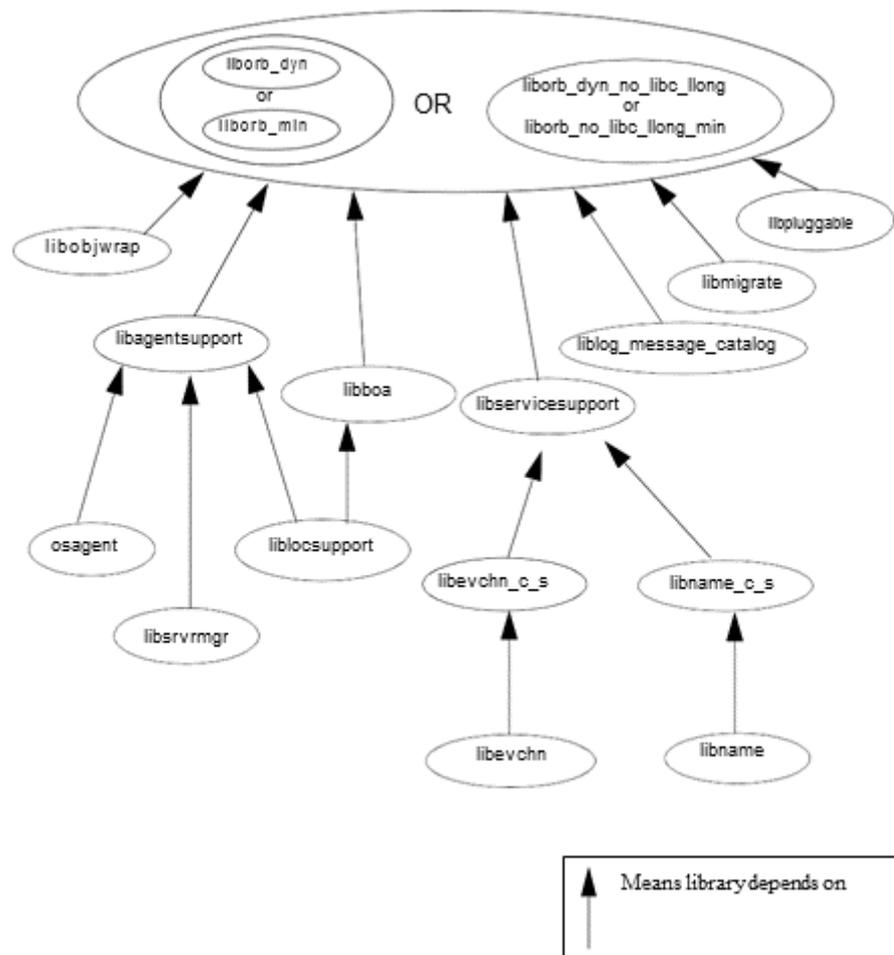
The following table describes the VisiBroker-RT runtime libraries and the features provided by each:

Library	Description
Relocatable Object Module: liborb_dyn.o  "munched" Relocatable Object Module: liborb_dyn_munched.o  Tornado 2.2 Component Name: "Dynamic ORB"	Dynamic CORBA version of the VisiBroker Object Request Broker library; includes all Dynamic functionality (i.e. DSI,DII,IR)  <b>Note:</b> the VisiBroker Object Activation Daemon is not supported in VisiBroker-RT for C++.
Relocatable Object Module: liborb_dyn_no_libc_llong.o  "munched" Relocatable Object Module: liborb_dyn_no_libc_llong_munched.o  Tornado 2.2 Component Name: Not Applicable	This library provides the same functionality as liborb_dyn.o library, with the exception that it DOES NOT INCLUDE the GCC libc long long arithmetic operators.  The long long arithmetic operators are not provided by the VxWorks libraries (e.g. libPPC604gnuvx.a), but are included for the default ORB libraries (liborb_dyn, liborb_min), since full support for the CORBA:Longlong is dependent on them.  Since other VxWorks products also include these long long arithmetic operators as well, these "no_libc_llong" libraries are delivered to support co-existence with these other products (e.g. VxWorks Personal JWorks).
Relocatable Object Module: liborb_min.o  "munched" Relocatable Object Module: liborb_min_munched.o  Tornado 2.2 Component Name: "Minimal ORB"	"Minimum CORBA" version of the VisiBroker Object Request Broker library  This version of the VisiBroker-RT for C++ ORB has been "scaled down" by removing functionality which typically would not be required for embedded applications. The following components are not part of the liborb_min.o library: <ul style="list-style-type: none"> <li>• Dynamic Invocation Interface (DII)**</li> <li>• Dynamic Skeleton Interface (DSI)**</li> <li>• Interface Repository**</li> <li>• Dynamic Any Types**</li> </ul> <p>Note, however that this library still provides support for the CORBA: :Any type.</p> <p>**This functionality has been removed based on the OMG Minimum CORBA specification.</p>

Library	Description
Relocatable Object Module: liborb_no_libc_llong_min.o  "munched" Relocatable Object Module: liborb_no_libc_llong_min_munched.o  Tornado 2.2 Component Name: Not Applicable	This library provides the same functionality as liborb_min.o library, with the exception that it DOES NOT INCLUDE the GCC lib long long arithmetic operators.  The long long arithmetic operators are not provided by the VxWorks libraries (e.g. libPPC604gnuvx.a), but are included for the default ORB libraries (liborb_dyn, liborb_min), since full support for the CORBA:Longlong is dependent on them.  Since other VxWorks products also include these long long arithmetic operators as well, these "no_libc_llong" libraries are delivered to support co-existence with these other products (e.g. VxWorks Personal JWorks).
Relocatable Object Module: libagentsupport.o  "munched" Relocatable Object Module: libagentsupport_munched.o  Tornado 2.2 Component Name: "Dynamic ORB"	Provides the functionality required for the ORB to communicate with the Osagent. This library is required if your application requires the services of the VisiBroker SmartAgent (Osagent).
Relocatable Object Module: libboa.o  "munched" Relocatable Object Module: libboa_munched.o  Tornado 2.2 Component Name: "Basic Object Adaptor (BOA) Support"	This library provides support for the Basic Object Adapter (BOA). Use of the library is required if your application requires the CORBA 2.1 BOA interface.
Relocatable Object Module: libevchn_c_s.o  "munched" Relocatable Object Module: libevchn_c_s_munched.o  Tornado 2.2 Component Name: "Event Service Client Support"	This library provides the interfaces to allow applications to be clients of the Visi- Broker-RT for C++ Event Service. If one of your VxWorks nodes intends to start a Event Service channel and/or factory it must include both this library as well as the library libevchn.o (described below)
Relocatable Object Module: libevchn.o  "munched" Relocatable Object Module: libevchn_munched.o  Tornado 2.2 Component Name: "Event Service"	This library provides the interfaces for creating and starting VisiBroker-RT for C++ Event Service channels and/or factories on a VxWorks node
Relocatable Object Module: liblocsupport.o  "munched" Relocatable Object Module: liblocsupport_munched.o  Tornado 2.2 Component Name: "Location Service"	This library provides support for the VisiBroker Location Service. Use of the library is required if your application requires use of the Location Service interface.  See <a href="#">"Using the Location Service"</a> for details on the VisiBroker Location Service.

Library	Description
Relocatable Object Module: liblog_message_catalog.o  "munched" Relocatable Object Module: liblog_message_catalog_munched.o  Tornado 2.2 Component Name: "Log Message Catalog"	This library provides support for the formatted output of ORB log messages. Use of the library is required if your application desires more verbose logging. By default VisiBroker logging only includes message keys not message text.  Please refer to <a href="#">"VisiBroker Logging"</a> for details on the VisiBroker Location Service.
Relocatable Object Module: libmigrate.o  "munched" Relocatable Object Module: libmigrate_munched.o  Tornado 2.2 Component Name: "Migrate Interceptors"	This library provides support for the 3.x style of VisiBroker Interceptors. Use of the library is required if you are migrating a 3.x application which use Interceptors and want to keep the 3.x style Interceptor API.  Please refer to <a href="#">"Migrating VisiBroker Code"</a> for details on migrating 3.x style interceptor applications.
Relocatable Object Module: libname_c_s.o  "munched" Relocatable Object Module: libname_c_s_munched.o  Tornado 2.2 Component Name: "Naming Service Client Support"	This library provides the interfaces for client applications which intend to ONLY use the VisiBroker-RT for C++ Naming Service. If one of your VxWorks target nodes intends to start a Naming Service "root context" it must include both this library as well as the library libname.o (described below).
Relocatable Object Module: libname.o  "munched" Relocatable Object Module: libname_munched.o  Tornado 2.2 Component Name: "Naming Service"	This library provides the interfaces for creating and starting a VisiBroker-RT for C++ Naming Service on a VxWorks node.
Relocatable Object Module: libobjwrap.o  "munched" Relocatable Object Module: libobjwrap_munched.o  Tornado 2.2 Component Name: "Object Wrappers"	This library provides support for VisiBroker Object Wrappers. Use of the library is required if your application requires use of Object Wrappers.  Please refer to <a href="#">"Using Object Wrappers"</a> for details on the Object Wrappers type of Interceptors.
Relocatable Object Module: ibpluggable.o  "munched" Relocatable Object Module: libpluggable_munched.o  Tornado 2.2 Component Name: "Pluggable Transport Interface"	This library provides support for the VisiBroker Pluggable Transport interfaces. Use of the library is required if your application requires use of a user provided transport other than TCP/IP.
Relocatable Object Module: libsvmgr.o  "munched" Relocatable Object Module: libsvmgr_munched.o  Tornado 2.2 Component Name: "Server Manager"	This library provide provides support for communicating with the VisiBroker Console.

Library	Description
Relocatable Object Module: libservicesupport.o	This library provides support for the VisiBroker Common Object Services. Use of the library is required if your application requires use of the Naming or Event Service.
"munched" Relocatable Object Module: libservicesupport_munched.o	
Tornado 2.2 Component Name: "Support for Common Object Services"	
Relocatable Object Module: osagent.o	The VisiBroker SmartAgent. This library is required to run the VisiBroker Smart Agent on a VxWorks node.
"munched" Relocatable Object Module: osagent_munched.o	
Tornado 2.2 Component Name: "Smart Agent"	



**Figure 5** Interdependencies between the VisiBroker-RT libraries

## VisiBroker-RT's use of VxWorks TCB extensions

The VisiBroker-RT for C++ ORB makes use of the VxWorks Task Control Block (TCB) spare fields. In particular the `spare1` field of the **WIND\_TCB** is used by the `liborb`.

### Note

This field must remain the exclusive use of the VisiBroker-RT for C++ `liborb`.

```
typedef struct windTcb      /* WIND_TCB - task control block */
{
    Q_NODE    qNode;        /* 0x00: multiway q node: rdy/pend q */
    Q_NODE    tickNode;     /* 0x10: multiway q node: tick q */
    Q_NODE    activeNode;   /* 0x20: multiway q node: active q */
    .
    .
    .
    spare1    <---Used by the liborb
}
```

## Integrating VisiBroker Libraries with Tornado 2.x

There are a couple of alternative configuration methods that the Tornado 2.x environment provides that can be used to integrate the VisiBroker-RT runtime libraries with VxWorks:

- 1 The first method involves *modifying the VxWorks configuration files* (i.e. `configAll.h`, `config.h`,...), as well as the VxWorks `Makefile`. This is the more traditional VxWorks configuration method.
- 2 The second method makes use of the *Tornado 2.x Project Facility*. Both of these configuration methods are described below.

### Integrating with a VxWorks BSP Makefile

- 1 Configure the VisiBroker required VxWorks Components using the VxWorks Configuration files (i.e. `config.h`, `configAll.h`)

The VxWorks components required by VisiBroker can be configured into a bootable VxWorks image by either:

- a modifying the file `$WIND_BASE/target/config/all/configAll.h` and adding the appropriate "#define" line(s) to the "INCLUDED SOFTWARE FACILITIES" section; or
  - b modifying the file `$WIND_BASE/target/config/<your BSP directory>/config.h` and adding the appropriate "#define" line(s) to that file.
- 2 Link the VisiBroker Libraries into a VxWorks bootable image by modifying the VxWorks Makefile.

The VisiBroker required VxWorks kernel components are (**Note** that they **MUST** be configured into the VxWorks kernel to ensure correct operation of the VisiBroker-RT for C++ for Tornado product):

- a `#define INCLUDE_TASK_HOOKS` (part of VxWorks default configuration)
- b To make use of VisiBroker-RT for C++ for Tornado "environment variables" for setting options ENVIRONMENT VARIABLE support must be included:

```
#define INCLUDE_ENV_VARS (part of VxWorks default configuration)
```

c C++ and IO STREAMS support must be included into the VxWorks image.

1 C++ support

```
#define INCLUDE_CPLUS (part of VxWorks default configuration)
```

2 FULL IOSTREAMS support

```
#define INCLUDE_CPLUS_IOSTREAMS_FULL
```

d Additionally the VisiBroker-RT runtime assumes that the VxWorks Networking component has been initialized prior to the `ORB_init` and `create_POA` calls.

3 Link the VisiBroker Libraries into a VxWorks bootable image by modifying the VxWorks Makefile.

The recommended way of using the VisiBroker-RT for C++ 2.x libraries is to build them into the VxWorks kernel.

The relocatable VisiBroker-RT object modules are installed under the path `<install location>/VisiBrokerRT60/lib/<Tornado_Version>/<your_target_cpu>`.

In order for an application to be able to make use of the VisiBroker-RT for C++ runtime, a version of the `liborb` (i.e. `liborb_dyn.o`, `liborb_min.o`,...) library must be linked in with the bootable VxWorks system image.

Linking the VisiBroker-RT runtime into a VxWorks system image can be accomplished by modifying the "Makefile" under the appropriate VxWorks Board Support Package (BSP) directory.

The example below assumes the following configuration:

- VxWorks node is a "ADS860" target board
- the Minimum CORBA version of the ORB library (i.e. `liborb_min.o`) with support for Osagent communications (i.e. `libagentsupport.o`) is required.
- and an `osagent` (`osagent.o`) is required to run on the node.

Change directory to the **`$WIND_BASE/target/config/ads860`** directory, and modify the file **`Makefile`** as follows. Replace the following line:

```
MACH_EXTRA =
```

with the line

```
MACH_EXTRA =  
<install location>/VisiBrokerRT60/lib/T2_2/PPC860/liborb_min.o  
\<br><install location>/VisiBrokerRT60/lib/T2_2/PPC860/  
libagentsupport.o  
<install location>/VisiBrokerRT60/lib/T2_2/PPC860/osagent.o
```

## Integrating VisiBroker Libraries with Tornado 2.2

There are a couple of alternative configuration methods that the Tornado 2.2 product provides that can be used to integrate the VisiBroker-RT runtime libraries with VxWorks:

- 1 The first method involves modifying the *VxWorks configuration files* (i.e. `configAll.h`, `config.h`,...), as well as the VxWorks Makefile. This is the more traditional VxWorks configuration method. This method is described in ["Integrating with a VxWorks BSP Makefile"](#).

- 2 The second method makes use of the *Tornado 2.2 Project Facility* and the VisiBroker-RT Tornado 2.2 Components. This method is described below.

## Integrating using the Tornado 2.2 Project facility

### Configure the VisiBroker-RT required VxWorks Components using the Tornado 2.2 Project facility

In addition to the VxWorks default Tornado 2.2 included components, VisiBroker requires:

- Full IO STREAMS support (component name "**full C++ iostreams**")
- Additionally the VisiBroker ORB assumes that the VxWorks Networking component has been initialized prior to the `ORB_init` call.

The component "**initialize network at boot time**" is usually included as part of the default VxWorks configuration, however this is NOT the case for the FULL VxSim simulator. This component must be included if VxWorks is to initialize the Networking subsystem for your application. Otherwise your application MUST perform this initialization prior to initializing the VisiBroker ORB.

#### Note

The VxWorks kernel components identified above MUST be configured into the VxWorks system image to ensure proper operation of the VisiBroker-RT runtime.

The VxWorks bootable system image must be configured to include these components. This can be accomplished by updating the *VxWorks view of your Workspace* to select each of these components for inclusion into the VxWorks system image.

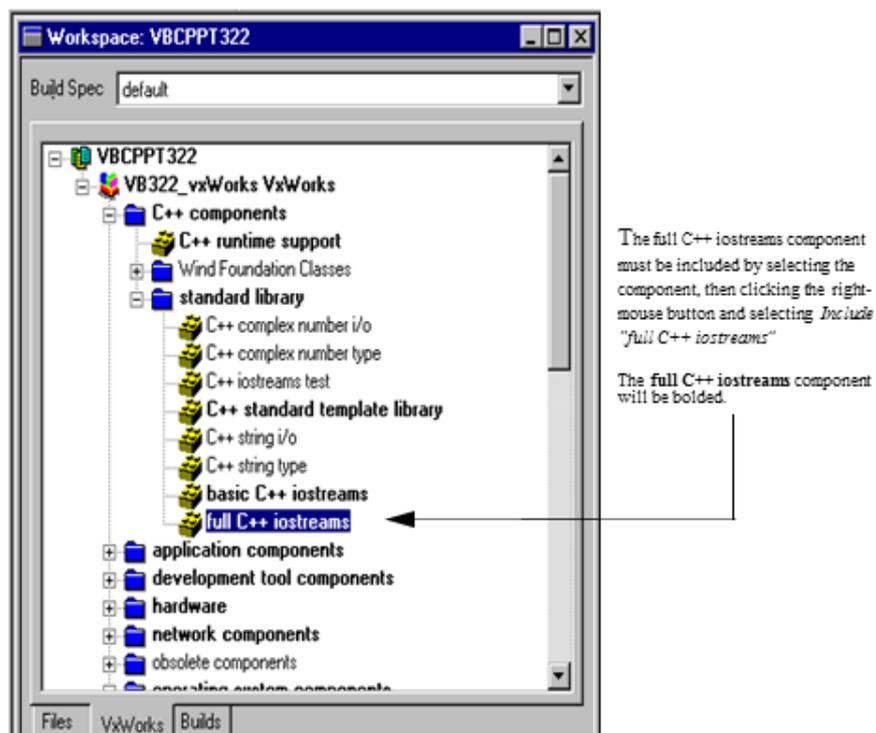
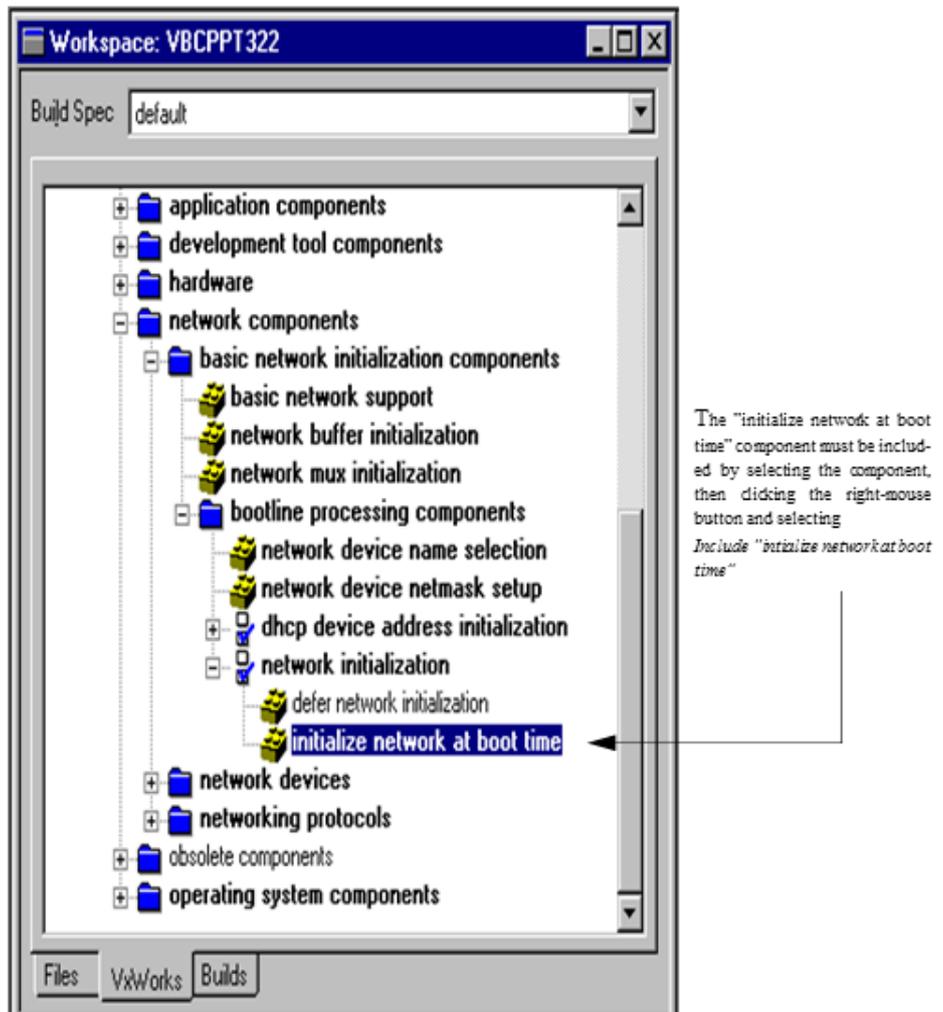


Figure 6 Including full iostreams component using the Project Workspace

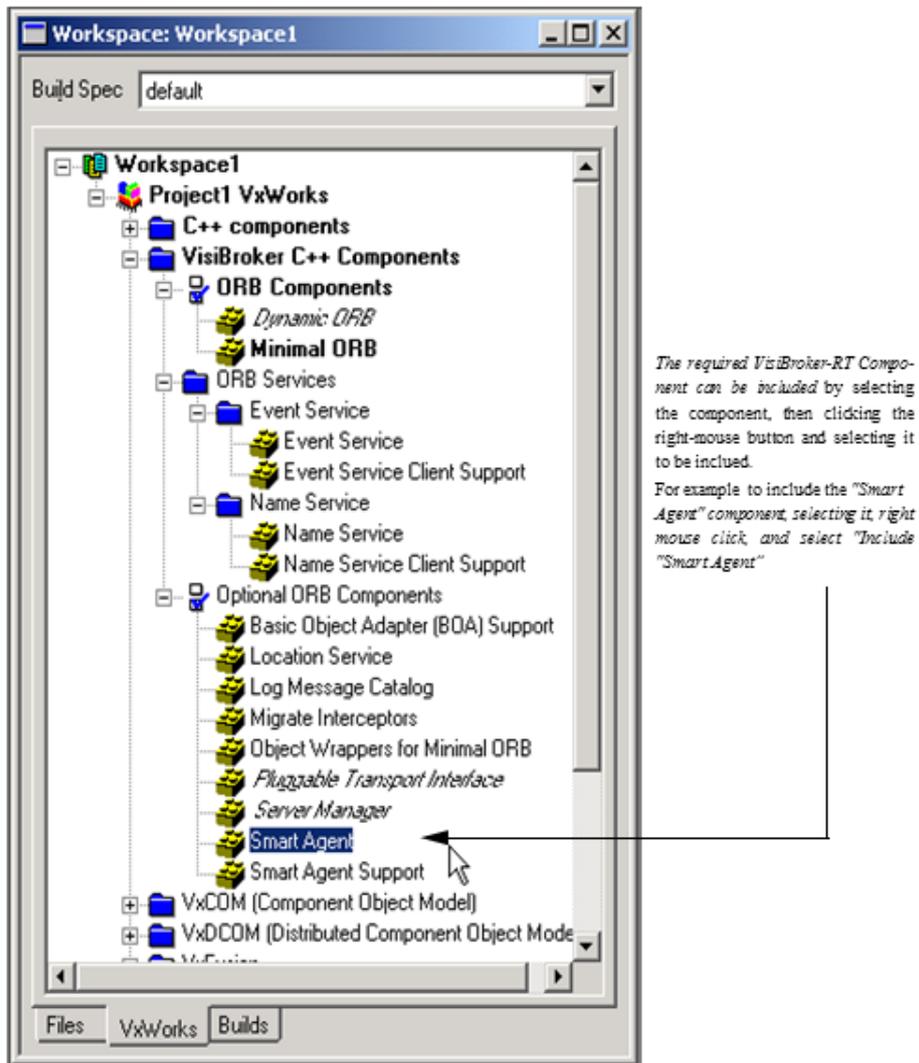


**Figure 7** Including "initialize network at boot time" component using the Project Workspace

### Build the VisiBroker-RT enabled bootable VxWorks system image

The recommended way of using any of the VisiBroker-RT runtime is to build them as part of the VxWorks system image. This is accomplished by selecting the VisiBroker-RT components to be included in the VxWorks system image.

The VisiBroker-RT runtime components are integrated with the Tornado 2.2 Project facility during the VisiBroker-RT for C++ installation. Using the Tornado 2.2 Project facility, VisiBroker-RT components can be included into a "bootable VxWorks image (custom configured)" by selecting the desired VisiBroker component, much like a VxWorks 5.5 component is selected.



**Figure 8** Including VisiBroker-RT components into a bootable VxWorks 5.5 image

### Failure to download

If your new VxWorks image fails to download (during a Network download using a VxWorks ROM), it is possible that the "new" VxWorks image (which now contains the ORB) may be overwriting the execution of your VxWorks BOOTROM during the actual download phase.

An obvious symptom is the following:

```
VxWorks Bootrom prints
>Loading ##### "
```

(where ### is the size of your image). The system then "hangs" during the download.

To correct this problem, you have two options:

- 1 Rebuild VxWorks bootroms increasing the value of "RAM\_HIGH\_ADRS".** This must be done in the Board Support Package directory in BOTH of the files **config.h** and **Makefile**.

For example to change the RAM high address from 1 megabyte to 2 megabytes you would make the following modification. (Please refer to the "VxWorks Programmers Guide - Version 5.3.1 section 8.4 Alternative VxWorks Configurations", for more details on the proper setting of this variable.)

change the line:

```
#define RAM_HIGH_ADRS 0x100000
```

to:

```
#define RAM_HIGH_ADRS 0x200000
```

- 2 Use the WindShell to download the munched version of the liborb (liborb\_min\_munched.o).

Loading the liborb\_munched version of the liborb dynamically from the shell, requires the VxWorks image to have sufficient memory for the VxWorks TOOLS. The portion of the VxWorks target systems heap which is preallocated for the VxWorks HOST Tools (i.e. WindShell, CrossWind,...) is controlled by the:

```
#define WDB_POOL_SIZE.
```

This definition is located in the \$WIND\_BASE/target/config/all/configAll.h file. The default value of WDB\_POOL\_SIZE is 1/16 of the VxWorks target system heap.

```
configAll.h WDB_POOL_SIZE default setting:
```

```
#define WDB_POOL_SIZE((sysMemTop() - FREE_RAM_ADRS)/16)
```

For example if your target system is a 68040 based VME board (mv162) with 8 megabytes(MB) of RAM, and you are downloading the liborb\_munched version of the liborb using the WindShell, your target system's WDB\_POOL\_SIZE should be set to a value large enough to support download of the liborb\_munched.o (943124 bytes) plus your CORBA application(s). So assuming you need ~1 M of additional host tools memory for your application prototyping, your WDB\_POOL\_SIZE needs to equa @ least 2M. On an 8 megabyte mv162 the setting of the WDB\_POOL\_SIZE would look something like:

```
configAll.h WDB_POOL_SIZE default setting:
```

```
#define WDB_POOL_SIZE ((sysMemTop() - FREE_RAM_ADRS)/4)
```

## Note

Option number 1 (rebuilding bootroms) is the recommended use.

## Using VisiBroker with VxSim

VxSim, the VxWorks simulator, is available for Solaris as well as Windows NT/2000. All delivered VisiBroker libraries support the VxWorks simulator; however the VisiBroker-RT runtime is *only* available for VxSim as relocatable object modules. This means that when working on either VxSim for Solaris or VxSim for Windows NT/2000/XP, a VxWorks bootable system image MUST include the VisiBroker-RT runtime libraries required for the application. Please refer to "[Integrating VisiBroker Libraries with Tornado 2.x](#)" for details on how to include the VisiBroker libraries as part of a bootable VxWorks system image.

The configuration requirements for VxSim are different depending on whether the Windows or the Solaris version of the VxSim simulator is being used. In order to take advantage of the distributed aspects of VisiBroker, the FULL version of the simulator is required, since VxWorks Networking is included only in the FULL Simulator.

## Using VisiBroker with VxSim for Solaris

To use VisiBroker-RT for C++ with the Tornado Solaris VxSim distribution, configuration of VxSim is required in the following areas.

### Network Interface configuration

When using the VisiBroker-RT for C++ VxSim target distribution, it is recommended that VxSim be configured to use a PPP (Point-to-Point Protocol) network interface, as the ULIP network interface is being phased out by Wind River Systems.

As part of configuring VxSim to use the PPP network interface Wind River Systems delivers an "asppd.cf" file (in the `WIND_BASE/target/config/solaris` directory) which is used when configuring the PPP interface on your Solaris system. The value in this file for the default IP address configuring the PPP interface is typically "127.0.1.254". *This value must be changed to ensure proper VisiBroker operation.* The modified value should be a non-existent subnet other than "127.xxx.xxx.xxx". In addition in this same file the IP address value of each of the VxSim targets must also be modified to reflect the PPP IP address which was used for configuring the PPP interface on the Solaris host. Below is a sample excerpt of the original asppd.cf file as delivered by WindRiver Systems, followed by a modified asppd.cf file. Note, the network-specific portion of the "modified" IP address is different. This is very important.

Original asppd.cf file as delivered by WindRiver Systems:

```
ifconfig ipd0 plumb 127.0.1.254 up private netmask 0xffffffff00
path
  interface ipd0
  peer_system_name vxsim0
  peer_ip_address 127.0.1.0
path
  interface ipd0
  peer_system_name vxsim1
  peer_ip_address 127.0.1.1
...
```

Modified asppd.cf file for correct operation of VisiBroker-RT for C++:

```
ifconfig ipd0 plumb 200.200.200.254 up private netmask 0xffffffff00
path
  interface ipd0
  peer_system_name vxsim0
  peer_ip_address 200.200.200.0      (for Tornado 2)
  peer_ip_address 200.200.200.1    (for Tornado AE)
path
  interface ipd0 peer_system_name vxsim1
  peer_ip_address 200.200.200.0    (for Tornado 2)
  peer_ip_address 200.200.200.2    (for Tornado AE)
...
```

### VxSim Boot parameters for Tornado 2

The VxSim boot parameters must also be modified to match the VxSim target addresses which are specified in the "modified" asppd.cf file described above. When the VxSim process is started it will attempt to read a file named "VxWorks.nvram#" where # corresponds to the VxSim processor number from the "Launch VxSim" dialog. This file must be modified/ created to contain the SAME IP address which was used in the "modified" asppd.cf file for that VxSim target.

For example:

If you are starting VxSim target #0. Then in the "Launch VxSim" dialog you will specify `Processor number => 0`.

Upon starting, this VxSim target will attempt to read the file "VxWorks.nvram0" to get its boot parameters. The IP address specified in this file MUST match the IP address for vxsim0 specified in the "asppp.cf" file. Here is a sample *VxWorks.nvram0* for this scenario:

```
VxWorks.nram0 file
ppp(0,0)MYHOSTNAME:/MYWINDBASE/vxSim/target/config/solaris/
VxWorks e=200.200.200.0:ffffff00 h=192.192.192.60
g=200.200.200.254 u=vxuser tn=vxsim0
```

## VxSim Host Table configuration (only applicable for Tornado 2.x Solaris Simulator)

The VxSim target does not automatically make a host-table entry in its host table for its own host name (e.g. "vxsim0" above). This is a difference between VxSim and other VxWorks targets, where the target name is automatically added to the VxWorks host table at boot time. The VisiBroker ORB requires that the "local host name" of the target have an entry in the VxWorks host table.

Adding an entry into the VxWorks host table can be accomplished by making a call to the hostLib function "hostAdd" from the VxWorks start-up function `usrRoot` (in file `usrConfig.C`). The modification to the `usrConfig.c` file would look something like this:

```
hostAdd("vxSim0", "200.200.200.0"); ...where "vxSim0" is the host
name of the VxSim target.
```

## Tornado 2.x VxSim process Memory Size

When starting up VxSim the "Launch VxSim" dialog window appears. There are two boxes in this dialog window, *Processor number* and *Memory size (bytes)*. The *Memory size* should be modified to 16000000 (16 megabytes) to ensure proper operation of VxWorks with VisiBroker-RT for C++.

## Using VisiBroker with VxSim for Windows NT/2000/XP

To use VisiBroker-RT for C++ with the Windows NT/2000 VxSim distribution, configuration of VxSim is required in the following areas.

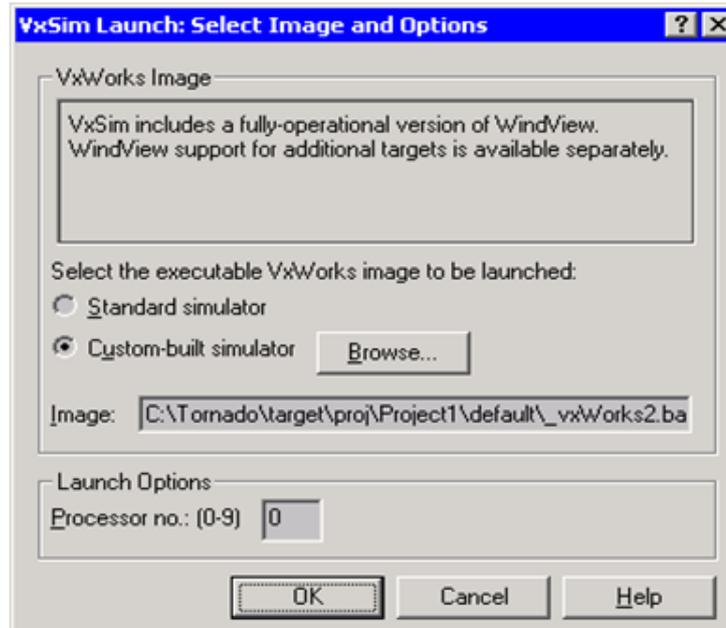
### Tornado 2.x VxSim process Memory Size

To ensure proper operation of VxSimNT with VisiBroker-RT for C++ the recommended memory size for the VxSim process is 16 megabytes (16000000). This can be configured as follows:

- 1 Create a dos batch file which contains the following (where `-p` specifies VxWorks processor number and `-r` specifies VxSim process memory size in bytes):

```
start vxWorks.exe -p1 -r16000000
```

- 2 When starting up VxSim the "VxSim Launch: Select Image and Options" dialog window appears. Select the "Custom Built Simulator" button, and specify the "Image" to be the dos batch file created above.



**Figure 9** The "VxSim Launch: Select Image and Options" dialog window

### Building a VisiBroker-ready VxSim NT/2000 Bootable VxWorks image

(only applicable for Tornado 2.x VxSimNT Simulator)

Attempting to link the multiple VisiBroker libraries (i.e. liborb\_dyn.o, libagentsupport.o) into a VxSim NT bootable image might fail with the following error message:

**Code example 4** VxSim NT Error when building a VisiBroker ready VxWorks Bootable image

```
ccsimpc -BD:\Tornado\host\x86-win32\lib\gcc-lib\ -U WINNT -UWIN32
-U WINNT -UWINNT -U MINGW32 -U_WIN32 -U WIN32 -U WIN32
-U WIN32 -nostdlib -r -Wl,-X \
-Wl,@D:\Tornado\target\proj\testproj\prjObjs.lstversion.o D:\
Tornado\target\lib\libSIMNTgnuvx.a D:\Tornado\VisiBrokerRT60\lib\
T2_2\SIMNT\liborb.o D:\Tornado\VisiBrokerRT60\lib\T2_2\SIMNT\
libagentsupport.o -o partialImage.o
\\D:\Tornado\host\x86-win32\i386-pc-mingw32\bin\ld.exe:
partialImage.o:
reloc overflow: 0x122fc > 0xffff
partialImage.o: final close failed: File truncated
make: *** [partialImage.o] Error 0x1
```

This is a known problem with the Tornado VxSim NT tools. In particular the VxSim NT linker.:

"Your problem has been identified as the SPR#30726: Problem with VxSim loader - can't handle multiple text, data sections.

The explanation/workaround below should be able to resolve your build problem which is caused by the limitation of the OMF i.e. PE-COFF. The COFF cannot represent more than 65535 relocations in a single section. Using the linker option, split-by-reloc and the attached linker script should be able to resolve the problem."

Please contact Wind River Technical support and obtain Patch SPR#30726. This can be easily obtained by going to <http://www.windriver.com/>

[corporate/html/tsmain.html](#) and logging into WindSurf (you will need a login ID and password).

**The above windriver link is Page Not Found. Suggestions?**

## Step 7: Starting the Smart Agent (osagent) Service

The Smart Agent provides VisiBroker's object location functions and must be started on at least one node on the local network. The Smart Agent (Osagent) is required to be initialized prior to any server objects attempting to register, and prior to any client applications attempting to bind to any server objects. The Smart Agent is described in detail in ["Using the Smart Agent"](#).

The VisiBroker SmartAgent is required if you are using the `_bind` operation in your client application to locate and connect to server implementations. For initial development and familiarity with the VisiBroker product use of the Smart Agent is recommended. However if your application will eventually use some alternative Location Service (e.g. VisiBroker Interoperable Naming Service, custom location service,...) the Smart Agent will not be required.

When use of the SmartAgent is not required, the library `libagentsupport` is not required resulting in a smaller footprint for the required VisiBroker ORB libraries. Please refer to ["Step 6: Integrating VisiBroker-RT with Tornado/VxWorks"](#) for a description of these libraries and their dependencies.

There are 2 categories of osagent executables which are delivered with the VisiBroker-RT for C++ product release, a **Development Host** osagent and a **VxWorks node** osagent. In order to be able to "start" the VxWorks node osagent, it MUST have been made available on the VxWorks node.

### Configuring the Osagent to work with VxSim

Configuration of Osagent to ORB communications is required on both the development host as well as the VxWorks VxSim virtual target.

### Configuring the VisiBroker ORB running on VxSim to support osagent communications

The default mechanism for establishing communications between the VisiBroker ORB and the OSAGENT as well as between OSAGENTS, uses the IP subnet broadcast mechanism. Since both the Windows Ulip driver and the Solaris PPP driver are non-broadcast protocol interfaces, this method of communications is not possible. Therefore use of the environment variable `OSAGENT_ADDR` or the `ORB_init` parameter `"-ORBagentAddr"` is required. See the section "ORB options" in the *VisiBroker-RT for C++ Programmers' Reference* for details on the use of the `-ORBagentAddr` parameter.

### Configuring the Smart Agent running on vxsim targets

The primary mechanism for initial OSAGENT to ORB and OSAGENT to OSAGENT communications is based on UDP broadcast. Since both the Windows Ulip driver and the Solaris PPP driver are non-broadcast protocol interfaces, this method of communications is not possible. So use of the `OSAGENT_LOCAL_FILE` (on the host) or the `OSAGENT_LOCAL_TABLE` (on the target) is required. See the section "Specifying interface usage for Smart Agents" in the *VisiBroker-RT for C++ Programmers' Reference*, for a description of how to use the `OSAGENT_LOCAL_TABLE`. Additionally "Working with multihomed hosts" in the *VisiBroker-RT for C++*

*Programmers' Reference* discusses how to use the OSAGENT\_LOCAL\_TABLE on target systems.

## Starting the Osagent on a Windows Development Host

The VisiBroker Smart Agent can be started from a Windows DOS prompt in console mode as follows:

```
prompt> osagent -C
```

## Starting the Osagent on a Solaris Development Host

The VisiBroker Smart Agent can be started from any Unix shell as follows:

```
prompt> osagent &
```

## Starting the Osagent on a VxWorks Node

The Osagent task is initialized and started via a call to the following function:

```
startOsagent(  
    unsigned long  priority, Osagent task priority (200 is default)  
    int           verbose = 0,  
    int           port=-1,      (default is 14000)  
    short         logger_priority=-1  (VisiBroker Logger Task  
priority),  
    OSAGENT_LOCAL_ENTRY*local_table = NULL,  (pointer to  
OSAGENT_LOCAL_TABLE)  
    OSAGENT_ADDR_ENTRY *addr_table=NULL)  (pointer to  
OSAGENT_ADDR_TABLE)  
    long         initial_heartbeat_window = 60,  (Osagent to ORB  
Heartbeat interval)  
    long         initial_heartbeat_frequency = 5, (Osagent to ORB  
initial Heartbeat frequency)  
    long         heartbeat_frequency = 300);  (Osagent to ORB  
Heartbeat frequency)
```

The header file `vosagent.h` must be included in the file which is calling this function. This header file provides the function prototype for `startOsagent`, as well as a description on the use of the `OSAGENT_LOCAL_TABLE` and the `OSAGENT_ADDR_TABLE`.

Please refer to the file `corba_init.C` in any of the example subdirectories which are delivered as part of the VisiBroker-RT for C++ product distribution. These example subdirectories can be found in the `<install_location>/VisiBrokerRT60/examples` directory.

### Note

To turn on the VERBOSE option for the osagent, set "Parameter #2 of startOsagent" above to a value of 1. Likewise if you need the osagent to run at a different port number than the default (14000) set "Parameter #3 of startOsagent" above to the desired port number value.

Please refer to "[Integrating with a VxWorks BSP Makefile](#)" for details on how to configure the osagent as part of your Tornado bootable VxWorks image.

The VisiBroker Smart Agent can be started from a Tornado WindShell as follows:

```
--> startOsagent()
```

## Step 8: Starting the server and running the example

You are now ready to run your first VisiBroker-RT for C++ application. Make sure that you have:

- 1 Compiled your client program and server implementation

- 2 Created a VxWorks bootable image containing the required VisiBroker libraries,
- 3 Started a VisiBroker Smart Agent (Osagent) on your local network.

In the scenario we describe below, the server will be running on VxWorks **node#1** and the client application will be running on VxWorks **node#2**.

Additionally, the steps below assume you are using the VxWorks WindShell to dynamically load the sample VisiBroker applications.

## Starting the server

From the Tornado WindShell:

- 1 Load the programs on VxWorks **node#1**.

From a Tornado 2 WindShell:

```
-> ld < corba_init
-> ld < server
```

Initialize the ORB on VxWorks **node#1**

```
-> start_corba
```

`start_corba` should be run only ONCE. This will initialize the ORB.

The program `corba_init` also has the server skeleton (`bank_s.cc`) and the client stub (`bank_c.cc`) linked in. This has been done in order to support loading the server and/or client program multiple times.

If you require multiple loads of the server skeleton or client stub, you will need to reboot your target. However as long as the IDL interface does not change (i.e. the `bank_s(_c).cc` files do not change, which is usually the case) the server implementation and the client stub can be loaded and unloaded multiple times. Without any adverse effects on the VisiBroker ORB libraries.

- 2 Start the bank server on VxWorks **node#1**

```
-> start_bank_server
```

You should see output similar to:

```
CORBA Object ==>
Repository ID: IDL:Bank/AccountManager:1.0
Object name: NONE
IOR:002020200000001c49444c3a42616e6b2f4163636f756e744d616e6167
65723a312e30000000000100000000000004c000102200000000e3230302e
3230302e3230302e300004010000002b00504d430000004000000102f6261
6e6b5f6167656e745f706f6100000000b42616e6b4d616e61676572200000
0000
is ready
```

- 3 Now you can run the `osfind` command from your Unix/Windows development host to see what interfaces and objects are currently available on the your network. You should see output similar to:

```
osfind: Found one agent at port 14000
      HOST: <hostname where osagent is running>
osfind: There are no OADs running on in your domain.
osfind: There are no Object Implementations registered with OADs.
osfind: Following are the list of Implementations started manually.
      HOST: <name of VxWorks target>
      REPOSITORY ID: IDL:Bank::Account:1.0
      OBJECT NAME: NONE
```

## NOTE

An alternative to using the **osfind** utility is the **VisiBroker Console**. The VisiBroker Console gives you a graphical interface into the VisiBroker Smart Agent database. Additionally the Console provides a view into the ORB instances running and the active objects on each as well as the configuration of each ORB instance. For details on using the VisiBroker Console see [“Using the VisiBroker-RT for C++ Console”](#).

## Running the client

### Windows

From the Tornado WindShell:

- 1 Load the programs on VxWorks **node#2**.

From a Tornado 2 WindShell:

```
-> ld < corba_init
-> ld < client
```

Initialize the ORB on VxWorks **node#2**

```
-> start_corba
```

`start_corba` should be run only ONCE. This will initialize the ORB.

The program `corba_init` also has the server skeleton (`bank_s.cc`) and the client stub (`bank_c.cc`) linked in. This has been done in order to support loading the server and/or client program multiple times.

If you require multiple loads of the server skeleton or client stub, you will need to reboot your target. However as long as the IDL interface does not change (i.e. the `bank_s(_c).cc` files do not change, which is usually the case) the server implementation and the client stub can be loaded and unloaded multiple times. Without any adverse effects on the VisiBroker ORB libraries.

- 2 Run the bank client program.

```
-> start_bank_client "john"
```

At this point you should see the following output on both VxWorks target #1 and VxWorks target #2's output console window:

Client	Server
	Created john's account.
	Returning john's account:
	Repository ID:
	IDL:Bank/Account:1.0
	Object name: NONE
The balance in john's account is \$243.06	

---

# Handling Exceptions

## Exceptions in the CORBA model

The exceptions in the CORBA model include both system and user exceptions. The CORBA specification defines a set of *system* exceptions that can be raised when errors occur in the processing of a client request. Also, system exceptions are raised in the case of communication failures. System exceptions can be raised at any time and they do not need to be declared in the interface. You can define *user* exceptions in IDL for objects you create and specify the circumstances under which those exceptions are to be raised. They are included in the method signature. If an object raises an exception while handling a client request, the ORB is responsible for reflecting this information back to the client.

## System exceptions

System exceptions are usually raised by the ORB, though it is possible for object implementations to raise them through interceptors discussed in the chapter “Using Portable Interceptors”. When the ORB raises a `SystemException`, it will be one of the CORBA-defined error conditions shown in the following table.

**Table 2** CORBA-defined system exceptions

Exception name	Description
BAD_CONTEXT	Error processing context object.
BAD_INV_ORDER	Routine invocations out of order.
BAD_OPERATION	Invalid operation.
BAD_PARAM	An invalid parameter was passed.
BAD_TYPECODE	Invalid typecode.
COMM_FAILURE	Communication failure.
DATA_CONVERSION	Data conversion error.
FREE_MEM	Unable to free memory.
IMP_LIMIT	Implementation limit violated.
INITIALIZE	ORB initialization failure.
INTERNAL	ORB internal error.
INTF_REPOS	Error accessing interface repository.
INV_FLAG	Invalid flag was specified.
INV_INDENT	Invalid identifier syntax.
INV_OBJREF	Invalid object reference specified.
MARSHAL	Error marshalling parameter or result.
INVALID_TRANSACTION	Specified transaction was invalid (used in conjunction with ITS/OTS).
NO_IMPLEMENT	Operation implementation not available.
NO_MEMORY	Dynamic memory allocation failure.
NO_PERMISSION	No permission for attempted operation
NO_RESOURCES	Insufficient resources to process request.
NO_RESPONSE	Response to request not yet available.
OBJ_ADAPTOR	Failure detected by object adaptor.
OBJECT_NOT_EXIST	Object is not available.
PERSIST_STORE	Persistent storage failure.

Exception name	Description
TRANSIENT	Transient failure.
TRANSACTION_REQUIRED	Transaction is required (used in conjunction with ITS/OTS).
TRANSACTION_ROLLEDBACK	Transaction was rolled back (used in conjunction with ITS/OTS).
TIMEOUT	Request timeout.
UNKNOWN	Unknown exception.

#### Code example 5 SystemException class

```
class SystemException : public CORBA::Exception {
public:
    static const char* _id;
    virtual ~SystemException();
    CORBA::ULong minor() const;
    void minor(CORBA::ULong val);
    CORBA::CompletionStatus completed() const;
    void completed(CORBA::CompletionStatus status);
    ...
    static SystemException *_downcast(Exception *);
    ...
};
```

## Obtaining completion status

System exceptions have a completion status that tells you whether or not the operation that raised the exception was completed. The `CompletionStatus` enumerated values are shown below. `COMPLETED_MAYBE` is returned when the status of the operation cannot be determined.

#### IDL sample 2 CompletionStatus values

```
enum CompletionStatus {
    COMPLETED_YES = 0;
    COMPLETED_NO = 1;
    COMPLETED_MAYBE = 2;
};
```

You can retrieve the completion status using these `SystemException` methods.

#### Code example 6 Retrieving completion status

```
CompletionStatus completed();
```

## Getting and setting the minor code

You can retrieve and set the `minor` code using these `SystemException` methods. Minor codes are used to provide better information about the type of error.

#### Code example 7 Retrieving and setting minor codes

```
ULong minor() const;
void minor(ULong val);
```

## Determining the type of a SystemException

The design of the VisiBroker-RT for C++ exception classes allows your program to catch any type of exception and then determine its type by using the `_downcast()` method. A static method, `_downcast()` accepts a pointer to any `Exception` object. As with the `_downcast()` method defined

on `CORBA::Object`, if the pointer is of type `SystemException`, `_downcast()` will return the pointer to you. If the pointer is not of type `SystemException`, `_downcast()` will return a `NULL` pointer. See Appendix A, "CORBA exceptions," for details.

## Catching system exceptions

Your applications should enclose the ORB and remote calls in a try catch block. [Code example 8](#) illustrates how the account client program, discussed in "Developing an Example Application with VisiBroker-RT for C++", prints an exception.

### Code example 8 Printing an exception

```
#include "Bank_c.hh"
...

void start_client(const char* name)
{
    // The client uses the "_bind" method by default which locates
    // the Server Object via the OSAgent. There is a provision
    // for the client to use the Server's stringified IOR ( cases
    // where using the OsAgent may not be supported). To use the
    // IOR method, copy the stringified IOR in place of the NULL
    // value below.This stringified IOR is typically displayed on
    // the server console after the server has been activated.
    char * IOR = NULL ;

    VISTRY {

        // Locate an account manager. Give the full POA name and
        // the servant ID.
        Bank::AccountManager_var manager;

        if ( IOR!=NULL ) {
            // convert the stringified IOR to an object reference
            CORBA::Object_var object = orb->string_to_object(IOR);

            VISIFNOT_EXCEP
            manager = Bank::AccountManager::_narrow(object);
            VISEND_IFNOT_EXCEP
        }
        else {
            PortableServer::ObjectId_var managerId =
            PortableServer::string_to_ObjectId("BankManager");

            VISIFNOT_EXCEP
            manager = Bank::AccountManager::_bind("/bank_account_poa",
            managerId);
            VISEND_IFNOT_EXCEP
        }

        Bank::Account_var account;

        ....

    }
    VISCATCH(CORBA::Exception, e) {
        cerr << e << endl;
    }
    VISEND_CATCH

    return 0;
}
```

If you were to execute the client program with these modifications and without a server present, the following output would indicate that the operation did not complete and the reason for the exception.

```

-> start_bank_client
Exception: CORBA::OBJECT_NOT_EXIST
Minor: 0
Completion Status: NO

```

## Downcasting exceptions to a system exception

You can modify the bank\_account client program to attempt to downcast any exception that is caught to a `SystemException`. [Code example 9](#) shows how you might modify the client program. [Code example 10](#) shows how the output would appear if a system exception occurred.

### Code example 9 Downcasting an exception to a system exception

```

void bank_client(const char* name)

VISTRY {

    // Bind to an account.
    Account_var account = Account::_bind();

    // Get the balance of the account.
    CORBA::Float acct_balance;
    VISIFNOT_EXCEP
    acct_balance = account->balance(); VISEND_IFNOT_EXCEP

    // Print out the balance
    VISIFNOT_EXCEP.
    cout << "The balance in the account is $"
    << acct_balance << endl VISEND_IFNOT_EXCEP;
}
VISCATCH(CORBA::Exception, e) {
    CORBA::SystemException var sys_excep;
    sys_excep = CORBA::SystemException::_downcast(&e);
    if(sys_excep != NULL) {
        cerr << "System Exception occurred:" << endl;
        cerr << "exception name: " <<
        sys_excep->_name() << endl;
        cerr << "minor code: " << sys_excep->minor() << endl;
        cerr << "completion code: " << sys_excep->completed() <<
        endl;

    }else {
        cerr << "Not a system exception" << endl;
        cerr << e << endl;
    }
}
VISEND_CATCH
}

```

### Code example 10 Output from the system exception

```

System Exception occurred:
exception name: CORBA::NO_IMPLEMENT
minor code: 0
completion code: 1

```

## Catching specific types of system exceptions

Rather than catching all types of exceptions, you may choose to specifically catch each type of exception that you expect. [Code example 11](#) shows this technique.

**Code example 11** Catching specific types of exceptions

```
...
void bank_client(const char* name)

    VISTRY {

        // Bind to an account.
        Account_var account = Account::_bind();

        // Get the balance of the account.
        CORBA::Float acct_balance;
        VISIFNOT_EXCEP
            acct_balance = account->balance();
        VISEND_IFNOT_EXCEP

        // Print out the balance
        VISIFNOT_EXCEP.
            cout << "The balance in the account is $"
                << acct_balance << endl;
        VISEND_IFNOT_EXCEP;
    }
    VISCATCH(CORBA::SystemException, sys_excep){
        // Check for system errors
        cout << "System Exception occurred:" << endl;
        cout << "exception name: " << sys_excep->_name() << endl;
        cout << "minor code: " << sys_excep->minor() << endl;
        cout << "completion code: " << sys_excep->completed()
            << endl;
    }
    VISEND_CATCH
}
...
```

## User exceptions

When you define your object's interface in IDL you can specify the user exceptions that the object may raise. [Code example 12](#) shows the `UserException` code from which the `idl2cpp` compiler will derive the user exceptions you specify for your object.

**Code example 12** UserException class

```
class UserException:
    public Exception { public:
    ...
        static const char* id;
        virtual ~UserException();

        static UserException *_downcast(Exception *);
};
```

## Defining user exceptions

Suppose that you want to enhance the `bank_account` application, introduced in "[Developing an Example Application with VisiBroker-RT for C++](#)", so that the `account` object will raise an exception. If the `account` object has insufficient funds, you want a user exception named `AccountFrozen` to be raised. The additions required to add the user exception to the IDL specification for the `Account` interface are shown in bold.

### IDL sample 3 Defining user exceptions

```
// Bank.idl
module Bank {
    interface Account {
        exception AccountFrozen {
        };
        float balance() raises(AccountFrozen);
    };
};
```

The `idl2cpp` compiler will generate the following code for a `AccountFrozen` exception class.

### Code example 13 AccountFrozen class generated by the idl compiler

```
class Account : public virtual CORBA::Object {
    ...
    class AccountFrozen: public CORBA_UserException {
    public:
        static const CORBA_Exception::Description description;

        AccountFrozen() {}
        static CORBA::Exception *_factory() {
            return new AccountFrozen();
        }
        ~AccountFrozen() {}
        virtual const CORBA_Exception::Description& _desc() const;
        static AccountFrozen *_downcast(CORBA::Exception *exc);
        CORBA::Exception *_deep_copy() const {
            return new AccountFrozen(*this);
        }

        void _raise() const { VISTHROW_INST(this) }
    }
    ...
};
```

## Modifying the object to raise the exception

The `AccountImpl` object must be modified to use the exception by raising the exception under the appropriate error conditions.

### Code example 14 Modifying the object implementation to raise an exception

```
CORBA::Float AccountImpl::balance()
{
    if( _balance < 50 ) {
        VISTHROW(Account::AccountFrozen());
        VISRETURN(return 0.0;);
    } else {
        return _balance;
    }
}
```

## Catching user exceptions

When an object implementation raises an exception, the ORB is responsible for reflecting the exception to your client program. Checking for a `UserException` is similar to checking for a `SystemException`. To modify the account client program to catch the `AccountFrozen` exception, make modifications like those shown in [Code example 15](#).

### Code example 15 Catching a UserException

```
...
VISTRY {

    // Bind to an account.
    Account_var account = Account::_bind();

    // Get the balance of the account.
    CORBA::Float acct_balance;
```

```

VISIFNOT_EXCEP
    acct_balance = account->balance();
VISEND_IFNOT_EXCEP
}
VISCATCH (Account::AccountFrozen, e){
    cerr << "AccountFrozen returned:" << endl;
    cerr << e << endl;
    return(0);
}
// Check for system errors
VISAND_CATCH(CORBA::SystemException, sys_except){
}
VISEND_CATCH
...

```

## Adding fields to user exceptions

You can associate values with user exceptions. [Code example 16](#) shows how to modify the IDL interface specification to add a reason code to the `AccountFrozen` user exception. The object implementation that raises the exception is responsible for setting the reason code. The reason code is printed automatically when the exception is put on the output stream.

**Code example 16** Adding a reason code to the `AccountFrozen` exception

```

// Bank.idl
module Bank {
    interface Account {
        exception AccountFrozen {
            int reason;
        };
        float balance() raises(AccountFrozen);
    };
};

```

# The VisiBroker C++ Exception Support

The CORBA specification defines an `Environment` class for reflecting exceptions. VisiBroker uses the `Environment` class, along with a set of macros, to provide your applications with efficient C++ exception handling capabilities.

## The Exception Macros

The `Environment` class is used internally by the ORB and is transparent to you as a programmer. The only requirement is that you use these exception macros to throw, try and catch exceptions. These macros will transparently manipulate the `Environment` class and provide efficient, reentrant safe exception handling.

**Table 3** The VIS exception macros.

Macro name	Purpose
VISTRY	Use this as you would use the try statement.
VISTHROW(type_name)	Throws the specified exception.
VISTHROW_LAST	Used to re-throw the specified exception. Used only in an event handler or in a method called by an event handler.
VISCATCH(type_name, variable_name)	Use this to catch an exception of the specified type.

Macro name	Purpose
VISAND_CATCH	If several exceptions are to be specified for a VISTRY block, use VISCATCH for the first catch statement and VISAND_CATCH for all subsequent catch statements.
VISEND_CATCH	Used to terminate a VISCATCH block.
VISCATCH_ALL	Used to catch any exception which is thrown. As opposed to VISCATCH which catches the specified exception
VISAND_CATCHALL	If several exceptions are to be specified for a VISTRY block, use VISCATCH for the first catch statement and VISAND_CATCHALL to catch all other types of exceptions which are thrown.
VISTHROW_INST	Used to throw an exception from an object instance's "throw" method (e.g. instance->_throw)
VISIF_EXCEP	Used to check if an exception was thrown and perform a specified action which follows
VISCLEAR_EXCEP	Clears the current environments, exception information,
VISIFNOT_EXCEP	Used to check if an exception was NOT thrown and continue with the application processing
VISEND_IFNOT_EXCEP	Used to terminate a VISIFNOT_EXCEP block
VISRETURN(what)	Used to return after a VISTHROW, for example VISRETURN( <i>return?</i> )

# Part 2

## Server Concepts

### **In this part**

This part contains the following chapters:

<a href="#">Server basics</a>	<a href="#">page 51</a>
<a href="#">Using POAs</a>	<a href="#">page 61</a>
<a href="#">Using the Tie Mechanism</a>	<a href="#">page 81</a>



# Server basics

*This chapter outlines the tasks that are necessary to set up a server to receive client requests.*

## Overview

The basic steps that need to be performed in setting up your server are:

- Initialize the ORB
- Select policies and Create the POA
- Activate the POA Manager
- Activate objects
- Wait for client requests

This chapter describes each task in a global manner to give you an idea of what you must consider. The specifics of each step is dependent on your individual requirements.

## Initializing the ORB

As stated in the previous chapter, the ORB provides a communication link between client requests and object implementations. Each application must initialize the ORB before communicating with it.

**Code example 17** Initializing the ORB

```
// Initialize the ORB.  
CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
```

## Creating the POA

Early versions of the CORBA object adapter (the *Basic Object Adapter*, or *BOA*) didn't permit portable object server code. A new specification was developed by the OMG to address these issues and the *Portable Object Adapter* (or *POA*) was created.

### Note

A discussion of the POA can be quite extensive. This section introduces you to some of the basic features of the POA. For detailed information, see ["Using POAs"](#) and the OMG specification.

In basic terms, the POA (and its components) determine which *servant* should be invoked when a client request is received, and then invokes that servant. A servant is a programming object that provides the implementation of an *abstract object*. A servant is not a CORBA object.

One POA (called the *root POA*) is supplied by each ORB. You can create additional POAs and configure them with different behaviors. You can also define the characteristics of the objects the POA controls.

The steps to setting up a POA with a servant include:

- Obtaining a reference to the root POA
- Defining the POA policies
- Creating a POA as a child of the root POA

- Creating a servant and activating it
- Activating the POA through its manager

Some of these steps may be different for your application.

## Obtaining a reference to the root POA

All server applications must obtain a reference to the root POA to manage objects or to create new POAs.

### Code example 18 Obtaining a reference to the root POA

```
// get a reference to the root POA
CORBA::Object_var obj =
    orb->resolve_initial_references("RootPOA");
// narrow the object reference to a POA reference
PortableServer::POA_var rootPOA =
    PortableServer::POA::_narrow(obj);
```

You can obtain a reference to the root POA by using `resolve_initial_references`. `resolve_initial_references` returns a value of type `CORBA::Object`. You are responsible for narrowing the returned object reference to the desired type, which is `PortableServer::POA` in the above example.

You can then use this reference to create other POAs, if needed.

## Creating the child POA

The root POA has a predefined set of *policies* that cannot be changed. A policy is an object that controls the behavior of a POA and the objects the POA manages. If a different behavior, such as different lifespan policy is desired, creation of a new POA is needed.

POAs are created as children of existing POAs using `create_POA`. As many POAs as required can be created.

### Note

Child POAs do not inherit the policies of their parent POAs.

In the following example, a child POA is created from the root POA and has a persistent lifespan policy. The POA Manager for the root POA is used to control the state of this child POA. More information on POA Managers are described later in this chapter.

### Code example 19 Creating the policies and the child POA

```
CORBA::PolicyList policies;
policies.length(1);
policies[(CORBA::ULong)0] =
    rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);
// Create myPOA with the right policies
PortableServer::POAManager_var rootManager =
    rootPOA->the_POAManager();
PortableServer::POA_var myPOA = rootPOA->create_POA(
    "bank_account_poa", rootManager, policies );
```

## Implementing servant methods

IDL has a syntax similar to C++ and can be used to define modules, interfaces, data structures, and more. When you compile an IDL that contains an interface, a class is generated which serves as the base class for your servant. For example, in the **bank.idl** file, an `AccountManager` interface is described.

**Code example 20** Interfaces described in bank.idl

```

module Bank{
    interface Account {
        float balance();
    };
    interface AccountManager {
        Account open (in string name);
    };
};

```

The AccountManager implementation on the server side is shown below.

**Code example 21** AccountManagerImpl code

```

//
// We inherit from PortableServer::RefCountServantBase so that
// the servant object will be automatically deleted when the
// object is deactivated
// The _remove_ref method is called as part object
// deactivation by the POA
//
class AccountManagerImpl : public POA_Bank::AccountManager,
    public virtual
    PortableServer::RefCountServantBase
{
public:
    AccountManagerImpl() {}

    Bank::Account_ptr open(const char* name) {
        // Lookup the account in the account dictionary.
        PortableServer::ServantBase_var servant =
            _accounts.get(name);

        if (servant == PortableServer::ServantBase::_nil()) {
            // Seed the random number generator
            srand((unsigned)time(&random_time));

            // Make up the account's balance, between 0 and 1000
            // dollars.
            CORBA::Float balance = abs(rand()) % 100000 / 100.0;

            // Create the account implementation, given the balance.
            servant = new AccountImpl(balance);

            // Print out the new account
            cout << "Created " << name << "'s account." << endl;

            // Save the account in the account dictionary.
            _accounts.put(name, servant);
        }
    }
    VISTRY {
        // Activate it on the default POA which is root POA for this
        // servant
        PortableServer::POA_var default_poa = _default_POA();

        CORBA::Object_var ref;

        VISIFNOT_EXCEP
            ref = default_poa->servant_to_reference(servant);
        VISEND_IFNOT_EXCEP

        Bank::Account_var account;

        VISIFNOT_EXCEP
            account = Bank::Account::_narrow(ref);
        VISEND_IFNOT_EXCEP

        VISIFNOT_EXCEP
            // Print out the new account
            cout << "Returning " << name << "'s account: " << account
            << endl;

        // Return the account

```

```

        return Bank::Account::_duplicate(account);
    VISEND_IFNOT_EXCEP
    }
    VISCATCH(CORBA::Exception, e) {
        cerr << "_narrow caught exception: " << e << endl;
        return;
    }
    VISEND_CATCH

    return Bank::Account::_nil();
}

private:
    static AccountRegistry _accounts;
};

```

The AccountManager implementation must be created and activated in the server code. In this example, AccountManager is activated with `activate_object_with_id`, which passes the object ID to the *Active Object Map* where it is recorded. The Active Object Map is simply a table that maps IDs to servants. This approach ensures that this object is always available when the POA is active and is called *explicit object activation*.

#### Code example 22 Creating and activating the servant

```

// Create the servant
AccountManagerImpl *managerServant = new AccountManagerImpl;
// Create the object ID PortableServer::ObjectId_var managerId;

VISIFNOT_EXCEP
    managerId = PortableServer::string_to_ObjectId("BankManager");
VISEND_IFNOT_EXCEP

// Activate the servant with the ID on myPOA
VISIFNOT_EXCEP
    myPOA->activate_object_with_id(managerId, managerServant);
VISEND_IFNOT_EXCEP

```

## Activating the POA

The last step is to activate the POA Manager associated with your POA. By default, POA Managers are created in a *holding* state. In this state, all requests are routed to a holding queue and are not processed. To allow requests to be dispatched, the *POA Manager* associated with the POA must be changed from the holding state to an active state. A POA Manager is simply an object that controls the state of the POA (whether requests are queued, processed, or discarded.) A POA Manager is associated with a POA during POA creation. If a POA Manager is not specified the system will create a new one (enter `NULL` as the POA Manager name in `create_POA()`).

#### Code example 23 Activating the POA manager

```

// Activate the POA Manager
PortableServer::POAManager_var mgr=rootPoa ->the_POAManager();
VISIFNOT_EXCEP
    mgr->activate();
VISEND_IFNOT_EXCEP

```

## Activating objects

In the preceding section, there was a brief mention of explicit object activation. There are several ways in which objects can be activated:

- Explicit—all objects are activated upon server start-up via calls to the POA
- On-demand—the servant manager activates an object when it receives a request for a servant not yet associated with an object ID
- Implicit—objects are implicitly activated by the server in response to an operation by the POA, not by any client request
- Default servant—the POA uses the default servant to process the client request

A complete discussion of object activation is in the chapter “Using POAs”. For now, just be aware that there are several means for activating objects.

## Complete example

The following shows the complete code described in this chapter. You can find this code in the example “bank\_account”, which is part of the installation of VisiBrokerRT60.

**Code example 24** Complete Servant Implementation for Server side code (bankImpl.h)

```
//bankImpl.h

#include <vxWorks.h>
#include <math.h>
#include <time.h>
#include <vport.h>
#include <tickLib.h>
#include "bank_s.hh"

#define _MAX_SIZE256
#define _TYPE_SIZE 32

// The AccountRegistry is a holder of Bank account
// implementations
class AccountRegistry
{
public:
    AccountRegistry() : _count(0), _max(16), _data((Data*)NULL)
    {
        _data = new Data[16];
    }

    ~AccountRegistry() { delete[] _data; }

    void put(const char* name, PortableServer::ServantBase_ptr
servant) {
        VISMutex_var lock(_lock);
        if (_count + 1 == _max) {
            Data* oldData = _data;
            _max += 16;
            _data = new Data[_max];
            for (CORBA::ULong i = 0; i < _count; i++)
                _data[i] = oldData[i];
            delete[] oldData;
        }

        _data[_count].name = name;
        servant->_add_ref();
        _data[_count].account = servant;
        _count++;
    }
}
```

```

    PortableServer::ServantBase_ptr get(const char* name) {
        VISMutex_var lock(_lock);
        for (CORBA::ULong i = 0; i < _count; i++) {
            if (strcmp(name, _data[i].name) == 0) {
                _data[i].account->_add_ref();
                return _data[i].account;
            }
        }
        return PortableServer::ServantBase::_nil();
    }

private:
    struct Data {
        CORBA::String_var      name;
        PortableServer::ServantBase_var  account;
    };

    CORBA::ULong _count;
    CORBA::ULong _max;
    Data*_data;
    VISMutex_lock;      // Lock for synchronization
};

//
// We inherit from PortableServer::RefCountServantBase so that
// the servant object will be automatically deleted when the
// object is deactivated
// The _remove_ref method is called as part object
// deactivation by the POA
//
class AccountImpl : public virtual POA_Bank::Account,
    public virtual PortableServer::RefCountServantBase
{
public:
    AccountImpl(CORBA::Float balance) : _balance(balance) {}
    CORBA::Float balance() { return _balance; }

private:
    CORBA::Float _balance;
};

//
// We inherit from PortableServer::RefCountServantBase so that
// the servant object will be automatically deleted when the
// object is deactivated
// The _remove_ref method is called as part object
// deactivation by the POA
//
class AccountManagerImpl : public POA_Bank::AccountManager,
    public virtual
    PortableServer::RefCountServantBase
{
public:
    AccountManagerImpl() {}

    Bank::Account_ptr open(const char* name) {
        // Lookup the account in the account dictionary.
        PortableServer::ServantBase_var servant =
            _accounts.get(name);

        if (servant == PortableServer::ServantBase::_nil()) {
            // Seed the random number generator
            srand((unsigned)tickGet());

            // Make up the account's balance, between 0 and 1000
            // dollars.

            CORBA::Float balance = abs(rand()) % 100000 / 100.0;

            // Create the account implementation, given the balance.
            servant = new AccountImpl(balance);
        }
    }
};

```

```

        // Print out the new account
        cout << "Created " << name << "'s account." << endl;

        // Save the account in the account dictionary.
        _accounts.put(name, servant);
    }
    VISTRY {
        // Activate it on the default POA which is root POA for
        // this servant
        PortableServer::POA_var default_poa = _default_POA();

        CORBA::Object_var ref;

        VISIFNOT_EXCEP
            ref = default_poa->servant_to_reference(servant);
        VISEND_IFNOT_EXCEP

        Bank::Account_var account;

        VISIFNOT_EXCEP
            account = Bank::Account::_narrow(ref);
        VISEND_IFNOT_EXCEP

        VISIFNOT_EXCEP
            // Print out the new account
            cout << "Returning " << name << "'s account: " <<
account << endl;

            // Return the account
            return Bank::Account::_duplicate(account);
        VISEND_IFNOT_EXCEP
    }
    VISCATCH(CORBA::Exception, e) {
        cerr << "_narrow caught exception: " << e << endl;
        taskSuspend(0);
    }
    VISEND_CATCH

    return Bank::Account::_nil();
}

private:
    static AccountRegistry _accounts;
};

```

**Code example 25** Server Implementation for Server side code (server.C)

```

//bank_account server

#include <vxWorks.h>
#include "bankImpl.h"

extern CORBA::ORB_var orb;

// Declare global objects
AccountRegistry AccountManagerImpl::_accounts;

static void bank_server(void);

void start_bank_server(void)
{
    char * taskName = "BANK_SRVR";
    int Prio = 100;
    int option = VX_FP_TASK;
    int stackSize = 20000;

    taskSpawn(taskName,
              Prio,
              option,
              stackSize,
              (FUNCPTR)bank_server,
              0,0,0,0,0,0,0,0,0,0);
}

```

```

void bank_server()
{
    PortableServer::POA_var rootPOA;

    VISTRY {
        //get a reference to the root POA
        CORBA::Object_var obj =
orb->resolve_initial_references("RootPOA");

        VISIFNOT_EXCEP
            rootPOA = PortableServer::POA::_narrow(obj);
        VISEND_IFNOT_EXCEP

        CORBA::PolicyList policies;
        policies.length(1);

        VISIFNOT_EXCEP
            policies[(CORBA::ULong)0] =
rootPOA->create_lifespan_policy(
    PortableServer::PERSISTENT);
        VISEND_IFNOT_EXCEP

        // get the POA Manager
        PortableServer::POAManager_var poa_manager;
        VISIFNOT_EXCEP
            poa_manager = rootPOA->the_POAManager();
        VISEND_IFNOT_EXCEP

        // Create myPOA with the right policies
        PortableServer::POA_var myPOA;
        VISIFNOT_EXCEP
            myPOA = rootPOA->create_POA("bank_account_poa",
poa_manager, policies);
        VISEND_IFNOT_EXCEP

        // Create the servant
        AccountManagerImpl *managerServant = new AccountManagerImpl;

        // Create the object ID
        PortableServer::ObjectId_var managerId;
        VISIFNOT_EXCEP
            managerId =
PortableServer::string_to_ObjectId("BankManager");
        VISEND_IFNOT_EXCEP

        // Activate the servant with the ID on myPOA
        VISIFNOT_EXCEP
            myPOA->activate_object_with_id(managerId, managerServant);
        VISEND_IFNOT_EXCEP

        // Activate the POA Manager VISIFNOT_EXCEP
            poa_manager->activate(); VISEND_IFNOT_EXCEP

        CORBA::Object_var ref;

        VISIFNOT_EXCEP
            ref = myPOA->servant_to_reference(managerServant);
        VISEND_IFNOT_EXCEP

        CORBA::String_var string_ref;

        VISIFNOT_EXCEP
            string_ref = orb->object_to_string(ref.in());
        VISEND_IFNOT_EXCEP

        VISIFNOT_EXCEP
            cout << endl << "CORBA Object ==> " << endl << endl;
            cout << ref << endl;
            cout << string_ref << endl << endl;
            cout << " is ready" << endl << endl;
        VISEND_IFNOT_EXCEP
    }
}

```

```
VISCATCH(CORBA::Exception, e) {  
    cerr << e << endl;  
    taskSuspend(0);  
}  
VISEND_CATCH  
  
return;  
}
```



# Using POAs

*This chapter describes the Portable Object Adapter (POA), instances of which are used in the construction of the server-side of VisiBroker applications. The description of the POA in this chapter is derived from the corresponding chapter of the CORBA specification, which should be consulted for a complete description.*

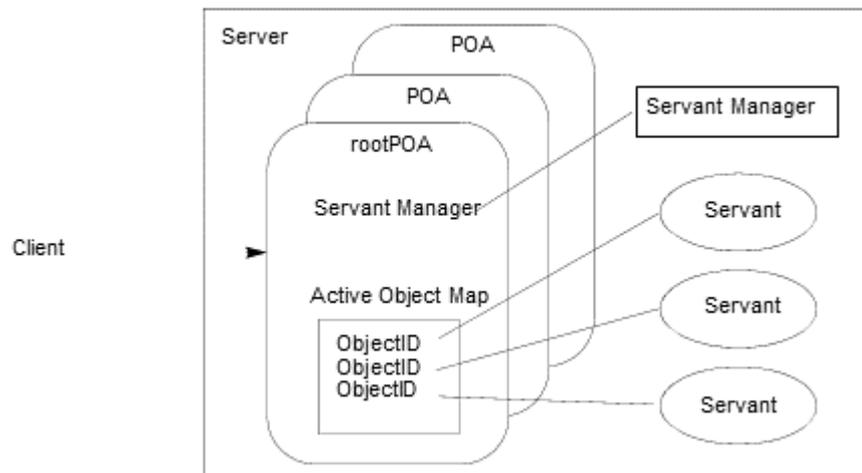
*Advanced server-side configuration topics, which involve the POA, are described in the chapters of [Part 6](#) of this manual ('Advanced Concepts').*

## What is a Portable Object Adapter?

A POA is the intermediary between the implementation of an object (a 'servant') and the ORB. In its role as an intermediary, a POA routes requests to servants. If necessary, it may cause servants and even other POAs to be created.

An ORB can support multiple POAs. At least one POA is always present, which is called the rootPOA. The rootPOA is created automatically. All other POAs are created by the application. The set of POAs is hierarchical; all POAs have the rootPOA as an ancestor.

Servant managers locate and assign servants to objects for the POA. When an abstract object is assigned to a servant, it is called an active object and the servant is said to incarnate the active object. Every POA has one Active Object Map which keeps track of the object IDs of active objects and their associated active servants.



**Figure 10** Overview of the POA

## POA terminology

The following are definitions of some terms with which you will become more familiar as you read through this chapter.

**Table 4** Portable Object Adapter terminology

Term	Description
Active Object Map	Table that maps active CORBA objects (through their object IDs) to servants. There is one Active Object Map per POA.
adapter activator	Object that can create a POA on demand when a request is received for a child POA that does not exist.
etherealize	Remove the association between a servant and an abstract CORBA object. <code>incarnateAssociate</code> a servant with an abstract CORBA object.
ObjectID	Way to identify a CORBA object within the object adapter. An ObjectID can be assigned by the object adapter or the application and is unique only within the object adapter in which it was created. Servants are associated with abstract objects through ObjectIDs.
persistent object	CORBA objects that live beyond the ORB instance was used to create them.
POA manager	Object that controls the state of the POA; for example, whether the POA is receiving or discarding incoming requests.
Policy	Object that controls the behavior of the associated POA and the objects the POA manages.
rootPOA	Each ORB is created with one POA called the rootPOA. You can create additional POAs (if necessary) from the rootPOA.
servant	Any code that implements the methods of a CORBA object, but is not the CORBA object itself.
servant manager	An object responsible for managing the association of objects with servants, and for determining whether an object exists. More than one servant manager can exist.
transient object	A CORBA object that lives only within the ORB instance that created it.

## Steps for creating and using POAs

Although the exact process can vary, the following are the basic steps that occur during the POA lifecycle:

- Define the POA's policies.
- Create the POA.
- Activate the POA through its POA manager.
- Create and activate servants.
- Create and use servant managers.
- Use adapter activators.

Depending on your needs, some of these steps may be optional. For example, you only have to activate the POA if you want it to process requests.

# POA policies

Each POA has a set of policies that define its characteristics. When creating a new POA, you can use the default set of policies or use different values to suit your requirements. You can only set the policies when creating a POA; you can not change the policies of an existing POA. POAs do not inherit the policies from their parent POA.

The following sections lists the POA policies, their values, and the default value (used by the rootPOA).

## Minimum CORBA and POA Policies

VisiBroker delivers both a "full" and "minimum" version of the ORB libraries. In VisiBroker-RT for C++ for Tornado the "minimum" version supports the full set of POA policy values.

For details, see the minimum CORBA specification document, OMG document number orbos/98-08-04. This document is available for download using the URL <ftp://ftp.omg.org/pub/docs/orbos/98-08-04.pdf>.

## Thread policy

The thread policy specifies the threading model to be used by the POA. The thread policy can have the following values:

ORB_CTRL_MODEL	The POA is responsible for assigning requests to threads.  In a multi-threaded environment, concurrent requests may be delivered to the same servant via using multiple threads.
SINGLE_THREAD_MODEL	The POA processes requests sequentially. In a multi-threaded environment, all calls made by the POA to servants and servant managers are thread-safe.  This policy value is NOT SUPPORTED in VisiBroker-RT which always supports the multithreaded behavior

---

### Default Values

- Root POA default: ORB\_CTRL\_MODEL
- Child POA default: ORB\_CTRL\_MODEL

## Lifespan policy

The lifespan policy specifies the lifespan of the objects implemented in the POA. The lifespan policy can have the following values:

TRANSIENT	A transient object activated by a POA cannot outlive the POA that created it. Once the POA is deactivated, an OBJECT_NOT_EXIST exception occurs if an attempt is made to use any object references generated by the POA.
PERSISTENT	A persistent object activated by a POA can outlive the ORB instance under which it was first created. Requests invoked on a persistent object may result in the implicit activation of a POA and the servant that implements the object.

---

### Default Values

- Root POA default: TRANSIENT
- Child POA default: TRANSIENT

## Object ID Uniqueness policy

The Object ID Uniqueness policy allows a single servant to be shared by many Object ID's (and hence object references). The Object ID Uniqueness policy can have the following values:

UNIQUE_ID	Activated servants support only one Object ID.
MULTIPLE_ID	Activated servants can have one or more Object IDs. The Object ID must be determined within the method being invoked at run time.

---

### Default Values

- Root POA default: UNIQUE\_ID
- Child POA default: UNIQUE\_ID

## ID Assignment policy

The ID assignment policy specifies whether object IDs are generated by server applications or by the POA. The ID Assignment policy can have the following values:

USER_ID	Objects are assigned object IDs by the application.
SYSTEM_ID	Objects are assigned object IDs by the POA. If the PERSISTENT policy is also set, object IDs must be unique across all instantiations of the same POA.

---

Typically, USER\_ID is used for persistent objects, and SYSTEM\_ID is used for transient objects. If you want to use SYSTEM\_ID value for persistent objects, you can extract them from the servant or object reference.

### Default Values

- Root POA default: SYSTEM\_ID
- Child POA default: SYSTEM\_ID

## Servant Retention policy

The Servant Retention policy specifies whether the POA retains active servants in the Active Object Map. The Servant Retention policy can have the following values:

RETAIN	The POA tracks object activations in the Active Object Map. RETAIN is usually used with ServantActivators or explicit activation methods on POA.
NON_RETAIN	The POA does not retain active servants in the Active Object Map.

---

ServantActivators and ServantLocators are types of servant managers. For more information on servant managers, see [“Using servants and servant managers”](#).

### Default Values

- Root POA default: RETAIN
- Child POA default: RETAIN

## Request Processing policy

The Request Processing policy specifies how requests are processed by the POA. `USE_ACTIVE_OBJECT_MAP_ONLY` If the Object ID is not listed in the Active Object Map, an `OBJECT_NOT_EXIST` exception is returned. The POA must also use the `RETAIN` policy with this value.

<code>USE_DEFAULT_SERVANT</code>	If the Object ID is not listed in the Active Object Map or the <code>NON_RETAIN</code> policy is set, the request is dispatched to the default servant. If no default servant has been registered, an <code>OBJ_ADAPTER</code> exception is returned. The POA must also use the <code>MULTIPLE_ID</code> policy with this value.
<code>USE_SERVANT_MANAGER</code>	If the Object ID is not listed in the Active Object Map or the <code>NON_RETAIN</code> policy is set, the servant manager is used to obtain a servant.

---

### Default Values

- Root POA default: `USE_ACTIVE_OBJECT_MAP_ONLY`
- Child POA default: `USE_ACTIVE_OBJECT_MAP_ONLY`

## Implicit Activation policy

The Implicit Activation policy specifies whether the POA supports implicit activation of servants. The Implicit Activation policy can have the following values:

<code>IMPLICIT_ACTIVATION</code>	The POA supports implicit activation of servants.  Servants can be activated by converting them to an object reference with <code>POA::servant_to_reference()</code> or by invoking <code>_this()</code> on the servant. The POA must also use the <code>SYSTEM_ID</code> and <code>RETAIN</code> policies with this value.
<code>NO_IMPLICIT_ACTIVATION</code>	The POA does not support implicit activation of servants.

---

### Default Values

- Root POA default: `IMPLICIT_ACTIVATION`
- Child POA default: `NO_IMPLICIT_ACTIVATION`

## Bind Support policy

The Bind Support policy (a VisiBroker-RT for C++-specific policy) controls the registration of POAs and active objects with the VisiBroker-RT for C++ osagent. If you have several thousands of objects, it is not feasible to register all of them with the osagent. Instead, you can register the POA with the osagent. When a client request is made, the POA name and the object ID is included in the bind request so that the osagent can correctly forward the request.

The BindSupport policy can have the following values:

BY_INSTANCE	All active objects are registered with the osagent. The POA must also use the PERSISTENT and RETAIN policy with this value.
BY_POA	Only POAs are registered with the osagent. The POA must also use the PERSISTENT policy with this value.
NO_REGISTRATION	Neither POAs nor active objects are registered with the osagent.

---

### Default Values

- Root POA default: BY\_POA
- Child POA default: BY\_POA

## Server Engine policy

The Server Engine policy (a VisiBroker-RT for C++-specific policy) controls the association of POAs with Server Engines.

The value of a Server Engine policy is a `CORBA::StringSequence` specifying a list of Server Engines that a particular POA is to be associated with. For details on using a Server Engine policy, see [“Associating a POA with Server Engines”](#).

## Creating POAs

To implement objects using the POA, at least one POA object must exist on the server. To ensure that a POA exists, a rootPOA is provided during the ORB initialization. This POA uses the default POA policies described earlier in this chapter.

Once the rootPOA is obtained, you can create child POAs that implement a specific server-side policy set.

### POA naming convention

Each POA keeps track of its name and its full POA name (the full hierarchical path name.) The hierarchy is indicated by a slash (/). For example, `/A/B/C` means that POA C is a child of POA B, which in turn is a child of POA A. The first slash (see the above example) indicates the rootPOA. If the Bind Support:BY\_POA policy is set on POA C, then `/A/B/C` is registered with the osagent and the client binds with `/A/B/C`.

If your POA name contains escape characters or other delimiters, VisiBroker precedes these characters with a double backslash (\\) when recording the names internally. For example, if you have two POAs in a hierarchy like

```
PortableServer::POA_var myPOA1 = rootPOA->create_POA("A/B",
poa_manager,policies);
PortableServer::POA_var myPOA2 = myPOA1->create_POA("\t",
poa_manager,policies);
```

a client would bind using:

```
Bank::AccountManager_var manager = Bank::AccountManager::_bind("/
A\\B/\t", managerId);
```

## Obtaining the rootPOA

The following code sample illustrates how a server application can obtain its rootPOA.

### Code example 26 Obtaining the rootPOA

```
// Initialize the ORB.
CORBA::Object_var obj =
    orb->resolve_initial_references("RootPOA");
// get a reference to the root POA PortableServer::POA_var
rootPOA = PortableServer::POA::_narrow(obj);
```

### Note

The `resolve_initial_references` method returns a value of type `CORBA::object`. You are responsible for narrowing the returned object reference to the desired type, which is `PortableServer::POA` in the previous example.

## Setting the POA properties

Policies are not inherited from the parent POA. If you want a POA to have a specific characteristic, you must identify all the policies that are different from the default value. For more information about POA policies, see [“POA policies”](#).

### Code example 27 Example of creating policies for a POA

```
CORBA::PolicyList policies;
policies.length(1);
policies[(CORBA::ULong)0] =
rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);
```

## Creating and activating the POA

A POA is created using `create_POA` on its parent POA. You can name the POA anything you like; however, the name must be unique with respect to all other POAs with the same parent. If you attempt to give two POAs the same name, a CORBA exception (`AdapterAlreadyExists`) is raised.

To create a new POA, use `create_POA` as follows:

```
PortableServer::POA_ptr create_POA(POA_Name, POAManager,
PolicyList);
```

The POA manager controls the state of the POA (for example, whether it is processing requests). If `null` is passed to `create_POA` as the POA manager name, a new POA manager object is created and associated with the POA. Typically, you'll want to have the same POA manager for all POAs. For more information about the POA manager, see [“Managing POAs with the POA manager”](#).

POA managers (and POAs) are not automatically activated once created. Use `activate()` to activate the POA manager associated with your POA.

**Code example 28** Example of creating a POA

```
CORBA::PolicyList policies;
policies.length(1);
policies[(CORBA::ULong)0] =
rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);

// Create myPOA with the right policies
VISIFNOT_EXCEP
PortableServer::POAManager_var rootManager =
rootPOA->the_POAManager();
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
PortableServer::POA_var myPOA =rootPOA->create_POA
("bank_agent_poa", rootManager, policies);
VISEND_IFNOT_EXCEP
```

## Activating objects

When CORBA objects are associated with an active servant, that CORBA Object is considered Activated. If that POA's Servant Retention Policy is `RETAIN`, then the associated object ID of that CORBA Object is recorded in the POA's Active Object Map.

CORBA Object Activation can occur in one of several ways:

- **Explicit activation**  
The server application itself explicitly activates objects by calling `activate_object` or `activate_object_with_id`.
- **On-demand activation**  
The server application instructs the POA to activate objects through a user-supplied servant manager. The servant manager must first be registered with the POA through `set_servant_manager`.
- **Implicit activation**  
The server activates objects solely in response to certain operations. If a servant is not active, there is nothing a client can do to make it active (for example, requesting for an inactive object does not make it active.)
- **Default servant**  
The POA uses a single servant to implement all of its objects.

## Activating objects explicitly

By setting `IdAssignmentPolicy::SYSTEM_ID` on a POA, objects can be explicitly activated without having to specify an object ID. The server invokes `activate_object` on the POA which activates, assigns and returns an object ID for the object. This type of activation is most common for transient objects. No servant manager is required since neither the object nor the servant is needed for very long.

Objects can also be explicitly activated using object IDs. A common scenario is during server initialization where the user invokes `activate_object_with_id` to activate all the objects managed by the server. No servant manager is required since all the objects are already activated. If a request for a non-existent object is received, an `OBJECT_NOT_EXIST` exception is raised. This has obvious negative effects if your server manages large numbers of objects.

### Code example 29 Example of explicit activation using `activate_object_with_id`

```
// Create the servant
AccountManagerImpl managerServant;
// Decide on the ID for the servant
PortableServer::ObjectId_var managerId =
    PortableServer::string_to_ObjectId("BankManager");
// Activate the servant with the ID on myPOA
VISIFNOT_EXCEP
    myPOA->activate_object_with_id(managerId, &managerServant);
VISEND_IFNOT_EXCEP
// Activate the POA Manager
VISIFNOT_EXCEP
    PortableServer::POAManager_var rootManager =
        rootPOA->the_POAManager();
VISEND_IFNOT_EXCEP

rootManger->activate();
```

## Activating objects on demand

On-demand activation occurs when a client requests an object that does not have an associated servant. After receiving the request, the POA searches the Active Object Map for an active servant associated with the object ID. If none is found, the POA invokes `incarnate` on the servant manager which passes the object ID value to the servant manager. The servant manager can do one of three things:

- Find an appropriate servant which then performs the appropriate operation for the request
- Raise an `OBJECT_NOT_EXIST` exception that is returned to the client
- Forward the request to another object

The POA policies determine any additional steps that may occur. For example, if `RequestProcessingPolicy::USE_SERVANT_MANAGER` and `ServantRetentionPolicy::RETAIN` are enabled, the Active Object Map is updated with the servant and object ID association.

An example of on-demand activation is shown in [Code example 32](#).

## Activating objects implicitly

A servant can be implicitly activated by certain operations if the POA has been created with `ImplicitActivationPolicy::IMPLICIT_ACTIVATION`, `IdAssignmentPolicy::SYSTEM_ID` and `ServantRetentionPolicy::RETAIN`. Implicit activation can occur with:

- the `POA::servant_to_reference` member function
- the `POA::servant_to_id` member function
- the `_this()` servant member function

If the POA has `ObjectIdUniquenessPolicy::UNIQUE_ID` set, implicit activation can occur when any of the above operations are performed on an inactive servant.

If the POA has `ObjectIdUniquenessPolicy::MULTIPLE_ID` set, `servant_to_reference` and `servant_to_id` operations always perform implicit activation, even if the servant is already active.

## Activating with the default servant

Use the `RequestProcessing::USE_DEFAULT_SERVANT` policy to have the POA invoke the same servant no matter what the object ID is. This is useful when little data is associated with each object.

**Code example 30** Example of activating all objects with the same servant

```
void bank_server()
{
    PortableServer::POA_var rootPOA;
    PortableServer::Current_var cur;

    VISTRY {
        cur = PortableServer::Current::_instance();

        CORBA::Object_var obj;
        // get a reference to the root POA
        VISIFNOT_EXCEP
        obj = orb->resolve_initial_references("RootPOA");
        VISEND_IFNOT_EXCEP

        VISIFNOT_EXCEP
        rootPOA = PortableServer::POA::_narrow(obj);
        VISEND_IFNOT_EXCEP

        // Create policies for our persistent POA
        CORBA::PolicyList policies;
        policies.length(3);
        VISIFNOT_EXCEP
        policies[(CORBA::ULong)0] =
            rootPOA->create_lifespan_policy(
PortableServer::PERSISTENT);
        VISEND_IFNOT_EXCEP

        VISIFNOT_EXCEP
        policies[(CORBA::ULong)1] =
            rootPOA->create_request_processing_policy(
PortableServer::USE_DEFAULT_SERVANT);
        VISEND_IFNOT_EXCEP

        VISIFNOT_EXCEP
        policies[(CORBA::ULong)2] =
            rootPOA->create_id_uniqueness_policy(
PortableServer::MULTIPLE_ID);
        VISEND_IFNOT_EXCEP

        PortableServer::POAManager_var poa_manager;
        VISIFNOT_EXCEP
        poa_manager = rootPOA->the_POAManager();
        VISEND_IFNOT_EXCEP

        // Create myPOA with the right policies
        PortableServer::POA_var myPOA;
        VISIFNOT_EXCEP
        myPOA = rootPOA->create_POA("bank_default_servant_poa",
            poa_manager, policies);
        VISEND_IFNOT_EXCEP

        // Set the default servant
        AccountManagerImpl *managerServant;
        VISIFNOT_EXCEP
        managerServant = new AccountManagerImpl(cur);
        VISEND_IFNOT_EXCEP

        VISIFNOT_EXCEP
        myPOA->set_servant(managerServant);
        VISEND_IFNOT_EXCEP
    }
}
```

```

    // Call _remove_ref since POA will invoke _add_ref on the
    default servant
    managerServant->_remove_ref();

    // Activate the POA Manager
    VISIFNOT_EXCEP
    poa_manager->activate();
    VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
    cout << "Bank Manager is ready" << endl;
    VISEND_IFNOT_EXCEP

}
VISICATCH(CORBA::Exception, e) {
    cerr << e << endl;
    return;
}
VISEND_CATCH

return;
}

```

## Deactivating objects

A POA can remove a servant from its Active Object Map. This may occur, for example, as some form of garbage-collection scheme. When the servant is removed from the map, it is deactivated. You can deactivate an object using `deactivate_object()`. When an object is deactivated, it doesn't mean this object is lost forever. It can always be reactivated at a later time.

### Code example 31 Example of deactivating an object

```

// DeActivatorThread
class DeActivatorThread: public VISThread {
private :
    PortableServer::ObjectId _oid;
    PortableServer::POA_ptr _poa;

public :
    virtual ~DeActivatorThread(){}
    // Constructor
    DeActivatorThread(const PortableServer::ObjectId& oid,
        PortableServer::POA_ptr poa ): _oid(oid), _poa(poa) {
        // start the thread
        run();
    }

    // implement begin() callback
    void begin() {
        // Sleep for 15 seconds
        VISPortable::vsleep(15);

        CORBA::String_var s = PortableServer::ObjectId_to_string
        (_oid);
        // Deactivate Object
        cout << "\nDeactivating the object with ID =" << s << endl;
        if ( _poa )
            _poa->deactivate_object( _oid );
    }
};

```

## Using servants and servant managers

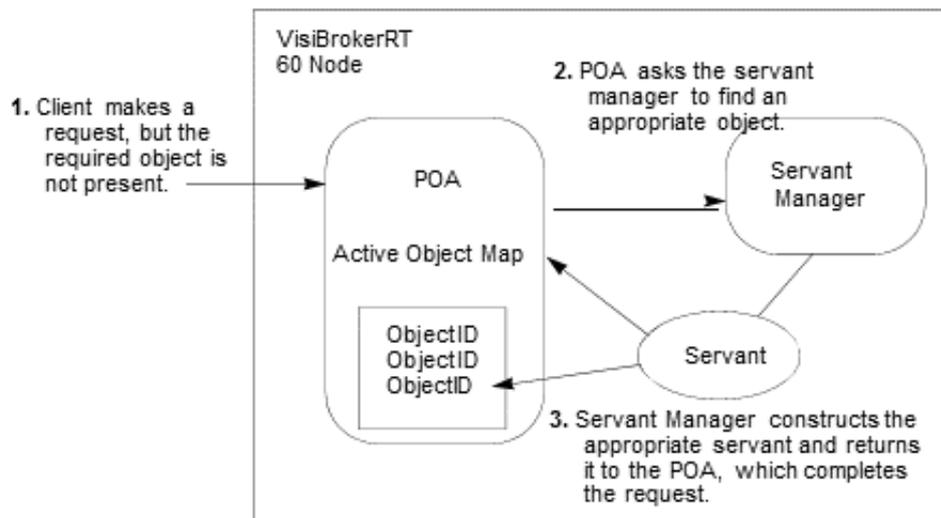
Servant managers perform two types of operations:

- 1 find and return a servant, and

2 deactivate a servant.

They allow the POA to activate objects when a request for an inactive object is received. Servant managers are optional. For example, servant managers are not needed when your application creates all CORBA objects at startup. Servant managers may also inform clients to forward requests to another object using `ForwardRequest`.

A servant is an active instance of an implementation. The POA maintains a map of the active servants and the object IDs of the servants. When a client request is received, the POA first checks this map to see if the object ID (embedded in the client request) has been recorded. If it exists, then the POA forwards the request to the servant. If the object ID is not found in the map, the servant manager is asked to locate and activate the appropriate servant. This is only an example scenario; the exact scenario depends on what POA policies you have in place.



**Figure 11** Example service manager function

There are two types of servant managers: *ServantActivator* and *ServantLocator*. The type of policy already in place determines which servant manager is used. For more information on POA policy, see "[POA policies](#)". Typically, a *ServantActivator* activates persistent objects and a *ServantLocator* activates transient objects.

To use servant managers,

`RequestProcessingPolicy::USE_SERVANT_MANAGER` must be set as well as the policy which defines the type of servant manager (`ServantRetentionPolicy::RETAIN` for *ServantActivator* or `ServantRetentionPolicy::NON_RETAIN` for *ServantLocator*.)

## ServantActivators

*ServantActivators* are used when `ServantRetentionPolicy::RETAIN` and `RequestProcessingPolicy::USE_SERVANT_MANAGER` are set. Servants activated by this type of servant manager are tracked in the Active Object Map.

The following events occur while processing requests using servant activators:

- 1 A client request is received (client request contains the POA name, the object ID.)
- 2 The POA first checks the active object map. If the object ID is found there, the operation is passed to the servant, and the response is returned to the client.
- 3 If the object ID is not found in the active object map, the POA invokes `incarnate` on a servant manager. `incarnate` passes the object ID and the POA in which the object is being activated.
- 4 The servant manager locates the appropriate servant.
- 5 The object ID is entered into the active object map, and the response is returned to the client.

### Note

The `etheralize` and `incarnate` method implementations are user-supplied code.

At a later date, the servant can be deactivated. This may occur from several sources, including the `deactivate_object` operation, deactivation of the POA manager associated with that POA, and so forth. More information on deactivating objects is described in [“Deactivating objects”](#).

### Code example 32 Example server code illustrating servant activator-type servant manager

```
void bank_server()
{
    VISTRY {
        // get a reference to the root POA
        CORBA::Object_var obj;
        VISIFNOT_EXCEP
        obj = orb->resolve_initial_references("RootPOA");
        VISEND_IFNOT_EXCEP

        VISIFNOT_EXCEP
        rootPOA = PortableServer::POA::_narrow(obj);
        VISEND_IFNOT_EXCEP

        CORBA::PolicyList policies;
        policies.length(2);
        VISIFNOT_EXCEP
        policies
            [(CORBA::ULong)0] = rootPOA->create_lifespan_policy(
PortableServer::PERSISTENT);
        VISEND_IFNOT_EXCEP

        VISIFNOT_EXCEP
        policies[(CORBA::ULong)1] =
            rootPOA->create_request_processing_policy(
PortableServer::USE_SERVANT_MANAGER);
        VISEND_IFNOT_EXCEP

        PortableServer::POAManager_var poa_manager;
        VISIFNOT_EXCEP
        poa_manager = rootPOA->the_POAManager(); VISEND_IFNOT_EXCEP

        PortableServer::POA_var myPOA; VISIFNOT_EXCEP
        // Create myPOA with the right policies
        myPOA = rootPOA->create_POA("bank_servant_activator_poa",
            poa_manager, policies);
        VISEND_IFNOT_EXCEP
    }
}
```

```

// Create a Servant activator AccountManagerActivator
*servant_activator_impl;
VISIFNOT_EXCEP
servant_activator_impl = new AccountManagerActivator;
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
// Set the servant activator
myPOA->set_servant_manager(servant_activator_impl);
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
// Activate the POA Manager poa_manager->activate();
VISEND_IFNOT_EXCEP

// Waiting for incoming requests
cout << " BankManager is ready" << endl;
}
VISCATCH(CORBA::Exception, e) { cerr << e << endl;
return;
} VISEND_CATCH

return;
}

```

The servant manager for this example follows.

**Code example 33** Servant manager for servant activator example

```

// Servant Activator
class AccountManagerActivator : public
PortableServer::ServantActivator {
public:
    virtual PortableServer::Servant incarnate (const
PortableServer::ObjectId& oid,
        PortableServer::POA_ptr poa) {
        CORBA::String_var s = PortableServer::ObjectId_to_string
(oid);
        cout << "\nAccountManagerActivator.incarnate called with ID
= " << s << endl;
        PortableServer::Servant servant;

if ( VISPortable::vstricmp( (char *)s,
    "SavingsAccountManager" ) == 0 )
    // Create CheckingAccountManager Servant
    servant = new SavingsAccountManagerImpl;
else if ( VISPortable::vstricmp( (char *)s,
    "CheckingAccountManager" ) == 0 )
    // Create CheckingAccountManager Servant
    servant = new CheckingAccountManagerImpl;
else
    VISTHROW(CORBA::OBJECT_NOT_EXIST());
// Create a deactivator thread new DeActivatorThread( oid, poa
);

// return the servant
servant->_add_ref();
return servant;
}
virtual void etherealize (const PortableServer::ObjectId& oid,
    PortableServer::POA_ptr adapter,
    PortableServer::Servant servant,
    CORBA::Boolean cleanup_in_progress,
    CORBA::Boolean remaining_activations) {
// If there are no remaining activations i.e ObjectIds
// associated with the servant delete it.
CORBA::String_var s = PortableServer::ObjectId_to_string
(oid);
    cout << "\nAccountManagerActivator.etherealize called with ID
= " << s << endl;
    if (!remaining_activations) delete servant;
}
};

```

## ServantLocators

In many situations, the POA's Active Object Map could become quite large and consume memory. To reduce memory consumption, a POA can be created with `RequestProcessingPolicy::USE_SERVANT_MANAGER` and `ServantRetentionPolicy::NON_RETAIN`, meaning that the servant-to-object association is not stored in the active object map. Since no association is stored, `ServantLocator` servant managers are invoked for each request.

The following events occur while processing requests using servant locators:

- 1 A client request, which contains the POA name and the object id, is received.
- 2 Since `ServantRetentionPolicy::NON_RETAIN` is used, the POA does not search the active object map for the object ID.
- 3 The POA invokes `preinvoke` on a servant manager. `preinvoke` passes the object ID, the POA in which the object is being activated, and a few other parameters.
- 4 The servant locator locates the appropriate servant.
- 5 The operation is performed on the servant and the response is returned to the client.
- 6 The POA invokes `postinvoke` on the servant manager.

### Note

The `preinvoke` and `postinvoke` methods are user-supplied code.

### Code example 34 Example server code illustrating servant locator-type servant managers

```
void bank_server()
{
    VISTRY {
        //get a reference to the root POA
        CORBA::Object_var obj =
orb->resolve_initial_references("RootPOA");

        VISIFNOT_EXCEP
            rootPOA = PortableServer::POA::_narrow(obj);
        VISEND_IFNOT_EXCEP

        // Create a child POA with Persistence life span policy that
        // uses servant manager with non-retain retention policy(no
        // Active Object Map) causing the POA to use the servant locator.

        CORBA::PolicyList policies;
        policies.length(3);

        VISIFNOT_EXCEP
            policies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy(
PortableServer::PERSISTENT);
        VISEND_IFNOT_EXCEP

        VISIFNOT_EXCEP
            policies[(CORBA::ULong)1] =
                rootPOA->create_servant_retention_policy(
PortableServer::NON_RETAIN);
        VISEND_IFNOT_EXCEP

        VISIFNOT_EXCEP
            policies[(CORBA::ULong)2] =
```

```

        rootPOA->create_request_processing_policy(
PortableServer::USE_SERVANT_MANAGER);
    VISEND_IFNOT_EXCEP

    PortableServer::POAManager_var poa_manager;
    VISIFNOT_EXCEP
        poa_manager = rootPOA->the_POAManager();
    VISEND_IFNOT_EXCEP

    PortableServer::POA_var myPOA;
    VISIFNOT_EXCEP
        myPOA = rootPOA->create_POA("bank_servant_locator_poa",
            poa_manager, policies);
    VISEND_IFNOT_EXCEP

    // Create the servant locator
    AccountManagerLocator *servant_locator_impl;
    VISIFNOT_EXCEP
        servant_locator_impl = new AccountManagerLocator;
    VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
        myPOA->set_servant_manager(servant_locator_impl);
    VISEND_IFNOT_EXCEP

    // Activate the POA Manager
    VISIFNOT_EXCEP
        poa_manager->activate();
    VISEND_IFNOT_EXCEP

    // Ready for incoming requests
    VISIFNOT_EXCEP
        cout << "Bank Manager is ready" << endl;
    VISEND_IFNOT_EXCEP
}
VISICATCH(CORBA::Exception, e)
{ cerr << e << endl;
  return;
}
VISEND_CATCH

return;
}

```

The servant manager for this example follows.

**Code example 35** Servant manager for servant locator example

```

// Servant Locator
class AccountManagerLocator : public
    PortableServer::ServantLocator
{
public:
    AccountManagerLocator (){}

// preinvoke is very similar to ServantActivator's incarnate
// method but gets called every time a request comes in unlike
// incarnate() which gets called every time the POA does not find
// a servant in the active object map
virtual PortableServer::Servant preinvoke (const
    PortableServer::ObjectId& oid,
        PortableServer::POA_ptr adapter,
        const char* operation,
        PortableServer::ServantLocator::Cookie& the_cookie) {
    CORBA::String_var s = PortableServer::ObjectId_to_string
(oid);
    cout << "\nAccountManagerLocator.preinvoke called with ID =
        " << s << endl;
    PortableServer::Servant servant;

    if ( VISPortable::vstricmp( (char *)s, "SavingsAccountManager"
) == 0 )

```

```

        // Create CheckingAccountManager Servant
        servant = new SavingsAccountManagerImpl;
    else if ( VISPortable::vstricmp( (char *)s,
        "CheckingAccountManager" ) == 0 )
        // Create CheckingAccountManager Servant
        servant = new CheckingAccountManagerImpl;
    else
        VISTHROW(CORBA::OBJECT_NOT_EXIST());

    // Note also that we do not spawn of a thread to explicitly
    // deactivate an object unlike a servant activator, this is
    // because the POA itself calls post invoke after the request is
    // complete. In the case of a servant activator the POA calls
    // etherealize() only if the object is deactivated by calling
    // poa->de_activateobject or the POA itself is destroyed.

    // return the servant
    servant->_add_ref();
    return servant;
}

virtual void postinvoke (const PortableServer::ObjectId& oid,
    PortableServer::POA_ptr adapter,
    const char* operation,
    PortableServer::ServantLocator::Cookie the_cookie,
    PortableServer::Servant the_servant) {
    CORBA::String_var s = PortableServer::ObjectId_to_string
    (oid);
    cout << "\nAccountManagerLocator.postinvoke called with ID = "
    << s << endl;
    the_servant->_remove_ref();
}
};

```

## Managing POAs with the POA manager

A POA manager controls the state of the POA (whether requests are queued or discarded), and can deactivate the POA. Each POA is associated with a POA manager object. A POA manager can control one or many POAs.

A POA manager is associated with a POA when the POA is created. You can specify the POA manager to use, or specify `null` to have a new POA Manager created.

### Code example 36 Naming the POA and its POA Manager

```

PortableServer::POAManager_var rootManager =
    rootPOA->the_POAManager();

VISIFNOT_EXCEP
PortableServer::POA_var myPOA =
    rootPOA->create_POA("bank_servant_locator_poa", rootManager,
    policies);
VISEND_IFNOT_EXCEP

```

A POA manager is “destroyed” when all its associated POAs are destroyed.

A POA manager can have four states. These states in turn determine the state of the POA.

- Holding
- Active
- Discarding
- Inactive

## Getting the current state

To get the current state of the POA manager, use:

```
PortableServer::POAManager::State get_state();
```

The valid state values are:

```
enum State{HOLDING, ACTIVE, DISCARDING, INACTIVE};
```

## Holding state

By default, when a POA manager is created, it is in the holding state. When the POA manager is in the holding state, the POA queues all incoming requests.

Requests that require an adapter activator are also queued when the POA manager is in the holding state.

To change the state of a POA manager to holding, use `void hold_requests(wait_for_completion)` raises `(AdapterInactive)`;

`wait_for_completion` is Boolean. If `FALSE`, this operation returns immediately after changing the state to holding. If `TRUE`, this operation returns only when all requests started prior to the state change have completed or when the POA manager is changed to a state other than holding. `AdapterInactive` is the exception raised if the POA manager was in the inactive state prior to calling this operation.

### Note

POA managers in the inactive state cannot change to the holding state.

Any requests that have been queued but not yet started will continue to be queued during the holding state.

## Active state

When the POA manager is in the active state, its associated POAs process requests.

To change the POA manager to the active state, use:

```
void activate()  
raises (AdapterInactive);
```

`AdapterInactive` is the exception raised if the POA manager was in the inactive state prior to calling this operation.

### Note

POA managers currently in the inactive state can not change to the active state.

## Discarding state

When the POA manager is in the discarding state, its associated POAs discard all requests that have not yet started. In addition, the adapter activators registered with the associated POAs are not called. This state is useful when the POA is receiving too many requests. You need to notify the client that their request has been discarded and to resend their request. There is no inherent behavior for determining if and when the POA is receiving too many requests.

To change the POA manager to the discarding state, use:

```
void discard_requests(wait_for_completion)
    raises (AdapterInactive);
```

The `wait_for_completion` option is Boolean. If `FALSE`, this operation returns immediately after changing the state to discarding. If `TRUE`, this operation returns only when all requests started prior to the state change have completed or when the POA manager is changed to a state other than discarding. `AdapterInactive` is the exception raised if the POA manager was in the inactive state prior to calling this operation.

### Note

POA managers currently in the inactive state can not change to the discarding state.

## Inactive state

When the POA manager is in the inactive state, its associated POAs reject incoming requests. This state is used when the associated POAs are to be shut down.

### Note

POA managers in the inactive state can not change to any other state.

To change the POA manager to the inactive state, use

```
void deactivate(etherealize_objects, wait_for_completion)
    raises (AdapterInactive);
```

After the state changes, if `etherealize_objects` is `TRUE`, then all associated POAs that have `Servant RetentionPolicy::RETAIN` and `RequestProcessingPolicy::USE_SERVANT_MANAGER` set call `etherealize` on the servant manager for all active objects. If `etherealize_objects` is `FALSE`, then `etherealize` is not called.

The `wait_for_completion` option is Boolean. If `FALSE`, this operation returns immediately after changing the state to inactive. If `TRUE`, this operation returns only when all requests started prior to the state change have completed or `etherealize` has been called on all associated POAs (that have `Servant RetentionPolicy::RETAIN` and `RequestProcessingPolicy::USE_SERVANT_MANAGER`).

`AdapterInactive` is the exception raised if the POA manager was in the inactive state prior to calling this operation.

## Adapter activators

Adapter activators are associated with POAs and provide the ability to create child POAs

on-demand. This can be done during the `find_POA` operation, or when a request is received that names a specific child POA.

An adapter activator supplies a POA with the ability to create child POAs on demand, as a side-effect of receiving a request that names the child POA (or one of its children), or when `find_POA` is called with an `activate` parameter value of `TRUE`. An application server that creates all its needed POAs at the beginning of execution does not need to use or provide an adapter activator; it is necessary only for the case in which POAs need to be created during request processing.

While a request from the POA to an adapter activator is in progress, all requests to objects managed by the new POA (or any descendant POAs) will

be queued. This serialization allows the adapter activator to complete any initialization of the new POA before requests are delivered to that POA.

For an example on using adapter activators, see the POA adaptor\_activator example located in `<install_location>/VisiBrokerRT60/examples/poa/ adaptor_activator`.

## Processing requests

Requests contain the Object ID of the target object and the POA that created the target object reference. When a client sends a request, the ORB first locates the appropriate server, it then locates the appropriate POA within that server.

Once the ORB has located the appropriate POA, it delivers the request to that POA. How the request is processed at that point depends on the policies of the POA and the object's activation state. For information about object activation states, see ["Activating objects"](#).

- If the POA has `ServantRetentionPolicy::RETAIN`, the POA looks at the Active Object Map to locate a servant associated with the Object ID from the request. If a servant exists, the POA invokes the appropriate method on the servant.
- If the POA has `ServantRetentionPolicy::NON_RETAIN` or has `ServantRetentionPolicy::RETAIN` but did not find the appropriate servant, the following may take place:
  - If the POA has `RequestProcessingPolicy::USE_DEFAULT_SERVANT`, the POA invokes the appropriate method on the default servant.
  - If the POA has `RequestProcessingPolicy::USE_SERVANT_MANAGER`, the POA invokes `incarnate` or `preinvoke` on the servant manager.
  - If the POA has `RequestProcessingPolicy::USE_OBJECT_MAP_ONLY`, an exception is raised.

If a servant manager has been invoked but can not incarnate the object, the servant manager can raise a `ForwardRequest` exception.

# Using the Tie Mechanism

*This chapter describes how the tie mechanism may be used to integrate existing C++ code into a distributed object system. This chapter will enable you to create a delegation implementation or to provide implementation inheritance.*

## How does the tie mechanism work?

Object implementation classes normally inherit from a servant class generated by the `idl2cpp` compiler. The servant class, in turn, inherits from `PortableServer::Servant`. When it is not convenient or possible to change existing classes to inherit from the VisiBroker-RT for C++ servant skeleton class, the *tie* mechanism offers an appropriate alternative.

The tie mechanism provides object servers with a *delegator implementation* class that inherits from `PortableServer::Servant`. The delegator implementation does not provide any semantics of its own. It simply delegates every request it receives to the real implementation class, which can be implemented separately. The real implementation class is not required to inherit from `PortableServer::Servant`.

With using the tie mechanism, two additional generated classes are required:

- `<InterfaceName>POATie` defers implementation of all IDL defined methods to a delegate. The delegate implements the interface `<InterfaceName>Operations`. Legacy implementations can be trivially extended to implement the operations interface and in turn delegate to the real implementation.
- `<InterfaceName>Operations` defines all of the methods that must be implemented by the object implementation. This interface acts as the delegate object for the associated `<InterfaceName>POATie` class when the tie mechanism is used.

## Example program

### Location of an example program using the tie mechanism

A version of the Bank example using the tie mechanism can be found in the VisiBroker for C++ distribution under `<install_location>/VisiBrokerRT60/examples/basic/ bank_tie`.

## Looking at the tie template

The `idl2cpp` compiler will automatically generate a `_tie_Account` template class, as shown in [Code example 37](#). The `POA_Bank_Account_tie` class is instantiated by the object server and initialized with an instance of `AccountImpl`. The `POA_Bank_Account_tie` class delegates every operation request it receives to `AccountImpl`, the real implementation class. In this example, the class `AccountImpl` does not inherit from the `POA_Bank::Account` class.

### Code example 37 Looking at the POA\_Bank\_Account\_tie template

```
...
template <class T> class POA_Bank_Account_tie : public
POA_Bank::Account {
private:
    CORBA::Boolean _rel;
    PortableServer::POA_ptr _poa;
    T * _ptr;
    POA_Bank_Account_tie(const POA_Bank_Account_tie&) {}
    void operator=(const POA_Bank_Account_tie&) {}

public:
    POA_Bank_Account_tie (T& t): _ptr(&t), _poa(NULL),
    _rel((CORBA::Boolean)0) {}

    POA_Bank_Account_tie (T& t, PortableServer::POA_ptr poa):
    _ptr(&t),
    _poa(PortableServer::_duplicate(poa)),
    _rel((CORBA::Boolean)0) {}

    POA_Bank_Account_tie (T *p, CORBA::Boolean release= 1) :
    _ptr(p),
    _poa(NULL), _rel(release) {}

    POA_Bank_Account_tie (T *p, PortableServer::POA_ptr poa,
    CORBA::Boolean release =1):
    _ptr(p), _poa(PortableServer::_duplicate(poa)),
    _rel(release) {}

    virtual ~POA_Bank_Account_tie() {
        CORBA::release(_poa);
        if (_rel) {
            delete _ptr;
        }
    }
    T* _tied_object() { return _ptr; }

    void _tied_object(T& t) {
        if (_rel) {
            delete _ptr;
        }
        _ptr = &t;
        _rel = 0;
    }

    void _tied_object(T *p,
    CORBA::Boolean release=1) { if (_rel) {
        delete _ptr;
    }
        _ptr = p;
        _rel = release;
    }

    CORBA::Boolean _is_owner() { return _rel; }

    void _is_owner(CORBA::Boolean b) { _rel = b; }

    CORBA::Float balance() {
        return _ptr->balance();
    }

    PortableServer::POA_ptr _default_POA() {
        if ( !CORBA::is_nil(_poa) ) {
            return _poa;
        } else {
            return PortableServer_ServantBase::_default_POA();
        }
    }
};
```

## Changing the server to use the `_tie_account` class

Code example 38 shows the modifications to the `Server.C` file required to use the `_tie_account` class.

**Code example 38** Example of a server using the `_tie` class

```
//bank_tie_server
#include <vxWorks.h>
#include "corba.h"
#include "bankImpl.h"
/*-----*/
/* Forward Declarations. */
/*-----*/
extern "C" void start_bank_server(void);
static void bank_server(void);

extern CORBA::ORB_var orb;

// Static initialization
AccountRegistry AccountManagerImpl::_accounts;

void start_bank_server(void)
{
    char * taskName = "BANK SRVR"; intPrio = 100;
    int option = VX_FP_TASK;
    int stackSize = 20000;

    taskSpawn(taskName,
              Prio,
              option,
              stackSize,
              (FUNCPTR)bank_server,
              0,0,0,0,0,0,0,0,0,0);
}

void bank_server()
{
    PortableServer::POA_var rootPOA;

    VISTRY {

        //get a reference to the root POA
        CORBA::Object var obj =
            orb->resolve_initial_references("RootPOA");

        VISIFNOT_EXCEP
            rootPOA = PortableServer::POA::_narrow(obj);
        VISEND_IFNOT_EXCEP

        CORBA::PolicyList policies;
        policies.length(1);

        policies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy(
PortableServer::PERSISTENT);

        // get the POA Manager
        PortableServer::POAManager_var poa_manager;

        VISIFNOT_EXCEP
            poa_manager = rootPOA->the_POAManager();
        VISEND_IFNOT_EXCEP

        // Create myPOA with the right policies
        PortableServer::POA_var myPOA;

        VISIFNOT_EXCEP
            myPOA = rootPOA->create_POA("bank_account_poa", poa_manager,
policies);
        VISEND_IFNOT_EXCEP

        // Create the servant
```

```

    AccountManagerImpl *managerServant = new AccountManagerImpl;

    // Create the delegator
    POA_Bank_AccountManager_tie<AccountManagerImpl> *tieServer;

    VISIFNOT_EXCEP
        tieServer = new
    POA_Bank_AccountManager_tie<AccountManagerImpl>(*managerServant);
    VISEND_IFNOT_EXCEP

    // Create the object ID for the servant
    PortableServer::ObjectId_var managerId;

    VISIFNOT_EXCEP
        managerId =
    PortableServer::string_to_ObjectId("BankManager");
    VISEND_IFNOT_EXCEP

    // Activate the servant with the ID on myPOA
    VISIFNOT_EXCEP
        myPOA->activate_object_with_id((CORBA_OctetSequence
&)managerId,tieServer);
    VISEND_IFNOT_EXCEP

    // Activate the POA Manager
    VISIFNOT_EXCEP
        poa_manager->activate();
    VISEND_IFNOT_EXCEP

    CORBA::Object_var reference;
    VISIFNOT_EXCEP
        reference = myPOA->servant_to_reference(tieServer);
    VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
        cout << endl << "CORBA Object ==> " << endl << endl;
        cout << reference << endl;
        cout << " is ready" << endl << endl;
    VISEND_IFNOT_EXCEP

}
VISICATCH(CORBA::Exception, e) {
    cerr << e << endl;
    taskSuspend(0);
}
VISEND_CATCH
return;
}

```

## Building the tie example

The instructions described in the chapter [“Developing an Example Application with VisiBroker-RT for C++”](#) are also valid for building the tie example.

# Part 3

## Client Concepts

### **In this part**

This part contains the following chapters:

<a href="#">Client basics</a>	<a href="#">page 87</a>
-------------------------------	-------------------------



# Client basics

*This chapter describes how client programs access and use distributed objects.*

## Initializing the ORB

The Object Request Broker (ORB) provides a communication link between the client and the server. When a client makes a request, the ORB locates the object implementation, delivers the request to the object (and activates the object if necessary), and returns the response to the client. The client is unaware that the object may be on the same machine or across a network.

Though much of the work done by the ORB is transparent to you, your client program must explicitly initialize the ORB. ORB options, described in the VisiBroker-RT for C++ *Reference Guide* can be specified as command-line arguments. Therefore, you must pass `argc` and `argv` to `ORB_init` to ensure that these options take effect.

### Code example 39 Initializing the ORB

```
...
/*-----*/
/* function ==>do_corba */
/* This function will perform the necessary */
/* initialization for the ORB (i.e. ORB_init,...) */
/*-----*/

void do_corba(void)
{
    int argc = 3;
    char *argv[] = {"DO_CORBA", "-ORBagentport", OSAGENT_PORT};

    /*-----*/
    /* Call ORB_init */
    /*-----*/

    VISTRY
    {
        // Initialize the ORB
        orb = CORBA::ORB_init(argc, argv);
    }
    VISCATCH(CORBA::Exception, e)
    {
        cerr << e << endl;
        taskSuspend(0);
    }
    VISEND_CATCH

    return;
}
```

## Binding to objects

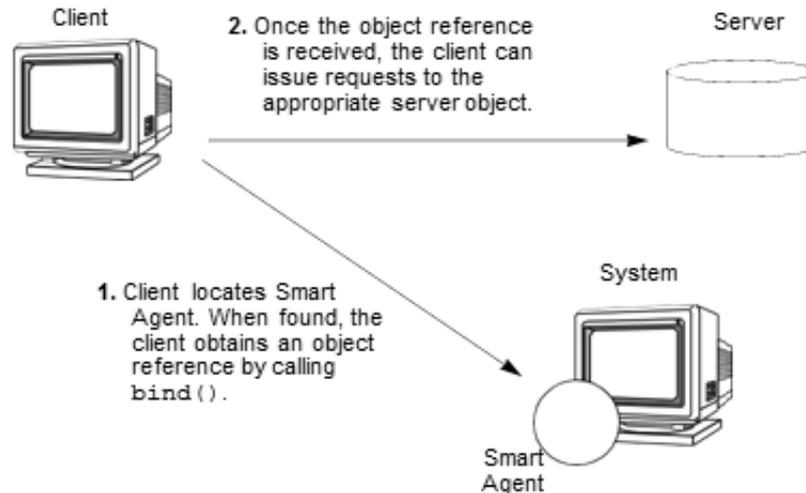
A client program uses a remote object by obtaining a reference to the object. Object references are usually obtained using the `<interface>_bind()` member function. The ORB hides most of the details involved with obtaining the object reference, such as locating the server that implements the object and establishing a connection to that server.

## Action performed during the bind process

When the CORBA server application starts, it performs a `CORBA::ORB_init()` and announces POA names and object ids to Smart Agents on the network.

When your client program invokes the `_bind()` member function, the ORB performs several functions on behalf of your program.

- The ORB contacts the Smart Agent to locate an object implementation that offers the requested interface. If an object name was specified when `_bind()` was invoked, that name will be used to further qualify the directory service search.
- When an object implementation is located, the ORB attempts to establish a connection between the object implementation that was located and your client program.
- Once the connection is successfully established, the ORB will create a proxy object and return a reference to that object. The client will invoke methods on the proxy object which will, in turn, interact with the server object.



**Figure 12** Client interaction with the Smart Agent

### Note

Your client program will never invoke a constructor for the server class. Instead, an object reference is obtained by invoking the static `_bind()` member function

There are two forms of the static `_bind()` member functions which are generated by the IDL2CPP compiler.

- 1 One form of the `_bind` interface must be used if the CORBA object implementation that the Client intends to bind to has been activated on a POA whose BIND SUPPORT POLICY was BY\_POA. This is referred to as the "2 parameter `_bind` interface". An example of the use of the 2 parameter `_bind` interface is shown below.

Note that `BY_POA` is the default policy value for the `BIND SUPPORT POLICY`:

**Code example 40** Example of use of the 2 parameter `_bind` interface.

```
...
PortableServer::ObjectId_var manager_id =
PortableServer::string_to_ObjectId("BankManager");
Bank::AccountManager_var = Bank::AccountManager::_bind("/
bank_agent_poa", manager_id);
...
```

2 The second form of the `_bind` interface must be used if the CORBA object implementation that the Client intends to bind to has been activated on a POA whose `BIND SUPPORT POLICY` value was `BY_INSTANCE`. This is referred to as the "*one parameter \_bind interface*". An example of the use of the one parameter `_bind` interface is shown below.

Note the one parameter `_bind` interface gives equivalent functionality as in prior versions of VisiBroker-RT for C++ (e.g version 3.2.2)

**Code example 41** Example of use of the 1 parameter `_bind` interface.

```
...
Bank::AccountManager_var =
Bank::AccountManager::_bind("BankManager");
...
```

For more information on the `BIND SUPPORT POLICY` please refer to "[Bind Support policy](#)".

## Invoking operations on an object

Your client program uses an object reference to invoke an operation on an object or to reference data contained by the object. "[Manipulating object references](#)" describes the variety of ways that object references can be manipulated.

**Code example 42** Invoking an operation using an object reference

```
...
// Invoke the balance operation.
balance = account->balance();
cout << "Balance is $" << balance << endl;
...
```

## Manipulating object references

The object reference returned to your client program by the `_bind()` member function represents a CORBA object. Your client program can use the object reference to invoke operations on the object that have been defined in the object's IDL interface specification. In addition, there are member functions that all ORB objects inherit from the class `CORBA::Object` that you can use to manipulate the object.

## Checking for nil references

You can use the `CORBA` class method `is_nil()` shown below to determine if an object reference is `nil`. This method returns 1 if the object reference passed is `nil`. It returns 0 if the object reference is not `nil`.

**Code example 43** Method for checking for a nil object reference

```
class CORBA {
...
static Boolean is _nil(CORBA::Object_ptr obj);
...
};
```

## Obtaining a nil reference

You can obtain a nil object reference using the `CORBA::Object` class `_nil()` member function. It returns a `NULL` value that is cast to an `Object_ptr`.

**Code example 44** Method for obtaining a nil reference

```
class Object {
...
static CORBA::Object_ptr _nil();
...
};
```

## Duplicating an object reference

When your client program invokes the `_duplicate` member function, the reference count for the object reference is incremented by one and the same object reference is returned. Your client program can use the `_duplicate()` member function to increase the reference count for an object reference so that the reference can be stored in a data structure or passed as a parameter. Increasing the reference count ensures that the memory associated with the object reference will not be freed until the reference count has reached zero.

The IDL compiler generates a `_duplicate()` member function for each object interface you specify. The `_duplicate()` member function accepts and returns a generic `Object_ptr`.

**Code example 45** Method for duplicating an object reference

```
class Object {
...
static CORBA::Object_ptr _duplicate(CORBA::Object_ptr obj);
...
};
```

### Note

The `_duplicate()` member function has no meaning for the POA or ORB because these objects do not support reference counting.

## Releasing an object reference

You should release an object reference when it is no longer needed. One way of releasing an object reference is by invoking the `CORBA::Object` class `_release()` member function.

### Caution

Always use the `_release()` member function. **Never invoke operator delete on an object reference.**

**Code example 46** Releasing an object reference

```
class CORBA { class Object {
...
...
};
```

```

        void _release();
        ...
    };
};

```

You may also use the CORBA class `release()` member function, which is provided for compatibility with the CORBA specification.

**Code example 47** CORBA method for releasing an object reference

```

class CORBA {
    ...
    static void release(Object_ptr);
    ...
};

```

## Obtaining the reference count

Each object reference has a reference count that you can use to determine how many times the reference has been duplicated. When you first obtain an object reference by invoking `_bind()`, the reference count is set to one. Releasing an object reference will decrement the reference count by one. Once the reference count reaches 0, VisiBroker-RT for C++ automatically deletes the object reference. [Code example 48](#) shows the `_ref_count()` member function for retrieving the reference count.

### Note

When a remote client duplicates or releases an object reference, the server's object reference count is not affected.

**Code example 48** Method for obtaining the reference count

```

class Object {
    ...
    CORBA::Long _ref_count() const;
    ...
};

```

## Converting a reference to a string

VisiBroker-RT for C++ provides an ORB class member function that allows you to convert an object reference to a string or convert a string back into an object reference. The CORBA specification refers to this process as stringification.

**Table 5** Member functions for stringification and de-stringification

Method	Description
<code>object_to_string</code>	Converts an object reference to a string.
<code>string_to_object</code>	Converts a string to an object reference.

A client program can use the `object_to_string` member function to convert an object reference to a string and pass it to another client program. The second client may then de-stringify the object reference, using the `string_to_object` member function and use the object reference without having to explicitly bind to the object.

### Note

The caller of `object_to_string()` is responsible for calling `CORBA::string_free()` on the returned string.

## Note

Transient object references (i.e. Object references created via a POA whose lifespan policy is set to TRANSIENT) that are stringified are not guaranteed to be valid beyond the life of the ORB instance that created the reference.

## Obtaining object and interface names

Table 6 shows the member functions provided by the `Object` class that you can use to obtain the interface and object names as well as the repository id associated with an object reference. The interface repository is discussed in “Using Interface Repositories”.

## Note

If you did not specify an object name when you invoked the `_bind()` member function, invoking the `_object_name()` member function with the resulting object reference will return `NULL`.

**Table 6** Member functions for obtaining interface and object names

Method	Description
<code>_interface_name</code>	Returns the interface name of this object.
<code>_object_name</code>	Returns this object’s name.
<code>_repository_id</code>	Returns the repository’s type identifier.

## Determining the type of an object reference

You can check whether an object reference is of a particular type by using the `_is_a()` member function. You must first obtain the repository id of the type you wish to check using the `_repository_id()` member function. This method returns 1 if the object is either an instance of the type represented by `repository_id()` or if it is a sub-type. The member function returns 0 if the object is not of the type specified. Note that this may require remote invocation to determine the type.

You can use the `_is_equivalent()` member function to check if two object references refer to the same object implementation. This member function returns 1 if the object references are equivalent. This member function returns 0 if the object references are distinct, but does not necessarily indicate that the object references are two distinct objects. This is a lightweight member function and does not involve actual communication with the server object.

The `_hash()` member function can be used to obtain a hash value for an object reference. While this value is not guaranteed to be unique, it will remain consistent through the lifetime of the object reference and can be stored in a hash table.

**Table 7** Member functions for determining the type of an object reference

Method	Description
<code>_hash</code>	Returns a hash value for the object reference.
<code>_is_a</code>	Determines if an object implements a specified interface.
<code>_is_equivalent</code>	Returns <code>true</code> if two objects refer to the same interface implementation.

## Determining the location and state of bound objects

Given a valid object reference, your client program can use the `_is_bound()` member function to determine if the object is bound, (i.e. if a connection is currently active for this object). The method returns 1 if the object is bound and 0 if the object is not bound.

The `_is_local()` member function returns 1 if the client program and the object implementation reside within the same address space.

The `_is_remote()` member function returns 1 if the client program and the object implementation reside in a different address space.

**Table 8** Member functions for determining location and state of object reference

Method	Description
<code>_is_bound</code>	Returns 1 if a connection is currently active for this object.
<code>_is_local</code>	Returns 1 if this object is implemented in the local address space.
<code>_is_remote</code>	Returns 1 if this object's implementation does not reside in the local address space.

### Note

If the object is in the same address space where the method is invoked, `_is_local()` returns 1.

## Checking for non-existent objects

You can use the `_non_existent()` member function to determine if the object implementation associated with an object reference still exists. This method actually "pings" the object to determine if it still exists and returns 1 if it does not exist.

## Narrowing object references

The process of converting an object reference's type from a general super-type to a more specific sub-type is called *narrowing*.

### Note

The `_narrow()` member function may construct a new C++ object and returns a pointer to that object. When you no longer need the object, you must release the object reference returned by `_narrow()`.

VisiBroker-RT for C++ maintains a typegraph for each object interface so that narrowing can be accomplished by using the object's `_narrow()` method. If the `narrow` member function determines it is not possible to narrow an object to the type you request, it will return `NULL`.

**Code example 49** Narrow method generated for the AccountManager

```
Account *acct;
Account *acct2;
Objec *obj;

acct = Account::_bind();
obj = (CORBA::Object *)acct;
acct2 = Account::_narrow(obj);
```

## Widening object references

Converting an object reference's type to a super-type is called *widening*. [Code example 50](#) shows an example of widening an `Account` pointer to an `Object` pointer. The pointer `acct` can be cast as an `Object` pointer because the `Account` class inherits from the `Object` class.

**Code example 50** Widening an object reference

```
...
Account *acct;
CORBA::Object *obj;
acct = Account::_bind();
obj= (CORBA::Object *)acct;
...
```

## Using Quality of Service

Quality of Service (QoS) utilizes policies to define and manage the connection between your client applications and the servers to which they connect.

### Understanding Quality of Service

Quality of Service policy management is performed through operations accessible in the following contexts:

- ORB level policies are handled by a locality constrained `PolicyManager`, through which you can set Policies and view the current `Policy` overrides. Policies set at the ORB level override system defaults.
- Thread level policies are set through `PolicyCurrent`, which contains operations for viewing and setting `Policy` overrides at the thread level. Policies set at the thread level override system defaults and values set at the ORB level.
- Object level policies can be applied by accessing the base `Object` interface's quality of service operations. Policies applied at the Object level override system defaults and values set at the ORB or thread level.

### Policy overrides and effective policies

The effective policy is the policy that would be applied to a request after all applicable policy overrides have been applied. The effective policy is determined by comparing the `Policy` as specified by the IOR with the effective override. The effective `Policy` is the intersection of the values allowed by the effective override and the IOR-specified `Policy`. If the intersection is empty a `INV_POLICY` exception is raised.

## QoS interfaces

The following interfaces are used to get and set QoS policies.

### **CORBA::Object**

`CORBA::Object` contains the following methods used to get the effective policy and get or set the policy override.

- `_get_policy` returns the effective policy for an object reference.
- `_set_policy_override` returns a **new object reference** with the requested list of `Policy` overrides at the object level.
- `_get_client_policy` returns the effective `Policy` for the object reference without doing the intersection with the server-side policies. The effective override is obtained by checking the specified overrides in first the object level, then at the thread level, and finally at the ORB level. If no overrides are specified for the requested `PolicyType` the system default value for `PolicyType` is used.
- `_get_policy_overrides` returns a list of `Policy` overrides of the specified policy types set at the object level. If the specified sequence is empty, all overrides at the object level will be returned. If no `PolicyTypes` are overridden at the object level, an empty sequence is returned.
- `_validate_connection` returns a boolean value based on whether the current effective policies for the object will allow an invocation to be made. If the object reference is not bound, a binding will occur. If the object reference is already bound, but current policy overrides have changed, or the binding is no longer valid, a rebind will be attempted, regardless of the setting of the `RebindPolicy` overrides. A `false` return value occurs if the current effective policies would raise an `INV_POLICY` exception. If the current effective policies are incompatible, a sequence of type `PolicyList` is returned listing the incompatible policies.

## CORBA::PolicyManager

The `PolicyManager` is an interface that provides methods for getting and setting `Policy` overrides for the ORB level.

- `get_policy_overrides` returns a `PolicyList` sequence of all the overridden policies for the requested `PolicyTypes`. If the specified sequence is empty, all `Policy` overrides at the current context level will be returned. If none of the requested `PolicyTypes` are overridden at the target `PolicyManager`, an empty sequence is returned.
- `set_policy_overrides` modifies the current set of overrides with the requested list of `Policy` overrides. The first input parameter, `policies`, is a sequence of references to `Policy` objects. The second parameter, `set_add`, of type `SetOverrideType` indicates whether these policies should be added onto any other overrides that already exist in the `PolicyManager` using `ADD_OVERRIDE`, or they should be added to a `PolicyManager` that doesn't contain any overrides using `SET_OVERRIDES`. Calling `set_policy_overrides` with an empty sequence of policies and a `SET_OVERRIDES` mode removes all overrides from a `PolicyManager`. Should you attempt to override policies that do not apply to your client, a `NO_PERMISSION` exception will be raised. If the request would cause the specified `PolicyManager` to be in an inconsistent state, no policies are changed or added, and an `CORBA::InvalidPolicies` exception is raised.

## CORBA::PolicyCurrent

The `PolicyCurrent` interface derives from `PolicyManager` without adding new methods. It provides access to the policies overridden at the thread level. A reference to a thread's `PolicyCurrent` is obtained by invoking `resolve_initial_references` and specifying an identifier of "PolicyCurrent."

## Messaging::RebindPolicy

`RebindPolicy` reads in a value of type `Messaging::RebindMode` to define the behavior of the client when rebinding. `RebindPolicies` are set only on the client side. It can have one of six values that determines the behavior in the case of a disconnection, an object forwarding request, or an object failure. The currently supported values are:

- `Messaging::TRANSPARENT` allows the ORB to silently handle object-forwarding and necessary reconnections during the course of making a remote request.
- `Messaging::NO_REBIND` allows the ORB to silently handle reopening of closed connections while making a remote request, but prevents any transparent object-forwarding that would cause a change in client-visible effective QoS policies. When `RebindMode` is set to `NO_REBIND`, only explicit rebind is allowed.
- `Messaging::NO_RECONNECT` prevents the ORB from silently handling object-forwards or the reopening of closed connections. You must explicitly rebind and reconnect when `RebindMode` is set to `NO_RECONNECT`.
- `QoSExt::VB_TRANSPARENT` is the default policy. It extends the functionality of `TRANSPARENT` by allowing transparent rebinding with both implicit and explicit binding. `VB_TRANSPARENT` is designed to be compatible with the object failover implementation in VisiBroker-RT for C++ 3.x.
- `QoSExt::VB_NOTIFY_REBIND` throws an exception if a rebind is necessary. The client catches this exception, and binds on the second invocation.

### Note

Be aware that if the effective policy for your client is `VB_TRANSPARENT` and your client is working with servers that hold state data, `VB_TRANSPARENT` could connect the client to a new server without the client being aware of the change of server, any state data held by the original server will be lost.

### Note

In the case of `NO_REBIND` or `NO_RECONNECT` the reopening of the closed connection or forwarding may be explicitly allowed by calling `_validate_connection` on the `CORBA::Object` interface.

The following table lists the behavior of the different `RebindMode` types.

**Table 9** RebindMode policies

RebindMode type	Reestablish closed connection to the same object?	Allow object forwarding?	Object failover? <sup>1</sup>
<code>NO_RECONNECT</code>	No, throws <code>REBIND</code> exception.	No, throws <code>REBIND</code> exception.	No
<code>NO_REBIND</code>	Yes	Yes, if QoS policies match	No
<code>TRANSPARENT</code>	Yes	Yes	No
<code>VB_NOTIFY_REBIND</code>	Yes	Yes	Yes. <code>VB_NOTIFY_REBIND</code> throws an exception after failure detection, and then tries a failover on subsequent requests.
<code>VB_TRANSPARENT</code>	Yes	Yes	Yes, transparently

<sup>1</sup>The appropriate CORBA exception will be thrown in the case of a communication problem or an object failure.

For more information on QoS policies and types, see the VisiBroker-RT for C++ *Reference Guide* and the OMG Messaging specification. Our QoS implementation is based on the OMG document *orbos/98-05-05*.

## **Messaging::RelativeRequestTimeoutPolicy**

`RelativeRequestTimeoutPolicy` is a local object (i.e. locality constrained) derived from **CORBA::Policy**. It is used to indicate the relative amount of time for which a Request may be delivered. After this amount of time the Request is cancelled. This policy is applied to both synchronous and asynchronous invocations. If asynchronous invocation is used, this policy only limits the amount of time during which the request may be processed. Assuming the request completes within the specified timeout, the reply will never be discarded due to timeout.

When instances of **RelativeRequestTimeoutPolicy** are created, a value of type **TimeBase::TimeT** is passed to **CORBA::ORB::create\_policy**. The value specified is the number of 100 nanoseconds which the client application will wait for a request to be delivered to the Server implementation. If the time-out period expires before the message is delivered to the Server implementation, a `::CORBA::NO_RESPONSE` exception is raised.

If a `RelativeRequestTimeoutPolicy` is not specified, `RelativeRequestTimeout` is set to 0, indicating that your client program wishes to block indefinitely.

This policy is only applicable as a client-side override.

## **Messaging::RelativeRoundtripTimeoutPolicy**

`RelativeRoundtripTimeoutPolicy` is a local object (i.e. locality constrained) derived from **CORBA::Policy**. It is used to indicate the relative amount of time for which a Request or its corresponding Reply may be delivered. After this amount of time the Request is cancelled (if a response has not yet been received from the target) or the Reply is discarded (if the Request had already been delivered and a Reply returned from the target). This policy is applied to both synchronous and asynchronous invocations.

When instances of **RelativeRoundtripTimeoutPolicy** are created, a value of type **TimeBase::TimeT** is passed to **CORBA::ORB::create\_policy**. The value specified is the number of 100 nanoseconds which the client application will wait for a request and its corresponding reply to be received. If the time-out period expires before the invocation is completed (i.e. reply received by the ORB), a `CORBA::NO_RESPONSE` exception is raised.

If a `RelativeRoundtripTimeoutPolicy` is not specified, `RelativeRoundtripTimeout` is set to 0, indicating that your client program wishes to block indefinitely.

This policy is only applicable as a client-side override.

## **QoSExt::RelativeConnectionTimeoutPolicy**

`RelativeConnectionTimeoutPolicy` is a local object (i.e. locality constrained) derived from **CORBA::Policy**. It is used to indicate the relative amount of time after which an attempt to connect to the server ORB using one of the available communication endpoints is aborted. This policy is applied to both synchronous and asynchronous invocations.

When instances of **RelativeConnectionTimeoutPolicy** are created, a value of type **TimeBase::TimeT** is passed to **CORBA::ORB::create\_policy**. The value specified is the number of 100 nanoseconds which the client application will wait for a connection to be established. If the time-out

period expires before the connection to the server ORB is established, a `CORBA::TIMEOUT` exception is raised.

If a `RelativeConnectionTimeoutPolicy` is not specified, `RelativeConnectionTimeoutPolicy` is set to 0 seconds, indicating that your client program wishes to block indefinitely.

This policy is only applicable as a client-side override.

## QoSExt::DeferBindPolicy

The `DeferBindPolicy` determines if the ORB will attempt to contact the remote object when it is first created, or to delay this contact until the first invocation is made. The possible values of `DeferBindPolicy` are `TRUE` and `FALSE`. If `DeferBindPolicy` is set to `TRUE`, all binds will be deferred until the first invocation using that Client proxy. **The default value is `FALSE`.**

If you create a client object, and `DeferBindPolicy` is set to `true`, you may delay the server startup until the first invocation. This option existed with prior versions of VisiBroker-RT for C++ as a bind option that could be specified as a parameter to the `_bind` method.

## QoSExt::SmartBindPolicy

`SmartBindPolicy` is a local object (i.e. locality constrained) derived from **`CORBA::Policy`**. It is used to control the VisiBroker SmartBinding optimization. The currently supported values are:

- `QoSExt::SMARTBIND_OFF`

When `SmartBindPolicy` is set to `QoSExt::SMARTBIND_OFF`, communications between the VisiBroker client and server will use the local IP LOOPBACK interface, thereby ignoring any optimization. This option existed with prior versions of VisiBroker-RT for C++ as a bind option that could be specified as a parameter to the `_bind` method.

- `QoSExt::SMARTBIND_POA_TRANSPARENT`

When `SmartBindPolicy` is set to `QoSExt::SMARTBIND_POA_TRANSPARENT`, all co-located invocations (i.e. between VisiBroker clients and servants in the same address space) are optimized. When using this policy value all POA policies and states applicable to that CORBA Server are honored.

- `QoSExt::SMARTBIND_CACHED`

When `SmartBindPolicy` is set to `QoSExt::SMARTBIND_CACHED`, all co-located invocations (i.e. between VisiBroker clients and servants in the same address space) are optimized. Using this policy value the servant pointer is cached during the initial invocation to the CORBA object. Subsequent requests to this server will use this cached pointer, thereby ignoring all POA policies and POA states. This policy value provides the highest level of optimization.

This cached pointer to the servant can be updated by calling `_bind`. This may be useful in cases where the servant goes away and the client needs to update its cached pointer to a new instance of the servant. In that case, the client application can catch the generated CORBA exception and call `_bind` again to update the cached pointer.

If the POA that the servant is activated or is created with a value other than `USE_ACTIVE_OBJECT_MAP_ONLY` for the `RequestProcessingPolicy`, the `SMARTBIND_CACHE` behavior reverts to

`QoSExt::SMARTBIND_POA_TRANSPARENT`.

The default value for this policy is `QoSExt::SMARTBIND_CACHED`. This policy applies to both synchronous and asynchronous invocations.

This policy is only applicable as a client-side override.

## QoS exceptions

- `CORBA::INV_POLICY` is raised when there is an incompatibility between `Policy` overrides.
- `CORBA::REBIND` is raised when the `RebindPolicy` has a value of `NO_REBIND`, `NO_RECONNECT`, or `VB_NOTIFY_REBIND` and an invocation on a bound object references results in an object-forward or location-forward message.
- `CORBA::PolicyError` is raised when the requested `Policy` is not supported.
- `CORBA::InvalidPolicies` can be raised when an operation is passed a `PolicyList` sequence. The exception body contains the policies from the sequence that are not valid, either because the policies are already overridden within the current scope, or are not valid in conjunction with other requested policies.



# Part 4

## Configuration and Management

### **In this part**

This part contains the following chapters:

<a href="#">Using the VisiBroker-RT for C++ Console</a>	page 103
<a href="#">Setting Properties</a>	page 115



# Using the VisiBroker-RT for C++ Console

*VisiBroker-RT for C++ provides a graphical user interface, the VisiBroker Console which functions as the main control point for the server. The VisiBroker Console lets you view servers on the network, change server configurations, and manage the services and tools that enable you to build, deploy, and manage CORBA-based applications.*

*This section provides an overview of how to use the VisiBroker Console to start and stop a server, change server configurations, and manage top-level services.*

## NOTE

The library libsvmgr.o is required when building a VisiBrokerRT60 application to support communicating with the VisiBroker Console. For a description of all the libraries provided by the VisiBroker-RT for C++ product, see ["Step 6: Integrating VisiBroker-RT with Tornado/VxWorks"](#).

## What is the VisiBroker Console?

The VisiBroker Console is a tool that allows you to view, configure, and monitor the Borland Enterprise Server ORB Services in a graphical interface. In particular, you can use the ORB Services browsers to manage object servers, control the configuration of gatekeepers, browse the interface repository, edit naming contexts, look up object instances, and view the OADs on your network.

The design of the VisiBroker Console is similar to the graphical interfaces of the Borland Enterprise Server Console product.

The VisiBroker Console provides browser support and is divided as follows into the following areas, which correspond to the ORB Services that it supports:

- Location Service
- Naming Services
- Interface Repositories<sup>1</sup>
- Implementation Repositories<sup>1</sup>
- Server Managers
- Gatekeepers<sup>1</sup>
- Integrated Transaction Services <sup>1</sup>

<sup>1</sup>Note that the VisiBroker-RT for C++ for Tornado console can be used to view and manage this service on the network; however the service itself is not available on a VisiBroker-RT for C++ for Tornado system.

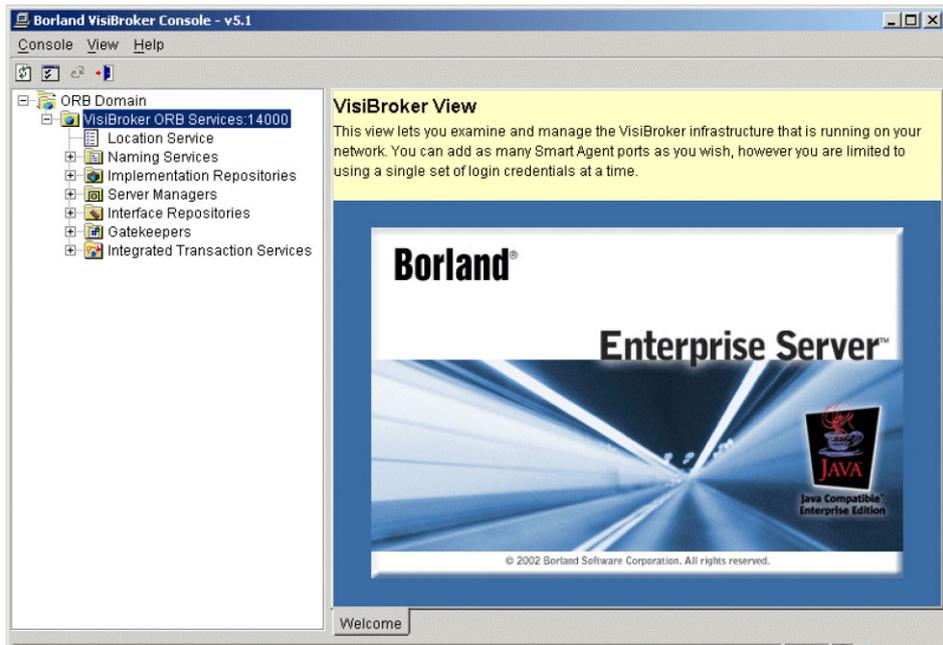
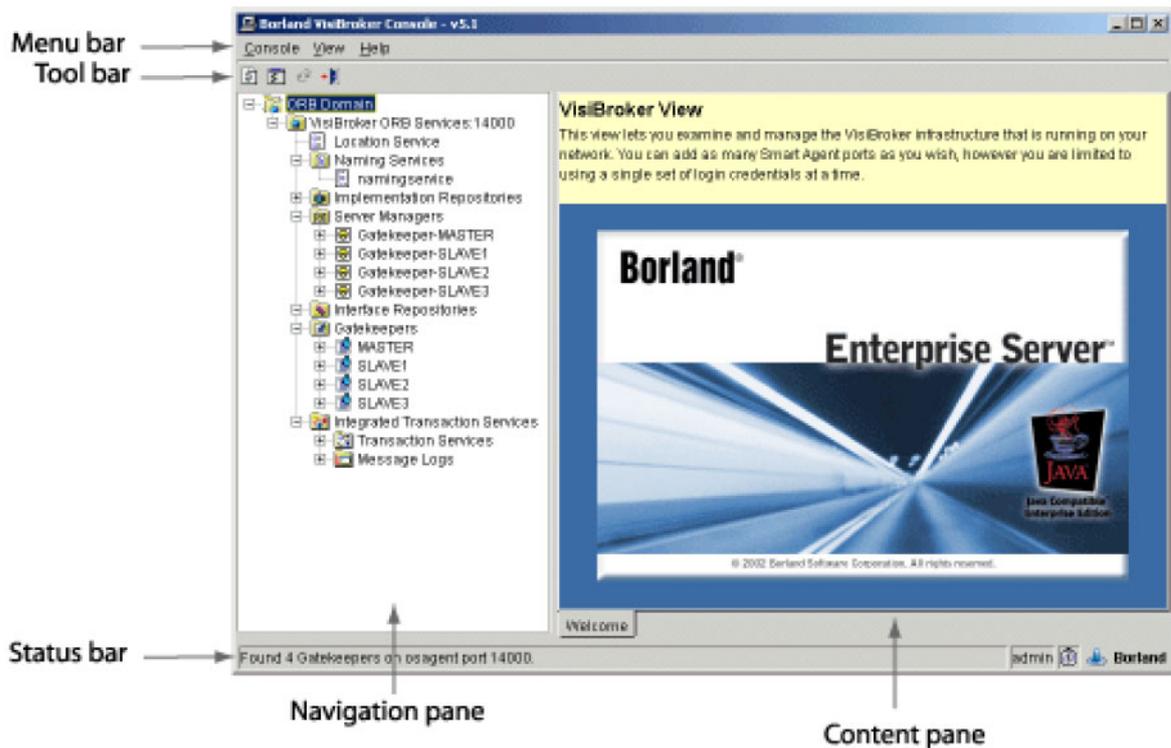


Figure 13 The VisiBroker Console

**Updated screen illustration, please? I presume the Borland branding is going**

## Navigating the VisiBroker Console

The VisiBroker Console has a typical Explorer-style user interface with elements such as menus, tools, and status bars; a navigation pane on the left side of the viewing area; and a content pane on the right side. You choose options from pull-down or context (right-click) menus to perform common functions; select specific ORB Services from the navigation pane; or perform tasks in the content pane (work area) related to the ORB Service that you select.



**Figure 14** Elements of the VisiBroker Console

***And updated version of this one too, please***

The VisiBroker Console's main window consists of the following elements that help you complete the tasks related to the specific ORB Service:

- Menu bar
- Toolbar
- Status bar
- Pull down or context menus
- Navigation pane
- Content pane

### **Menu bar**

The menu bar is located at the top of the VisiBroker Console's main window. The menu bar provides you with some of the common navigational and management options in the VisiBroker Console.

### **Toolbar**

The toolbar is located at the top of the VisiBroker Console main window, just under the menu bar. The toolbar lets you perform some of the VisiBroker Console functions with a single click of the mouse. Toolbar functions are dimmed when their functions are not available in a specific context.

## Status bar

The status bar is located at the bottom of the main window of the VisiBroker Console. The status bar displays information about the status of your actions and also displays any warning messages for the current session.

## Pull down or context menus

The pull down menus are located in the menu bar area, at the top of the VisiBroker Console's main window. The context menus display when you right-click an item on the VisiBroker Console. You can perform many common functions by either using pull-down or context (right-click) menus. In some cases, you have the option to use either menu to perform the same function.

## Navigation pane

The VisiBroker Console's viewing area is divided into two major parts: the Navigation pane on the left side and the Content pane on the right side.

The Navigation pane shows you a hierarchical tree structure in which you can expand items to navigate to the next level. The hierarchical tree contains folders that represent the ORB Services.

Clicking these folders selects the Service and displays a browser to the right of the tree. Right-clicking provides a menu of possible actions on the folder. Once you click an item, the right side of the panel--the Content pane--shows you information about the item you just selected.

## Content pane

The Content pane contains the content of the item you select in the Navigation pane. Depending on which item you select, different sets of tabs appear at the bottom of the Content pane. Selecting one of these tabs changes the information that appears in the Content pane.

# Supported ORB Services

With the VisiBroker Console, you can view, configure, and monitor the ORB Services. To access the ORB Services, click on a specific service in the navigation pane. The selected ORB Service is displayed in the content pane.

To browse the ORB Services on a particular Smart Agent port, right click on the root node (ORB Domain) of the navigation pane. The Smart Agent port entry dialog will appear. After entering the desired Smart Agent port number, a new VisiBroker ORB Services node will appear under the root node.

The VisiBroker Console supports the following ORB Services:

## Location Service

The Location Service is the interface to the Smart Agent. This browser provides general purpose facilities for locating object instances and displays all instances of an object to which a client can bind. Also, it provides a list of all Smart Agents running on the current port.

For more information about the Location Service, see [“Using the Location Service”](#).

## Naming Services

The Naming Services displays, in a hierarchical format, the contents of the naming services running on your Borland Enterprise Server domain. From here, you can select, navigate, and edit naming contexts and name bindings.

For more information about the Naming Service, see [“Using the Naming Service”](#).

## Interface Repositories

The Interface Repositories browser displays, in a hierarchical format, the contents of the interface repository on your Borland Enterprise Server domain. An interface repository is like a database of CORBA object interface information. The information in an interface repository is equivalent to the information in an IDL file.

For more information about the Interface Repositories, see [“Using Interface Repositories”](#).

### NOTE

The Interface Repository **is NOT available on a VisiBrokerRT60 system**; however the Console may still be used to browse Interface Repositories which may be present on other non-embedded VisiBroker nodes in your network.

## Implementation Repositories

The Implementation Repositories browser shows a list of all object implementations registered with each Object Activation Daemon (OAD).

### NOTE

The Implementation Repository is **NOT** available on a VisiBrokerRT60 system; however the Console may still be used to browse Implementation Repositories which may be present on other non-embedded VisiBroker nodes in your network.

## Server Manager

From within the Server Manager, an object server can publish its own properties. These properties appear in the content pane. The ORB properties are published by default, but each server can hide or rearrange the containers, methods, or properties if it chooses to. The Server Manager allows you to monitor and manage running servers, view the POA hierarchy, and set properties.

## GateKeeper

The GateKeeper displays a list of active GateKeeper instances from which you select, to browse and configure their properties. The selected GateKeeper instance displays in the content pane.

For more information on the GateKeeper, see the Borland Enterprise Server *VisiBroker GateKeeper Guide*.

#### NOTE

The Gatekeeper **is NOT available on a VisiBrokerRT60 system**; however the Console may still be used to browse Gatekeepers which may be present on other non-embedded VisiBroker nodes in your network.

## Integrated Transaction Services

The Integrated Transaction Services (ITS) provides a complex solution for distributed transactional CORBA applications. Implemented on top of the VisiBroker ORB, ITS simplifies the complexity of distributed transactions by providing an essential set of services, which includes a transaction service, recovery and logging, integration with database and legacy systems, and administration facilities within one, integrated architecture..

For more information on the Integrated Transaction Services (ITS), see the *VisiBroker Integrated Transaction Services (ITS) Programmer's Guide* by going to [http:// info.borland.com/techpubs/its/its12/](http://info.borland.com/techpubs/its/its12/).

#### NOTE

Integrated Transaction Service is **NOT** available on a VisiBroker-RT 60 system; however the Console may still be used to browse ITSs which may be present on other non-embedded VisiBroker nodes in your network.

## Starting the VisiBroker Console

To start the VisiBroker Console, use one of the following methods make sure that the following environment variables have been set:

- VBROKERDIR set to `<install_location>/VisiBrokerRT60`.
- OSAGENT\_PORT set to the port number where the osagent is running. Use one of the following methods to start the Console:

#### Windows

Run the `vbconsole` file from the `<install_location>\VisiBrokerRT60\bin` directory

#### UNIX

Run `vbconsole.sh` from the `<install_location>/VisiBrokerRT60/bin` directory

#### Note

To recognize the console command, your path system variable must include the Console bin directory (`<install_location>/VisiBrokerRT60/bin`), or you can enter the path explicitly.

Once the Console starts, the preferences that were configured during installation take effect. If you have problems, please check the path and classpath settings.

- 1 When the Console login window appears, enter your user name, password, and server realm (**default User Name=> admin, Default Password=> admin**). After logging in to the Console, select VisiBroker from the left most button bar of the Console to launch the VisiBroker Console.

enterprise Server, you need to enter that information using the Console before you can use it.

**!!!The half-sentence above is in the original pdf. I'm not sure how much material might be missing, but since there is a step numbered 1 with no corresponding 2 (or later numbers), it might be quite a bit. Any ideas, developers?**

### Note

The library `libsrvmgr.o` is required when building a VisiBrokerRT60 application to support communicating with the VisiBroker Console. For a description of all the libraries provided by the VisiBroker-RT for C++ product, see [Step 6: Integrating VisiBroker-RT with Tornado/VxWorks](#).

## VisiBroker Console main menu

The VisiBroker Console provides the following main menu items:

- Console menu
- View menu
- Help menu

### Console menu

The following table describes the commands on the VisiBroker Console menu.

**Table 10** VisiBroker Console menu commands

Select...	To...
<b>Refresh</b>	manually update server state information shown in the VisiBroker Console.
<b>Preferences...</b>	open the Preferences dialog box to set VisiBroker Console and VisiBroker Server configurations settings. See <a href="#">"Setting the VisiBroker Console preferences"</a> .
<b>Login</b>	log on to the console with your user name, password, and realm credentials.
<b>Logout</b>	log out of the console so that you can log on with new user name, password, and realm credentials.
<b>Exit</b>	dismiss the VisiBroker Console.

### View menu

The following table describes the commands on the View menu.

**Table 11** View menu commands

Select...	To...
<b>Messages</b>	show or hide the errors window.
<b>Tool bar</b>	show or hide the tool bar at the top of the Console window.
<b>Status bar</b>	show or hide the status bar at the bottom of the Console window.

## Help menu

The following table describes the commands on the Help menu.

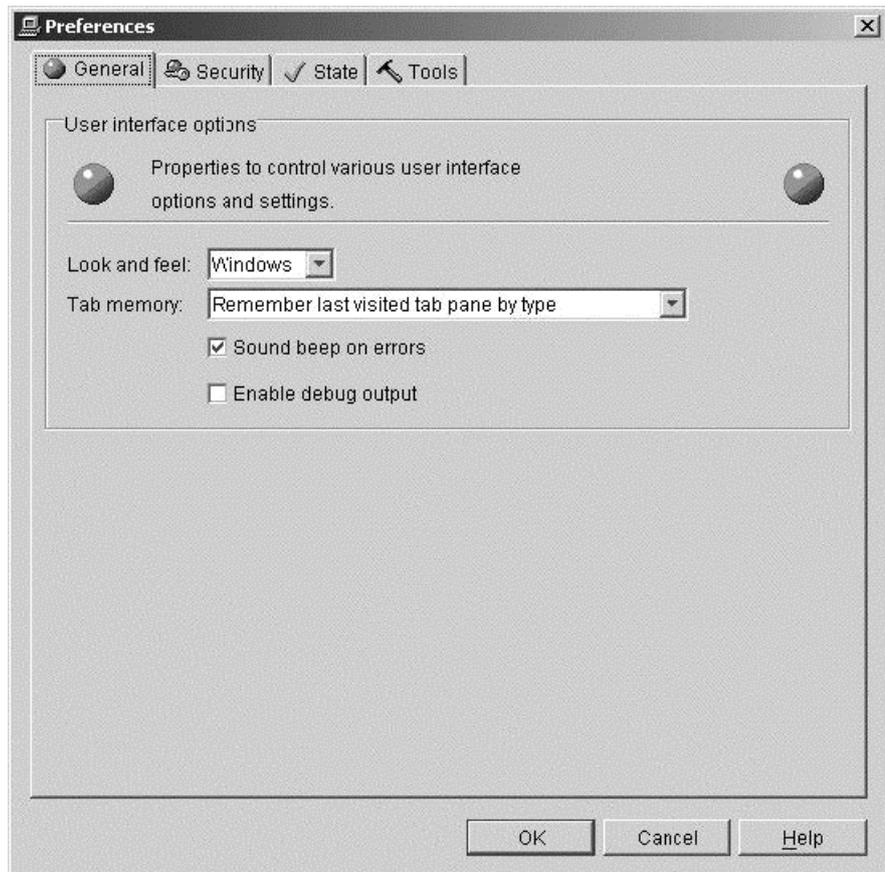
**OK, the following table definitely needs checking by developers and updating! Are we linking to these documents, where I have replaced "Borland" by "Micro Focus" or "VisiBroker-RT" is it correct, etc etc!!**

Table 12 Help menu commands

Select...	To...
<b>Installation Guide</b>	get online help on installing VisiBroker-RT
<b>User's Guide</b>	get online help on using the Console and other tools including the DDEditor and the Application Assembly Tool.
<b>Developer's Guide</b>	get online help on packaging, deployment, and management of distributed object-based applications.
<b>Deployment Descriptor Editor (DDE)</b>	get online help on using the DDEditor.
<b>VisiBroker Developer's Guide</b>	get online help on how to develop VisiBroker applications in Java or C++ and the configuration and management of the VisiBroker ORB.
<b>VisiBroker Programmer's Reference</b>	get online help on the classes and interfaces supplied with VisiBroker for Java and C++ and on using the programming tools and command-line options
<b>VisiBroker GateKeeper Guide</b>	get online help on the VisiBroker GateKeeper that enables VisiBroker clients to communicate with servers across networks while conforming to the security restrictions imposed by web browsers and firewalls.
<b>VisiNotify Guide</b>	get online on using the VisiNotify notification message framework.
<b>Micro Focus Home Page</b>	Access the Micro Focus web site.
<b>News Group</b>	Access the Micro Focus Newsgroups web site.
<b>About</b>	open a dialog box containing the following tabs: <ul style="list-style-type: none"><li>• <b>About:</b> Shows the Borland Enterprise Server version number and copyright information.</li><li>• <b>General System Information:</b> Shows various system configuration settings that Borland Enterprise Server has detected such as the operating system, Java version, Java vendor, Java Compiler, and so forth.</li><li>• <b>Java Properties:</b> Shows the Java virtual machine property settings in use by VisiBroker-RT.</li></ul>

## Setting the VisiBroker Console preferences

VisiBroker Console preferences enable you to specify configuration, operation, and appearance settings used by the VisiBroker Console such as the Smart Agent port, the default polling interval for performance information displayed, and so forth.



**Figure 15** Setting Console Preferences

To set Console preferences:

- 1 Start the VisiBroker Console and choose Preferences from the Console menu. A dialog box appears with a list of preferences grouped into the following tabs:
  - a General menu
  - b Security tab
  - c State tab
  - d Tools tab
- 2 Navigate through the tabs and select the preferences as desired. (If you want to restore the settings shown on a particular tab to the values last saved, click Reset.)
- 3 When you have finished making your selections, click OK.

The following sections provide details on each of the Preferences tabs.

## General tab

This tab provides the following options:

- **Look and feel:** Sets the display format and behavior of the Console windows. The available options are: Metal, Windows or CDE/Motif.

- **Tab Memory:** Specifies the view state information the Console uses. The following options are available:
  - **Don't Remember last visited tab pane:** Tells the Console to open each node in the tree with the General tab displayed on the right.
  - **Remember last visited tab pane by type:** Tells the Console to open a node on the same type of tab (on a similar node) that was most recently viewed. For example, if the Logs tab is currently in view and you click on another node that has a Logs tab, the Console first displays the Logs tab for that node.
  - **Remember last visited tab pane by type and name:** Tells the Console to open a node that had been expanded earlier in the Console session to the tab that was last in view when that node was selected.
- **Sound beep on errors:** If checked, the Console sounds an alarm when an error occurs.
- **Enable debug output:** Tells the Console to report debugging information in the Errors pane at the bottom of the Console.

## Security tab

This tab provides the following options:

- **Default Realm:** Specifies the name of the authentication realm used by the VisiBroker Console to interact with each Borland Enterprise Server.
- **Default User:** Specifies the user name used by the VisiBroker Console to interact with each Borland Enterprise Server.
- **Enable Security:** Determines how the VisiBroker Console handles security:
  - When checked, enables the VisiBroker Console to communicate with a server regardless if it has security enabled or not. When the VisiBroker Console receives a request from a server with security enabled, however, it must first pass the user's login credentials (realm, username, and password) to that server for authentication before it can access services on that server.
  - When this box is not checked, the VisiBroker Console will communicate only with servers that do not have security enabled.

## State tab

This tab provides the following options:

- **Enable polling for events:** When checked, tells the Console to automatically update information displayed about the state of the server and services (such as running, stopped, and so forth). The following settings determine the time intervals (in milliseconds) of how often the Console checks to verify the state of Borland Enterprise Servers, and they specify how often the state of services are updated in the tree in the Console Servers View:
  - **Background polling interval:** Determines how frequently the Console checks the state of the server when no user interaction is initiated.
  - **Foreground polling interval:** Determines how frequently the Console checks server state of the server when the user performs any action that causes the user name server state to change, such as stopping, refreshing, or restarting a server.

- **Number of foreground cycles:** Determines how many times within the specified Foreground Polling Interval that the Console check the server state.
- **Enable background refreshes:** When checked, tells the Console to automatically update information displayed about the changes in the navigation tree, such as when a server, service, or module is added or removed. Clear this check box to reduce the processing overhead used by Console polling activity. If this box is unchecked, however, the Console will not display changes in the navigation tree until the box is checked, or until you either restart the Console, or log out and log back in to the Console.
- **Refreshes every:** Determines (in milliseconds) how frequently the Console checks and refreshes the display of the state of the navigation tree.
- **State Legend:** Shows the icons used by the Console to represent the various server states.

## Tools tab

Use this tab to specify an absolute (fully qualified) path location in which OptimizeIt Profiler is installed on the machine on which the Console is running. Enter a path or click Browse to locate the local OptimizeIt installation directory. If you installed the OptimizeIt Suite, be sure to select the second level OptimizeIt folder (the folder that contains the lib directory, as well as other directories).

### Note

If you are using the Console to manage a remote server, you must also install OptimizeIt on the machine on which the server is running.

For more information about configuring OptimizeIt, see the Servers view in the *Borland Enterprise Server User's Guide*.

***I presume this BES User Guide is no longer available: do we actually use OptimizeIt any more, even?***



# Setting Properties

*This chapter describes how to set VisiBroker properties that can be used to configure many aspects of VisiBroker's behavior.*

## Overview

VisiBroker has number of properties that can be used to configure its behavior. For example, `vbroker.agent.debug` directs the ORB to turn on output of debugging information for all communication with the Smart Agent. Each property has a predetermined data type, either string, unsigned long or boolean, and one or more possible values. For example, `vbroker.agent.enableLocator=false` disables lookups to the smart agent.

Properties can be set:

- prior to starting the application, via environment variables (only a few properties may be set in this way)
- when starting applications - in a Property Table or as a command-line argument,
- after `ORB_init()` via the Property Manager interface.

The order in which these properties take precedence (starting with the highest precedence) is properties specified via:

- 1 the Property Manager interface
- 2 individually at `ORB_init`
- 3 a Property Table passed in at `ORB_init`
- 4 Environment variables
- 5 ORB defaults.

The properties data specified during `ORB_init()`, (i.e. item 3 above) are not referenced again after those properties have been copied into the memory of the Property Manager.

The following sections describe how to use each of the above methods for specifying properties and their values.

## Setting Properties Through the Property Manager Interface

The following code sample shows how to set properties using the Property Manager interface.

**Code example 51** Using the Property Manager interface to set properties after `ORB_init()`

```
...  
  
void do_corba(void)  
{  
    /*-----*/  
    /* ORB_init options can be specified in two ways. */  
    /* 1) By calling start_corba and specifying the */  
    /* ORB initialization string */  
    /* (e.g. start_corba("-ORBagentport 19000") */  
    /* 2) Programatically by specifying the */  
    /* ORB_initialization_options in the */  
}
```

```

/* default_argc and default_argv variables below. */
/* */
/* PLEASE NOTE THAT THE OPTIONS PASSED IN VIA start_corba */
/* OVERRIDE THE OPTIONS THAT ARE SET PROGRAMATICALLY. */
/*-----*/

int default_argc = 2;
char *default_argv[] = {"-ORBagentport", OSAGENT_PORT};
char **new_argv;
int new_argc = VISUtil::stringToArgv(&new_argv, default_argv,
default_argc, ORB_options_string);

/*-----*/
/* Call ORB_init */
/*-----*/
VISTRY
{
    // Initialize the ORB
    orb = CORBA::ORB_init(new_argc, new_argv);
VISUtil::freeArgv(new_argc, & new_argv);
}
VISCATCH(CORBA::Exception,e)
{
    //Handle exception here
}
VISEND_CATCH
// Get the property manager; notice the value returned is not
// placed into a 'var' type.
VISPropertyManager_ptr pm = orb->getPropertyManager();

VISTRY
    pm->addProperty("vbroker.se.mySe.scms", "scm1");

    pm->
addProperty("vbroker.se.mySe.scm.scm1.manager.connectionMax",
100UL);

    pm->
addProperty("vbroker.se.mySe.scm.scm1.manager.connectionMaxIdle",
300UL);
    pm->addProperty("vbroker.se.mySe.scm.scm1.listener.type",
"IIOp");
    pm->addProperty("vbroker.se.mySe.scm.scm1.listener.port",
1042UL);
    pm->
addProperty("vbroker.se.mySe.scm.scm1.listener.proxyPort",
0UL);
}
VISCATCH(CORBA::Exception,e)
{
    //Handle exception here
}
VISEND_CATCH

```

## Environment variables

The following table lists the environment variables that are the equivalent of some property names.

Property name	Environment variable
vbroker.agent.port	OSAGENT_PORT
vbroker.orb.clientPort	OSAGENT_CLIENT_HANDLER_UDP_PORT
vbroker.agent.localFile	OSAGENT_LOCAL_FILE
vbroker.agent.addr	OSAGENT_ADDR
vbroker.agent.addrFile	OSAGENT_ADD_RFILE

## Note

For information on setting VisiBroker-RT for C++ environment variables, see the chapter "Configuring VisiBroker-RT for C++" in the *VisiBroker-RT for C++ Installation Guide*.

## Setting Properties Through the Command-Line

Any property can be set through command-line arguments, added to the argument list passed into `ORB_init()`.

**Code example 52** Setting properties from the command-line

Tornado

```
->start_corba "-Dvbroker.agent.port=1024"
```

Properties set through the command-line override properties in the properties Table of the same name.

## Setting Properties Through a Property Table

A Property Table is a list of property entries, with the following format:

```
property_name=value
```

The ORB has a predefined set of property names available for use. These names are case-insensitive.

There are only three property data types.

- String
- Unsigned long
- Boolean

If the string value is null, you can enter "null" as the property value.

**Code example 53** Setting a null value

```
vbroker.repository.name=null
```

If the value is boolean, enter `true` or `false`.

**Code example 54** Setting a boolean value

```
vbroker.agent.enableLocator=true
```

To use your properties, place them in a Property Table and reference the Table through the following command-line argument:

```
-ORBpropTable=tableName
```

[Code example 55](#) illustrates the steps involved in setting properties by specifying a Property Table as a command line argument to `ORB_init()`.

**Query to developers: In the old manual the comment in the code below reads "Define three manager property values". As far as I can see it only defines two (though with a blank line between the two). I've fudged it by removing the word "three", but can you tell if there is anything missing here?**

**Code example 55** Using a Property Table to set properties at `ORB_init()`

```
void do_corba(void)
{
    // VISPropertyTable defining VisiBroker Properties required
    // for Server Engine configuration. Note that the array of
    // property strings and the VISPropertyTable object can be
    // destructed any time after the ORB_init that uses them.

    // Get the property manager; notice the value returned
    // is not placed into a 'var' type.
    const char * my_properties[] =
```

```

{
"vbroker.se.myServerEngine.scms=scml",

// Define manager property values
"vbroker.se.myServerEngine.scm.scml.manager.connectionMax=100"
"vbroker.se.myServerEngine.scm.scml.manager.connectionMaxIdle=
300",

// Define three listener property values
"vbroker.se.myServerEngine.scm.scml.listener.type=IIOP",
"vbroker.se.myServerEngine.scm.scml.listener.port=1042",
"vbroker.se.myServerEngine.scm.scml.listener.proxyPort=0",
NULL
};

VISPropertyTable property_table("my_properties",
my_properties);

cout << "Initialize the server" << endl; int argc = 5;
char *argv[] = {"DO_CORBA", "-ORBagentport", OSAGENT_PORT,
               "-ORBpropTable", "my_properties"};

/*-----*/
/* Call ORB_init */
/*-----*/
VISTRY
{
    // Initialize the ORB
    orb = CORBA::ORB_init(argc, argv);
...

```

## ORB Default Properties

If a property value is not specified for a given property by any of the above methods, then the ORB default value for that property will be used.

For a list of all VisiBroker-RT for C++ properties and their corresponding default values see Appendix B, "Using VisiBroker properties" in the VisiBroker-RT for C++ Reference Manual.

# Part 5

## Tools and Services

### **In this part**

This part contains the following chapters:

<a href="#">Using the IDL compiler</a>	<a href="#">page 121</a>
<a href="#">Using the Smart Agent</a>	<a href="#">page 129</a>
<a href="#">Using the Location Service</a>	<a href="#">page 145</a>
<a href="#">Using the Naming Service</a>	<a href="#">page 157</a>
<a href="#">Using the Event Service</a>	<a href="#">page 173</a>



# Using the IDL compiler

*This chapter describes how to use the IDL compiler.*

## Introduction to IDL

The Interface Definition Language (IDL) is a *descriptive language* (not a programming language) to describe the interfaces being implemented by the remote objects. Within IDL, you define the name of the interface, the names of each of the attributes and methods, and so forth. Once you've created the IDL file, you can use an IDL compiler to generate the client stub file and the server skeleton file in the C++ programming language.

The OMG has defined specifications for such language mapping. Information about the language mapping is not covered in this manual since VisiBroker-RT for C++ adheres to the specification set forth by OMG. If you need more information about language mapping, see the OMG web site at <https://www.omg.org/>. (The CORBA formal specification can be found at <http://www.omg.org/corba/corbaiop.html>. See “[Bidirectional Communication](#)” for mapping of OMG IDL to C++.)

### **Not sure what address to use in place of the corbainop one above**

Discussions on the IDL can be quite extensive. Since VisiBroker-RT for C++ adheres to the specification defined by OMG, you can visit the OMG site for more information about IDL.

## How the IDL compiler generates code

You use the Interface Definition Language (IDL) to define the object interfaces that client programs may use. The `idl2cpp` compiler uses your interface definition to generate code.

For details on usage syntax for the `idl2cpp` compiler, see the VisiBroker-RT for C++ *Reference Guide*.

## Example IDL specification

Your interface definition defines the name of the object as well as all of the methods the object offers. Each method specifies the parameters that will be passed to the method, their type, and whether they are for input or output or both. [IDL sample 4](#) shows an IDL specification for an object named `example`. The `example` object has only one method, `op1`.

### **IDL sample 4** Example IDL specification

```
// IDL specification for the example object
interface example {
    long op1(in char x, out short y);
};
```

## Looking at code generated for clients

[Code example 56](#) shows how the IDL compiler generates two client files—`example_c.hh` and `example_c.cc`. These two files provide an `example` class that the client uses. By convention, files generated by the IDL compiler always have either a “.cc” or “.hh” suffix to make them easy to

distinguish from files that you create yourself. If you wish, you can alter the convention to produce files with a different suffix. See the VisiBroker-RT for C++ *Reference Guide*.

### Caution

*Do not* modify the contents of the files generated by the IDL compiler.

**Code example 56** example generated class in `example_c.hh` generated file

```
class example : public virtual CORBA_Object {
protected:
    example() {}
    example(const example&) {}
public:
    virtual ~example() {}
    static const CORBA::TypeInfo * _desc();
    virtual const CORBA::TypeInfo * _type_info() const;
    virtual void * _safe_narrow(const CORBA::TypeInfo& ) const;
    static CORBA::Object* _factory();
    example_ptr this();
    static example_ptr _duplicate(example_ptr _obj) { /*... */
}

    static example_ptr _nil() { /*... */ }
    static example_ptr _narrow(CORBA::Object* _obj);
    static example_ptr _clone(example_ptr _obj) { /*... */ }
    static example_ptr _bind(
        const char * _object_name = NULL,
        const char * _host_name = NULL,
        const CORBA::BindOptions* _opt = NULL,
        CORBA::ORB_ptr _orb = NULL);
    static example_ptr _bind(
        const char * _poa_name,
        const CORBA::OctetSequence& _id,
        const char * _host_name = NULL,
        const CORBA::BindOptions* _opt = NULL,
        CORBA::ORB_ptr _orb = NULL);
    virtual CORBA::Long op1(CORBA::Char _x, CORBA::Short_out _y);
};
```

## Methods (stubs) generated by the IDL compiler

[Code example 56](#) shows the `op1` method generated by the IDL compiler, along with several other methods. The `op1` method is called a *stub* because when your client program invokes it, it actually packages the interface request and arguments into a message, sends the message to the object implementation, waits for a response, decodes the response, and returns the results to your program.

Since the `example` class is derived from the `CORBA::Object` class, several inherited methods are available for your use.

## Pointer type <interface name>\_ptr definition

The IDL compiler always provides a pointer type definition. [Code example 57](#) shows the type definition for the `example` class.

**Code example 57** `_ptr` type definition in the `example_c.hh` generated file

```
typedef example *example_ptr;
```

## Automatic memory management <interface name>\_var class

The IDL compiler also generates a class named `example_var`, which you can use instead of an `example_ptr`. The `example_var` class will automatically manage the memory associated with the dynamically allocated object reference. When the `example_var` object is deleted, the object associated with `example_ptr` is released. When an `example_var` object is assigned a new value, the old object reference pointed to by `example_ptr` is released after the assignment takes place. A casting operator is also provided to allow you to assign an `example_var` to a type `example_ptr`.

**Code example 58** `example_var` class in `example_c.hh` generated file

```
class example_var : public CORBA::_var {
...
public:
    static example_ptr _duplicate(example_ptr);
    static void _release(example_ptr);
    example_var();
    example_var(example_ptr);
    example_var(const example_var &);
    ~example_var();
    example_var& operator=(example_ptr);
    example_var& operator=(const example_var& _var) { /*... */
}
    operator example* () const { return _ptr; }
...
};
```

The following table describes the methods in the `_var` class.

**Table 13** Methods in the `_var` class

Method	Description
<code>example_var()</code>	Constructor that initializes the <code>_ptr</code> to NULL.
<code>example_var(example_ptr ptr)</code>	Constructor that creates an object with the <code>_ptr</code> initialized to the argument passed. The <code>var</code> invokes <code>release()</code> on <code>_ptr</code> at the time of destruction. When the <code>_ptr</code> 's reference count reaches 0, that object will be deleted.
<code>example_var(const example_var&amp; var)</code>	Constructor that makes a copy of the object passed as a parameter <code>var</code> and points <code>_ptr</code> to the newly copied object.
<code>~example()</code>	Destructor that invokes <code>_release()</code> once on the object to which <code>_ptr</code> points.
<code>operator=(example_ptr p)</code>	Assignment operator invokes <code>_release()</code> on the object to which <code>_ptr</code> points and then stores <code>p</code> in <code>_ptr</code> .
<code>operator=(const example_ptr p)</code>	Assignment operator invokes <code>_release()</code> on the object to which <code>_ptr</code> points and then stores a <code>_duplicate()</code> of <code>p</code> in <code>_ptr</code> .
<code>example_ptr operator-&gt;()</code>	Returns the <code>_ptr</code> stored in this object. This operator should not be called until this object has been properly initialized.

## Looking at code generated for CORBA server implementations

[Code example 59](#) shows how the IDL compiler generates two server files: `example_s.hh` and `example_s.cc`. These two files provide a `POA_example`

class that the server uses to derive an implementation class. There are two main classes which are generated for a CORBA Object implementation to use when implementing their servants. The PortableServer\_RefCountServantBase and the PortableServer\_ServantBase. Each is described below.

## The PortableServer\_RefCountServantBase class

The POA\_example class is derived from the PortableServer\_RefCountServantBase class. The POA class PortableServer\_RefCountServantBase is a thread-safe reference counting mix-in class which applications can use to obtain thread-safe reference counting for their CORBA objects. This class extends the base POA PortableServer\_ServantBase who provide virtual empty implementations for the \_add\_ref and \_remove\_ref methods. For details on the PortableServer\_ServantBase see [“The PortableServer\\_ServantBase class”](#) below.

### Caution

*Do not* modify the contents of the files generated by the IDL compiler.

**Code example 59** example using the RefCountServantBase class in example\_s.hh generated file

```
class POA_example : public virtual
PortableServer_RefCountServantBase {
protected:
    POA_example() {}
    virtual ~POA_example() {}
public:
    static const CORBA::TypeInfo _skel_info;
    virtual const CORBA::TypeInfo *_type_info() const;
    example_ptr _this();
    virtual void *_safe_narrow(const CORBA::TypeInfo& ) const;
    static POA_example * _narrow(PortableServer_ServantBase
*_obj);
    // The following operations need to be implemented
    virtual CORBA::Long opl(CORBA::Char _x, CORBA::Short_out _y) =
0;
    // Skeleton Operations implemented automatically
    static void _opl(void *_obj, CORBA::MarshalInBuffer &_istrm,
const char *_oper, VISReplyHandler& handler);
};
```

## The PortableServer\_ServantBase class

The POA class PortableServer\_ServantBase provides a base class for servants to inherit from. Unlike the PortableServer\_RefCountServantBase class above, this class provides empty implementations for the \_add\_ref and \_remove\_ref methods. A CORBA Object implementation can inherit from this class and implement its own \_add\_ref and \_remove\_ref methods if it chooses to provide its own reference counting mechanism; otherwise the recommendation when developing applications with VisiBroker-RT for C++ is to use [“The PortableServer\\_RefCountServantBase class”](#).

### Caution

*Do not* modify the contents of the files generated by the IDL compiler.

**Code example 60** example using the ServantBase class in example\_s.hh generated file

```
class POA_example : public virtual PortableServer_ServantBase {
protected:
    POA_example() {}
    virtual ~POA_example() {}
public:
    static const CORBA::TypeInfo _skel_info;
    virtual const CORBA::TypeInfo *_type_info() const;
    example_ptr _this();
    virtual void *_safe_narrow(const CORBA::TypeInfo& ) const;
    static POA_example *_narrow(PortableServer_ServantBase
    * obj);
    // The following operations need to be implemented
    virtual CORBA::Long op1(CORBA::Char _x, CORBA::Short_out _y)
    = 0;
    // Skeleton Operations implemented automatically
    static void _op1(void * obj, CORBA::MarshalInBuffer &_istrm,
    const char *_oper, VISReplyHandler& handler);
};
```

## Methods (skeletons) generated by the IDL compiler

Notice that the `op1` method declared in the IDL specification in [IDL sample 4](#) is generated, along with an `_op1` method. The `POA_example` class declares a pure virtual method named `op1`. The implementation class that is derived from `POA_example` must provide an implementation for this method.

The `POA_example` class is called a *skeleton* and its method (`_op1`) is invoked by the POA when a client request is received. The skeleton's internal method will marshal all the parameters for the request, invoke your `op1` method and then marshal the return parameters or exceptions into a response message. The ORB will then send the response to the client program.

The constructor and destructor are both protected and can only be invoked by inherited members. The constructor accepts an object name so that multiple distinct objects can be instantiated by a server.

## Class template generated by the IDL compiler

In addition to the `POA_example` class, the IDL compiler generates a class template named `_tie_example`. This template can be used if you wish to avoid deriving a class from `POA_example`. Templates can be useful for providing a wrapper class for existing applications that cannot be modified to inherit from a new class. [Code example 61](#) shows the template class generated by the IDL compiler for the `example` class.

**Code example 61** Template class generated for the example class

```
template <class T>
class POA_example_tie : public POA_example {
public:
    POA_example_tie (T& t): _ptr(&t), _poa(NULL),
    _rel((CORBA::Boolean)0) {}
    POA_example_tie (T& t, PortableServer::POA_ptr poa):
    _ptr(&t),
    _poa(PortableServer::_duplicate(poa)), _rel((CORBA::Boolean)0) {}
    POA_example_tie (T *p, CORBA::Boolean release= 1)
    : _ptr(p), _poa(NULL), _rel(release) {}
    POA_example_tie (T *p, PortableServer::POA_ptr poa,
    CORBA::Boolean release =1)
    : _ptr(p), _poa(PortableServer::_duplicate(poa)),
    _rel(release) {}
```

```

virtual ~POA_example_tie() { /*... */ }
T* tied_object() { /*... */ }
void tied_object(T& t) { /*... */ }
void _tied_object(T *p, CORBA::Boolean release=1) { /*...
*/ }

CORBA::Boolean _is_owner() { /*... */ }
void _is_owner(CORBA::Boolean b) { /*... */ }
CORBA::Long op1(CORBA::Char _x, CORBA::Short_out _y) { /*
... */ }
PortableServer::POA_ptr _default_POA() { /*... */ }
};

```

For complete details on using the `_tie` template class, see the chapter “Using the Tie Mechanism”.

You may also generate a `_ptie` template for integrating an object database with your servers.

## Defining interface attributes in the IDL

In addition to operations, an interface specification can also define attributes as part of the interface. By default, all attributes are *read-write* and the IDL compiler will generate two methods—one to set the attribute’s value, and one to get the attribute’s value. You can also specify *read-only* attributes, for which only the reader method is generated.

IDL sample 5 shows an IDL specification that defines two attributes—one read-write and one read-only. Code example 62 shows the operations class generated for the interface declared in the IDL.

**IDL sample 5** IDL specification with two attributes—one read-write and one read-only

```

interface Test {
    attribute long count;
    readonly attribute string name;
};

```

**Code example 62** Code generated for the testOperations interface

```

class test : public virtual CORBA::Object {
...
    // Methods for read-write attribute
    virtual CORBA::Long count();
    virtual void count(CORBA::Long __count);

    // Method for read-only attribute.
    virtual char * name();
...
};

```

## Specifying oneway methods with no return value

IDL allows you to specify operations that have no return value, called *oneway* methods. These operations may only have input parameters. When a *oneway* method is invoked, a request is sent to the server but there is no confirmation from the object implementation that the request was actually received. VisiBroker-RT for C++ uses TCP/IP for connecting clients to servers. This provides reliable delivery of all packets so the client can be sure the request will be delivered to the server, as long as the server remains available. Still, the client has no way of knowing if the request was actually processed by the object implementation itself.

## Note

Oneway operations cannot raise exceptions or return values.

### IDL sample 6 Defining a oneway operation

```
interface oneway_example {
    oneway void set_value(in long val);
};
```

## Specifying an interface in IDL that inherits from another interface

IDL allows you to specify an interface that inherits from another interface. The classes generated by the IDL compiler will reflect the inheritance relationship. All methods, data type definitions, constants and enumerations declared by the parent interface will be visible to the derived interface.

### IDL sample 7 Example of inheritance in an interface specification

```
interface parent {
    void operation1();
};
interface child : parent {
    ...
    long operation2(in short s);
};
```

The following code sample shows the code that is generated from the interface specification shown in the previous IDL sample.

### Code example 63 Code generated from the previous IDL sample

```
...
class parent : public virtual CORBA::Object {
    ...
    void operation1( );
    ...
};

class child : public virtual parent {
    ...
    CORBA::Long operation2(CORBA::Short s);
    ...
};
```



# Using the Smart Agent

*This chapter describes the Smart Agent (`osagent`), which Server programs register with in order for Clients to find object implementations. This chapter explains how to configure your own ORB domain, connect Smart Agents on different local networks, and migrate objects from one host to another.*

## NOTE

The library `libagentsupport.o` is required to support ORB to Smart Agent communications. If a Smart Agent is also required to be started on the VxWorks embedded node, the library `osagent.o` is necessary. For a description of all the libraries provided by the VisiBroker-RT for C++ product, see [“Step 6: Integrating VisiBroker-RT with Tornado/VxWorks”](#).

## What is the Smart Agent?

VisiBroker-RT for C++’s Smart Agent (`osagent`) is a dynamic, distributed directory service that provides facilities used by both client programs and object implementations. A Smart Agent must be started on at least one host within your local network, if the Smart Agent is to be used as the Location Service. When your client program invokes `_bind()` on an object, the Smart Agent is automatically consulted. The Smart Agent locates the specified implementation so that a connection can be established between the client and the implementation. The communication with the Smart Agent is completely transparent to the client program.

If the PERSISTENT policy is set on the POA, and `activate_object_with_id` is used, the Smart Agent registers the object or implementation so that it can be used by client programs. When an object or implementation is deactivated, the Smart Agent removes it from the list of available objects. As with client programs, the communication with the Smart Agent is completely transparent to the object implementation.

## Locating Smart Agents

VisiBroker-RT for C++ locates a Smart Agent for use by a client program or object implementation using a UDP broadcast message. The first Smart Agent to respond is used.

After a Smart Agent has been located, a point-to-point UDP connection is used for sending registration and look-up requests to the Smart Agent. The UDP protocol is used because it consumes fewer network resources than a TCP connection. All registration and locate requests are dynamic, so there are no required configuration files or mappings to maintain.

### Note

Broadcast messages are used only to locate a Smart Agent. All other communication with the Smart Agent makes use of point-to-point communication. See [“Using point-to-point communications”](#) for information on how to override the use of broadcast messages.

## Locating objects through Agent cooperation

When a Smart Agent is started on more than one node in the local network, each Smart Agent will recognize a subset of the objects available and

communicate with other Smart Agents to locate objects it cannot find. If one of the Smart Agent's should terminate unexpectedly, all implementations registered with that Smart Agent discover this event and they will automatically reregister with another available Smart Agent.

## Starting a Smart Agent (osagent)

At least one instance of the Smart Agent should be running on a node in your local network. Local network refers to a subnetwork within which broadcast message can be sent.

The VisiBroker-RT for C++ Smart Agent can be started in one of three ways:

- 1 From a command line of the Development Host
- 2 From a command line on the target system
- 3 Programmatically from within a VisiBrokerRT60 application.

### Starting the Smart Agent on the Development Host

To start the Smart Agent from a Windows or Unix Development Host, make sure that the PATH environment variable has been updated to include the VisiBroker-RT for C++ "bin" directory.

From a Windows host, enter the following command at the DOS prompt:

#### Windows

```
prompt> osagent [options]
```

To start the Smart Agent on a UNIX system, enter the following command.

#### UNIX

```
prompt> osagent &
```

The Development Host osagent command accepts the following command line arguments:

Option	Description
-p <i>UDP_port</i>	Overrides the setting of OSAGENT_PORT and the registry setting.
-v	Turns verbose mode on, which provides information and diagnostic messages during execution.
-help or -?	Prints the help message.
-n, -N	Disables system tray icon on Windows.

The following example of the `osagent` command specifies a particular UDP port:

```
osagent -p 17000
```

### Starting the Smart Agent on the Target System

To start the Smart Agent from a VxWorks target system, make sure that the `osagent` library has been included into the VxWorks target. Either the library `osagent.o` must be linked with the VxWorks image or `osagent_munched.o` must be downloaded to the VxWorks target to provide this support.

To start the Smart Agent on the VxWorks target:

#### Tornado

```
--> startOsagent()
```

Option	Value range	Description
Task Priority	0 - 255	The priority that the Osagent task will run at. If not specified the Osagent task defaults to run at priority 200.
Verbosity	0,1	Value=1 turns verbose mode on, which provides information and diagnostic messages during execution. Default is Verbosity off.
Port	1024-65536	UDP Port which the Osagent Communicates on. Default is 14000.
Logger Priority	0-255	The priority that the VisiBroker Logger Task will be started at, if not already running. If the priority is not specified, the Logger task will run at the priority specified for the osagent thread, or the default Osagent task priority if neither is specified.  This parameter only applies if <code>startOsagent</code> is called before <code>ORB_init</code> has been called, since the call to <code>ORB_init</code> enables forwarding for the Default Logger which includes starting the Forwarder Thread.
Osagent_Local_Table	Pointer to Array	The <code>OSAGENT_LOCAL_TABLE</code> is an array of network interfaces that the Smart Agent should use. Each entry in the <code>OSAGENT_LOCAL_TABLE</code> contains the <code>IP ADDRESS</code> , <code>SUBNET MASK</code> and <code>BROADCAST ADDRESS</code> for a single network interface.  Default is the Primary Network Interface.
Osagent_Addr_Table	Pointer to Array	The <code>OSAGENT_ADDR_TABLE</code> is an array of IP Addresses which the Osagent will use, when attempting to communicate with other Osagents.  The configuration of the table is relatively simple, just create array entries containing the <code>IP ADDRESS</code> of the <code>NODE</code> where the <code>REMOTE</code> Smart Agent is running.
initial_heartbeat_window	Time in Seconds	Default is the Primary Network. Specifies an "initial window size" for the <code>heartbeat_frequency</code> period once an Osagent has been started. After this <code>initial_heartbeat_window</code> period has passed, the rate of the Osagent heartbeat is controlled by the "heartbeat_frequency" parameter below.  Default is 60.

Option	Value range	Description
<code>initial_heartbeat_frequency</code>	Time in Seconds	Specifies the initial rate of the Osagent heartbeat. This heartbeat is used by the Osagent to manage and maintain Osagent to Osagent communications. This parameter will dictate the initial rate at which the heartbeat message is sent, once the Osagent is started. After the period specified by the " <code>initial_heartbeat_window</code> " above, has passed, the rate of the Osagent heartbeat is controlled by the " <code>heartbeat_frequency</code> " parameter below.  Default is 5.
<code>heartbeat_frequency</code>	Time in Seconds	Specifies the rate of the Osagent heartbeat. This heartbeat is used by the Osagent to manage and maintain Osagent to Osagent communications. This parameter will dictate the rate at which the heartbeat message is sent.  Default is 300.

## Starting the Smart Agent Programmatically from a VisiBroker-RT Development Host

The VisiBrokerRT60 Smart Agent can also be started from within a VisiBrokerRT60 application. The library "libosagent" is necessary to use the Smart Agent programmatically on a target system. For a description of all the libraries provided by the VisiBroker-RT for C++ product, see [<~XRef>"Step 6: Integrating VisiBroker-RT with Tornado/VxWorks"](#).

The sample application under `<install-location>/VisiBrokerRT/examples/osagent` demonstrates the Smart Agent programmatic API as well as the usage of libosagent library. The following code section shows the `startOsagent` prototype.

### Code example 64 Starting the Smart Agent

```
startOsagent(
    • unsigned long priority, Osagent task priority (200 is default)
    int verbose = 0,
    int port=-1, (default is 14000)
    short logger_priority = -1 (VisiBroker Logger Task priority),
    OSAGENT_LOCAL_ENTRY*local_table = NULL, (pointer to
    OSAGENT_LOCAL_TABLE)
    OSAGENT_ADDR_ENTRY *addr_table=NULL) (pointer to
    OSAGENT_ADDR_TABLE)
    long initial_heartbeat_window = 60, (Osagent to ORB
    Heartbeat interval)
    long initial_heartbeat_frequency = 5, (Osagent to ORB
    initial Heartbeat frequency)
    long heartbeat_frequency = 300); (Osagent to ORB
    Heartbeat frequency)
```

## Verbose output

### UNIX

On a UNIX and VisiBrokerRT target system, the verbose output for the Smart Agent is sent to stdout.

### Windows

On a Windows system, the verbose output for the Smart Agent is written to a log file stored at `<installation_location>\VisiBrokerRT60\log\osagent.log` or to the directory specified by the `VBROKER_ADM`

environment variable or use OSAGENT\_LOG\_DIR to specify a different directory to put the log.

## Disabling the agent

Communication with the Smart Agent can be disabled in two ways:

- 1 The preferred way is to not use the Osagent support library as part of your application. This is accomplished by not linking or loading the osagent support library (i.e. libagentsupport.o).
- 2 If the Osagent support library is part of your VisiBrokerRT60 application, an alternative to turning off communication with the Osagent is to pass an ORB property at ORB\_init time:

### Code example 65 Turning Off Agent Communication via a ORB property

```
void do_corba(void)
{
    /*-----*/
    /* ORB_init options can be specified in two ways.      */
    /* 1) By calling start_corba and specifying the        */
    /*    ORB initialization string                        */
    /*    (e.g. start_corba("-ORBagentport 19000"))       */
    /* 2) Programatically by specifying the              */
    /*    ORB initialization options in the               */
    /*    default_argc and default_argv variables below.  */
    /*-----*/
    /* PLEASE NOTE THAT THE OPTIONS PASSED IN VIA start_corba*/
    /* OVERRIDE THE OPTIONS THAT ARE SET PROGRAMATICALLY.  */
    /*-----*/

    int default_argc = 1;
    char *default_argv[] = {"-Dvbroker.agent.enableLocator=false"};
    char **new_argv;
    int new_argc = VISUtil::stringToArgv(&new_argv, default_argv,
                                         default_argc, ORB_options_string);

    /*-----*/
    /* Call ORB_init */
    /*-----*/
    VISTRY
    {
        // Initialize the ORB
        orb = CORBA::ORB_init(new_argc, new_argv);
        VISUtil::freeArgv(new_argc, & new_argv);
    }

    ...

    ...
}
```

If you use string-to-object references, a naming service, or pass in a URL reference, the Smart Agent is not required, and support can be either excluded or turned off. If your client uses the `_bind()` method, you must use the Smart Agent.

## Ensuring Agent availability

Starting a Smart Agent on more than one host within the local network allows clients to continue to bind to objects, even if one of the Smart Agents terminates unexpectedly. If a Smart Agent becomes unavailable, all object implementations registered with that Smart Agent will be automatically re-registered with another Smart Agent. If no Smart Agents are running on the local network, object implementations will continue retrying until a new Smart Agent can be contacted.

If a Smart Agent terminates, any connections between a client and an object implementation that were established before the Smart Agent

terminated will continue without interruption. However, any new `_bind()` requests issued by a client will cause a new Smart Agent to be contacted.

No special coding techniques are required to take advantage of these fault-tolerant features. You only need to make sure a Smart Agent is started on one or more hosts on the local network.

## Checking client existence

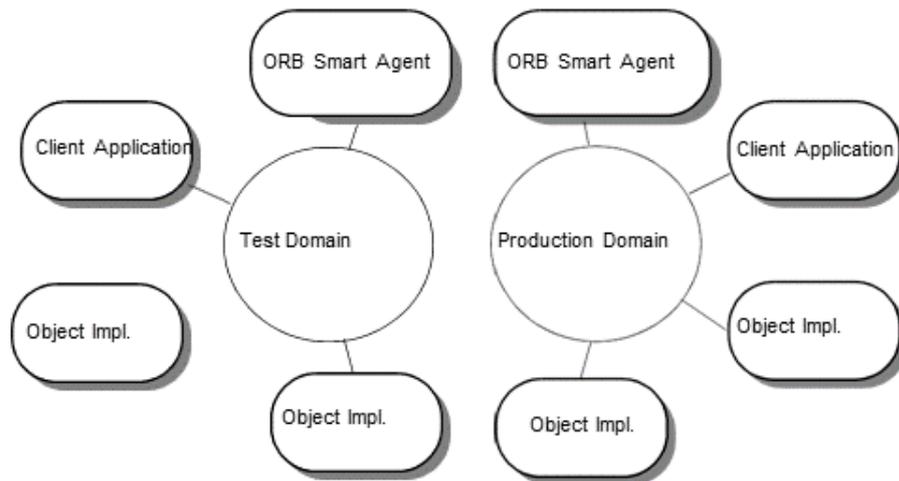
A Smart Agent sends an "Are You Alive" message (often called a *heartbeat* message) to its clients (i.e. each ORB instance it is communicating with) every two minutes to verify that the client ORB is still connected. If the client ORB does not respond, the Smart Agent assumes the client ORB has terminated the connection.

### NOTE

You can not change the interval for polling the client ORB.

## Working within ORB domains

It is often desirable to have two or more separate ORB domains running at the same time. One domain might consist of the production versions of client programs and object implementations while another domain might be made up of test versions of the same clients and objects that have not yet been released for general use. If several developers are working on the same local network, each may want to establish their own ORB domain so that their testing efforts do not conflict with one another.



**Figure 16** Running separate ORB domains simultaneously

VisiBroker-RT for C++ allows you to distinguish between multiple ORB domains on the same network by using a unique UDP port number for the Smart Agents for each domain. By default, the `OSAGENT_PORT` variable is set to 14000. If you wish to use a different port number, check with your system administrator to determine what port numbers are available. To override the default setting, the `OSAGENT_PORT` variable must be set accordingly before running a Smart Agent, an OAD, object implementations, or client programs assigned to that ORB domain.

**Code example 66** Setting the OSAGENT\_PORT environment variable for a UNIX system running csh

```
prompt> setenv OSAGENT_PORT 5678
prompt> osagent &
```

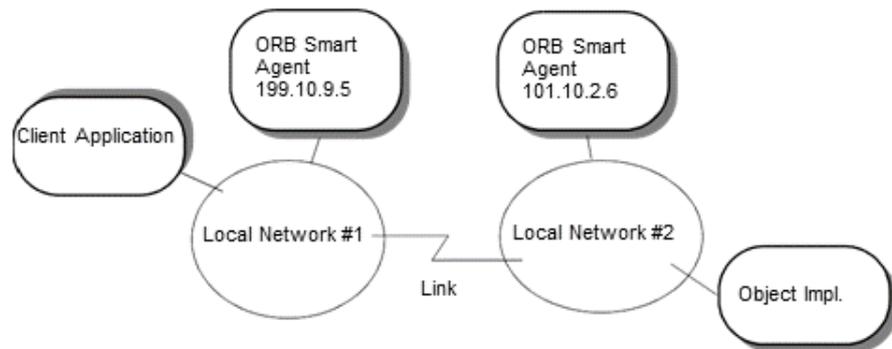
The Smart Agent also uses another port number internally. This port number can be set by using the OSAGENT\_CLIENT\_HANDLER\_PORT environment variable. This port number is used for both TCP and UDP protocols and is the same for both.

## Windows

Setting the OSAGENT\_PORT environment variable will provide the same behavior as above.

## Connecting Smart Agents on different local networks

If you start multiple Smart Agents on your local network, they will discover each other by using UDP broadcast messages. Your network administrator configures a local network by specifying the scope of broadcast messages using the IP subnet mask. [Figure 17](#) shows two local networks, located on separate, connected local networks.



**Figure 17** Two Smart Agents on separate local networks

To allow the Smart Agent on one network to contact a Smart Agent on another local network, you must make the host name or IP address of the remote Smart Agent available. On the **host system**, IP addresses of Smart Agents outside of the local network are specified in a file. The name of this file may be specified by setting the OSAGENT\_ADDR\_FILE environment variable.

On VisiBrokerRT60 target systems, the location of the Smart Agents outside of your local network can also be specified via the OSAGENT\_ADDR\_FILE interface when starting the osagent.

## Use of the OSAGENT\_ADDR\_FILE Environment Variable (applicable on Development Host systems only)

The OSAGENT\_ADDR\_FILE environment variable specifies the filename of the file containing the address of agents outside your local network. When a client program or object implementation has this environment variable set,

the ORB will try each address in the file until a Smart Agent is located. This mechanism has the lowest precedence of all the mechanisms for specifying a host. If this file is not specified, the `<VBROKER_ADM Environment variable>/agentaddr` file is used.

[Code example 67](#) shows what this file would need to contain to allow the Smart Agent on local network #1 to connect to the Smart Agent on the network #2.

**Code example 67** Content of the agentaddr file for the osagent on network #1.

```
101.10.2.6
```

## Use of the OSAGENT\_ADDR\_TABLE By Smart Agents (applicable on VxWorks Target systems only)

To allow the Smart Agent on one network to contact a Smart Agent on another local network, you must make the IP address of the remote Smart Agent available in the OSAGENT\_ADDR\_TABLE.

The OSAGENT\_ADDR\_TABLE is customer declared array data structure specifying the IP addresses of other Smart Agents. These addresses represent Smart Agents executing on hosts/ targets located outside the local network with which the osagent is to communicate.

The include file `vosagent.h` provides a typedef for the structure to use when declaring your own OSAGENT\_ADDR\_TABLE. Additionally this header file provides an example of how to declare and use your own OSAGENT\_ADDR\_TABLE when starting the osagent.

**Code example 68** Specifying an OSAGENT\_ADDR\_TABLE on VxWorks Target System

```
#include "vosagent.h"
...
struct OSAGENT_ADDR_ENTRY {
    char ip_address[INET_ADDR_LEN];
    char subnet_mask[INET_ADDR_LEN];
    char broadcast_address[INET_ADDR_LEN];
};
//
// Sample OSAGENT_LOCAL_TABLE
OSAGENT_ADDR_ENTRY my_osagent_addr_table[] =
{
    {"101.10.2.6"},
    {NULL}
}
// Then when starting the osagent specify the address of your
// OSAGENT_ADDR_TABLE when calling startOsagent
startOsagent(210, // Osagent task priority
             0, // verbose = 0,
             2100, //port=21000, (default is 14000)
             100, //VisLogger task priority
             NULL, //pointer to your
osagent_local_table))
    my_osagent_addr_table); // pointer to your osagent_addr_table
...
```

### Note

If a remote network has multiple Smart Agents running, you should list the IP addresses of all of the Smart Agents on the remote network.

## How Smart Agents detect each other

Suppose two agents, Agent 1 and Agent 2, are listening on the same UDP port from two different machines on the same subnet. Agent 1 starts before Agent 2. The following events occur:

- When Agent 2 starts, it UDP broadcasts its existence and sends a request message to locate any other Smart Agents.
- Agent 1 makes note that Agent 2 is available on the network and responds to the request message.
- Agent 2 makes note that another agent, Agent 1, is available on the network.

If Agent 2 is terminated gracefully (such as killing with **Ctrl-C** on Unix or Windows), Agent 1 is notified that Agent 2 is no longer available.

If Agent 2 is terminated abnormally (such as rebooting the VisiBrokerRT60 target system that Agent 2 is running on), Agent 1 is not notified that Agent 2 is no longer available. Agent 1 continues until:

- a client asks for an object reference that does not exist in Agent 1's dictionary, and Agent 1 forwards the request to Agent 2. Since Agent 2 is no longer available, Agent 1 is forced to clean up.

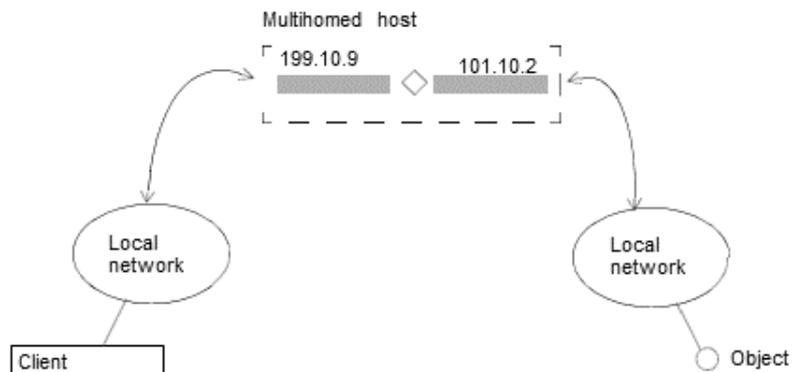
or

- until the Agent to Agent heartbeat mechanism identifies that Agent to Agent communication between Agent 1 and Agent 2 has failed at which point Agent 1 will clean up knowledge of Agent 2 from its data structures.

Until Agent 1 is forced to clean up, `osfind` still shows two agents listed and catches `ObjLocation::Fail` exception.

## Working with multihomed hosts

When you start the Smart Agent on a host that has more than one IP address (known as a multihomed host) it can provide a powerful mechanism for bridging objects located on separate local networks. All local networks to which the host is connected will be able to communicate with a single Smart Agent, effectively bridging the local networks.



**Figure 18** Smart Agent on a multihomed host

### UNIX

On a multi-homed UNIX development host or target system, the Smart Agent dynamically configures itself to listen and broadcast on all of the interfaces which support point-to-point connections or broadcast connections. You may explicitly specify interface settings using the

`localaddr` file, as described in “[Specifying interface usage for Smart Agents](#)”.

## Windows

On a multihomed Windows development host, the Smart Agent is not able to dynamically determine the correct subnet mask and broadcast address values. To overcome this limitation, you must explicitly specify the interface settings you want the Smart Agent to use with the `localaddr` file.

When you start the Smart Agent with the verbose option, each interface that the Smart Agent uses will be listed at the beginning of the messages produced. [Code example 69](#) shows the sample output from a Smart Agent started with the verbose option on a multihomed host.

**Code example 69** Verbose output from a Smart Agent started on a multihomed host

```
Bound to the following interfaces:
Address: 199.10.9.5 Subnet: 255.255.255.0 Broadcast:199.10.9.255
Address: 101.10.2.6 Subnet: 255.255.255.0 Broadcast:101.10.2.255
...
```

As shown in [Code example 69](#), the output shows the address, subnet mask, and broadcast address for each interface in the machine.

## UNIX

- This output should match the results from the command `ifconfig -a`.
- For Tornado targets this output should match the results from the WindShell function call `ifShow`.

## Windows

- This output should match the results from the Windows command `ipconfig -a`.

If you wish to override these settings, you can specify this interface information in the `localaddr` file. See “[Specifying interface usage for Smart Agents](#)” below for details.

## Specifying interface usage for Smart Agents

### Use of the LOCAL\_ADDR\_FILE For Multi-Homed hosts

#### Note

It is not necessary to specify interface information on a single-homed host.

You can specify interface information for each interface you wish the Smart Agent to use on your multihomed host in the `localaddr` file. The `localaddr` file should have a separate line for each interface that contains the host’s IP address, subnet mask, and broadcast address. By default, VisiBroker-RT for C++ searches for the `localaddr` file in the `VBROKER_ADM` directory. You can override this location by setting the `OSAGENT_LOCAL_FILE` environment variable to point to this file. Lines in this file that begin with a “#” character are treated as comments and ignored. [Code example 70](#) shows the contents of the `localaddr` file for the multihomed host listed above.

**Code example 70** Contents of an example `localaddr` file

```
#entries of format <address> <subnet_mask> <broadcast address>
199.10.9.5 255.255.255.0 199.10.9.255
101.10.2.6 255.255.255.0 101.10.2.255
```

## UNIX

Though the Smart Agent can automatically configure itself on a multihomed host running UNIX, you can use the `localaddr` file to explicitly specify the interfaces that your host contains. You can display all the available interface values for your host by using the following command:

```
prompt> ifconfig -a
```

Output from this command appears similar to the following:

```
lo0: flags=849<UP,LOOPBACK,RUNNING,MULTICAST> mtu 8232
inet 127.0.0.1 netmask ff000000
```

```

le0: flags=863<UP,BROADCAST,NOTRAILERS,RUNNING,MULTICAST> mtu
1500
    inet 199.10.9.5 netmask ffffffff0 broadcast 199.10.9.255
le1: flags=863<UP,BROADCAST,NOTRAILERS,RUNNING,MULTICAST> mtu
1500
    inet 101.10.2.6 netmask ffffffff0 broadcast 101.10.2.255

```

## Windows

The use of the `localaddr` file with multihomed hosts is required for hosts running Windows because the Smart Agent is not able to automatically configure itself. You can obtain the appropriate values for this file by accessing the TCP/IP protocol properties from the Network Control Panel. If your host is running Windows NT, the `ipconfig` command will provide the needed values. You run this command as follows:

```
prompt>ipconfig
```

Output from this command appears similar to the following:

```

Ethernet adapter El59x1:
    IP Address. . . . . : 199.10.9.5
    Subnet Mask. . . . . : 255.255.255.0
    Default Gateway. . . : 199.10.9.1

Ethernet adapter Elnk32:
    IP Address. . . . . : 101.10.2.6
    Subnet Mask. . . . . : 255.255.255.0
    Default Gateway. . . . : 101.10.2.1

```

## Use of the LOCAL\_ADDR\_FILE For Multi-Homed VxWorks Targets

### Note

This is applicable on VxWorks Target systems *only*.

You can specify multiple network interface information in the **OSAGENT\_LOCAL\_TABLE** table. The **OSAGENT\_LOCAL\_TABLE** is a customer defined table which contains a list of the network interfaces that the osagent is to use. Each entry in this table should contain the IP address, subnet mask, and broadcast address for a single interface.

The include file `vosagent.h` provides a typedef for the structure to use when declaring your own `OSAGENT_LOCAL_TABLE`. Additionally this header file provides an example of how to declare and use your own `OSAGENT_LOCAL_TABLE` when starting the osagent.

### Code example 71 Specifying an OSAGENT\_LOCAL\_TABLE on VxWorks Target System

```

#include "vosagent.h"
...
struct OSAGENT_LOCAL_ENTRY {
    char    ip_address[INET_ADDR_LEN];
    char    subnet_mask[INET_ADDR_LEN];
    char    broadcast_address[INET_ADDR_LEN];
};

// -----
// Sample OSAGENT LOCAL TABLE
OSAGENT_LOCAL_ENTRY my_osagent_local_table[] =
{
    {"224.192.128.56", "255.255.255.0", "224.192.128.255"},
    {"196.192.86.99", "255.255.255.0", "196.192.86.99"},
    {NULL}
}

// Then when starting the osagent specify the address of your
// OSAGENT_LOCAL_TABLE when calling startOsagent
startOsagent(210, // Osgent task priority
             0, // verbose = 0,

```

```

        2100.        //port=21000,      (default is 14000)
        100,        //VisLogger task priority
my_osagent_addr_table, //pointer to your osagent_local_table))
        NULL);    // pointer to your osagent_addr_table
...

```

## Using point-to-point communications

VisiBroker-RT for C++ provides you with three different mechanisms for circumventing the use of UDP broadcast messages for locating Smart Agents. When a Smart Agent is located with any of these alternate approaches, that Smart Agent will be used for all subsequent interactions. If a Smart Agent cannot be located using any of these alternate approaches, the ORB will revert to using the broadcast message scheme to locate a Smart Agent.

### Specifying a host as a runtime parameter

[Code example 70](#) shows how you can specify the IP address where a Smart Agent is running as a runtime parameter for your client program or object implementation. Since specifying an IP address will cause a point-to-point connection to be established, you can even specify an IP address of a node located outside your local network. This mechanism takes precedence over any other node address specification.

#### Code example 72 Turning Off Agent Communication via a ORB property

```

void do_corba(void)
{
    /*-----*/
    /* ORB init options can be specified in two ways.      */
    /* 1) By calling start_corba and specifying the        */
    /*    ORB initialization string                        */
    /*    (e.g. start_corba("-ORBagentport 19000"))        */
    /* 2) Programatically by specifying the                */
    /*    ORB initialization options in the                */
    /*    default_argc and default_argv variables below.  */
    /*-----*/
    /* PLEASE NOTE THAT THE OPTIONS PASSED IN VIA start_corba */
    /* OVERRIDE THE OPTIONS THAT ARE SET PROGRAMATICALLY.    */
    /*-----*/

    // Get the property manager; notice the value returned
    // is not placed into a 'var' type.
    const char * my_properties[] =
    {
        "vbroker.agent.addr=<ip address>",
        NULL
    };

    VISPropertyTable property_table("my_properties",
    my_properties);

    int default_argc = 2;
    char *default_argv[] = {"ORBpropTable", "my_properties"};
    char **new_argv;
    int new_argc = VISUtil::stringToArgv(&new_argv, default_argv,
    default_argc, ORB_options_string);

    /*-----*/
    /* Call ORB_init                                        */
    /*-----*/
    VISTRY
    {
        // Initialize the ORB

```

```
orb = CORBA::ORB_init(new_argc, new_argv);
VISUtil::freeArgv(new_argc, & new_argv);

...

...
```

By default, `vbroker.agent.addr` is set to `NULL`.

## Specifying an IP address with an environment variable

You can specify the IP address of a Smart Agent by setting the `OSAGENT_ADDR` environment variable prior to starting your client program or object implementation. This environment variable takes precedence if a node address is not specified as a runtime parameter.

**Figure 19** Setting the `OSAGENT_ADDR` environment variable using the Tornado WindShell

### Tornado

```
--> putenv("OSAGENT_ADDR=199.10.9.5")
--> start_corba
```

#### Note

This requires `ENV_VARS` as part of VxWorks Kernel.

## Specifying hosts with the `agentaddr` table

Your client program or object implementation can use the `agentaddr` table, described in [“Connecting Smart Agents on different local networks”](#), to circumvent the use of UDP broadcast message to locate a Smart Agent. Simply create a table containing the IP addresses or fully qualified hostname of each node where a Smart Agent is running and then specify this `OSAGENT_ADDR_TABLE` during `ORB_init()`. When a client program or object implementation has specified an `OSAGENT_ADDR_TABLE`, the ORB will try each address in the table until a Smart Agent is located. This mechanism has the lowest precedence of all the mechanisms for specifying a host. If an `OSAGENT_ADDR_TABLE` is not specified, the ORB will default to using UDP Broadcast to find a Smart Agent.

## Ensuring object availability

You can provide fault tolerance for objects by starting instances of those objects on multiple nodes. If an implementation becomes unavailable, the ORB will detect the loss of the connection between the client program and the object implementation and will automatically contact the Smart Agent to establish a connection with another instance of the object implementation, depending on the effective rebind policy established by the client. See [“Using Quality of Service”](#) for more information on establishing client policies.

#### Caution

The rebind option must be enabled if the ORB is to attempt to reconnect the client with a replica object implementation. This is the default behavior.

## Invoking methods on stateless objects

Your client program can invoke a method on an object implementation which does not maintain state without being concerned if a new instance of the object is being used.

## Achieving fault-tolerance for objects that maintain state

Fault tolerance can also be achieved with object implementations that maintain state, but it will not be transparent to the client program. In these cases, your client program must either use the Quality of Service (QoS) policy `VB_NOTIFY_REBIND` or register an interceptor for the ORB object. For information on using QoS, see [“Using Quality of Service”](#).

When the connection to an object implementation fails and the ORB reconnects the client to a replica object implementation, the `bind()` method of the bind interceptor will be invoked by the ORB. The client must provide an implementation of this bind method to bring the state of the replica up to date. Interceptors are described in [“Using Portable Interceptors”](#).

## Migrating objects between VisiBrokerRT60 Systems

Object migration is the process of terminating an object implementation on one VisiBrokerRT60 system, and then starting it on another VisiBrokerRT60. Object migration can be used to provide load balancing by moving objects from overloaded systems to systems that have more resources or processing power (there is no load balancing between servers registered with different `osagents`.) Object migration can also be used to keep objects available when a target has to be shutdown for hardware or software maintenance.

### Note

The migration of objects that do not maintain state is transparent to the client program. If a client is connected to an object implementation that has migrated, the Smart Agent will detect the loss of the connection and transparently reconnect the client to the new object on the new VisiBrokerRT60 system.

## Migrating objects that maintain state

The migration of objects that maintain state is also possible, but it will not be transparent to a client program that has connected before the migration process begins. In these cases, the client program must register an interceptor for the object. When the connection to the original object is lost and the ORB reconnects the client to the object, the interceptor's `rebind_succeeded()` member function will be invoked by the ORB. The client can implement this member function to bring the state of the object up to date. Interceptors are described in [“Using Portable Interceptors”](#).

## Migrating instantiated objects

If the objects that you wish to migrate were created by a VisiBrokerRT60 system instantiating the implementation's class, you need only start it on a new system and deactivate the object implementation from the original system. When the original instance is deactivated, it will be unregistered

with the Smart Agent. When the new instance is started on the new system, it will register with the Smart Agent. From that point on, client invocations will be routed to the object implementation on the new system.

## Reporting all objects and services

The Smart Finder (`osfind`) development host command reports on all VisiBroker-RT for C++ related objects and services which are currently available on a given network.

You can use `osfind` to determine the number of Smart Agent processes running on the network and the exact target on which they are executing. The `osfind` command also reports on all VisiBroker-RT for C++ objects that are active on the network. You can use `osfind` to monitor the status of the network and locate stray objects during the debugging phase.

The `osfind` command has the following syntax and can be run from any Windows or Unix development host:

```
osfind [options]
```

The following options are valid with `osfind`. If no options are specified, `osfind` lists all of the agents, OAD's, and implementations in your domain.

Option	Description
-a	Lists all Smart Agents in your domain.
-o	Lists all Object Activation Daemons in your domain.
-d	Prints hostnames as quad addresses.

---



# Using the Location Service

The *VisiBroker-RT for C++ Location Service* provides enhanced object discovery that enables you to find object instances based on particular attributes. Working with *VisiBroker-RT for C++ Smart Agents*, the Location Service notifies you of what objects are presently accessible on the network, and where they reside. The Location Service is a *VisiBroker-RT for C++* extension to the CORBA specification and is only useful for finding objects implemented with *VisiBroker-RT for C++*.

## NOTE

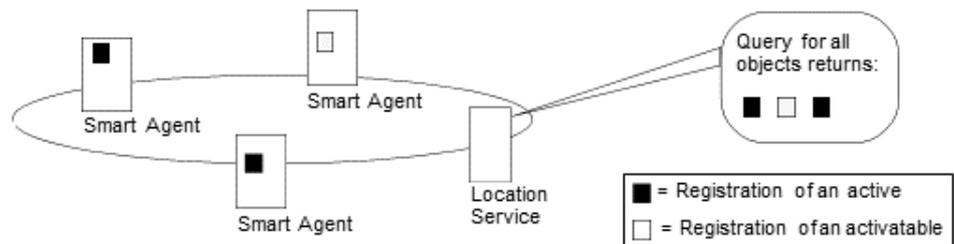
The libraries `libagentsupport.o` and `liblocsupport.o` are required when building a `VisiBrokerRT60` application to support use of the VisiBroker Location Service. For a description of all the libraries provided by the VisiBroker-RT for C++ product please refer to “[Step 6: Integrating VisiBroker-RT with Tornado/VxWorks](#)”.

## What is the Location Service?

The Location Service is an extension to the CORBA specification that provides general-purpose facilities for locating object instances. The Location Service communicates directly with one Smart Agent which maintains a *catalog*, which contains the list of the instances it knows about and the information it knows about the instances. When queried by the Location Service, a Smart Agent forwards the query to the other Smart Agents, and aggregates their replies in the result it returns to the Location Service.

The Location Service knows about all object instances that are registered on a POA with the `BY_INSTANCE` Policy and objects that are registered as persistent on a BOA.

The following diagram illustrates this concept.



**Figure 20** Using the Smart Agent to find instances of objects

## Note

A server specifies an instance’s scope when it creates the instance. Only globally-scoped instances are registered with Smart Agents.

The Location Service can make use of the information the Smart Agent keeps about each object instance. For each object instance, the Location Service maintains information encapsulated in the structure `ObjLocation::Desc` shown in [IDL sample 8](#).

### IDL sample 8 IDL for the Desc structure

```
struct Desc {
    Object ref;
    IIOP::ProfileBodyValue iiop_locator;
    string repository_id;
    string instance_name;
    boolean activable;
    string agent_hostname;
};
typedef sequence<Desc> DescSeq;
```

The IDL for the Desc structure contains the following information:

- *Object reference*, or a handle for invoking the object.
- *iiop\_locator* interface allows access to the host name and the port of the instance's server. This information is only meaningful if the object is connected with IIOP, which is the only supported protocol. Host names are returned as strings in the instance description.
- *Repository ID*, which is the interface designation for the object instance that can be looked up in the Interface and Implementation Repositories. If an instance satisfies multiple interfaces, the catalog contains an entry for each interface, as if there were an instance for each interface.
- *Instance name*, or the name given to the object by its server.
- Activatable flag which differentiates between instances that can be activated by an OAD, and instances that are manually started.
- Host name of the Smart Agent with which the instance is registered.

The Location Service is useful for purposes such as load balancing and monitoring. Suppose that replicas of an object are located on several VisiBrokerRT60 systems. You could deploy a bind interceptor that maintains a cache of the VisiBrokerRT60 systems names that offer a replica and each target's recent load average. The interceptor updates its cache by asking the Location Service for the systems currently offering instances of the object, and then queries the targets to obtain their load averages. The interceptor then returns an object reference for the replica on the target with the lightest load. See ["Using Portable Interceptors"](#) for more information about writing interceptors.

## Location Service components

The Location Service is accessible through the `Agent` interface. Methods for the `Agent` interface can be divided into two groups: those that query a Smart Agent for data describing instances and those that register and unregister *triggers*. Triggers provide a mechanism by which clients of the Location Service can be notified of changes to the availability of instances.

### What is the Location Service agent?

The Location Service Agent is a collection of methods that enable you to discover objects on a network of Smart Agents. You can query based on the interface's repository ID, or based on a combination of the interface's repository ID and the instance name. Results of a query can be returned as either *object references* or more complete *instance descriptions*. An object reference is simply a handle to a specific instance of the object located by a Smart Agent.

Instance descriptions contain the object reference, as well as the instance's interface name, instance name, host name and port number, and

information about its state (for example, whether it is running or can be activated).

**Note**

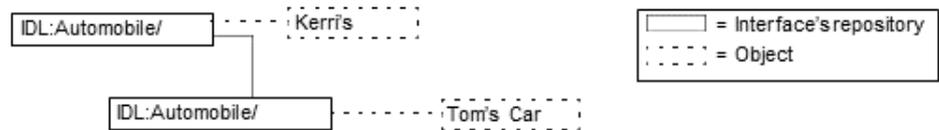
The Location Service is provided as a separate ORB library. To use the Location Service, you must add Location Service support to the VisiBrokerRT60 target system. Location Service support is delivered as "Add-On" functionality for VisiBroker-RT for C++.support can be included into a VisiBrokerRT60 application by building the application with the libagentsupport library. Please refer to ["Step 6: Integrating VisiBroker-RT with Tornado/VxWorks"](#) for information on adding the libagentsupport library to your application.

Figure 21 illustrates the use of interface repository IDs and instance names given the following example IDL:

```

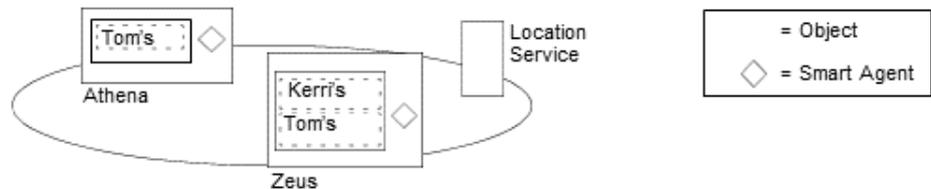
module Automobile {
    interface Car{...};
    interface Sedan:Car {...};
}

```



**Figure 21** Use of interface repository IDs and instance names

Given the example in Figure 21, the following diagram visually depicts Smart Agents on a network with references to instances of Car. In this example, there are three instances: one instance of Kerri's Car and two replicas of Tom's Car.



**Figure 22** Smart Agents on a network with instances of an interface

The following sections explain how the methods provided by the Agent class can be used to query VisiBroker-RT for C++ Smart Agents for information. Each of the query methods can raise the Fail exception, which provides a reason for the failure.

**Obtaining names of all hosts running Smart Agents**

Using the HostnameSeq all\_agent\_locations() method, you can find out which servers are hosting VisiBroker-RT for C++ Smart Agents. In the example shown in Figure 22, this method would return the names of two hosts: Athena and Zeus.

## Finding all accessible interfaces

You can query the VisiBroker-RT for C++ Smart Agents on a network to find out about all accessible interfaces. To do so, you can use the `RepositoryIDSeq all_repository_ids()` method. In the example shown in [Figure 22](#), this method would return the repository IDs of two interfaces: Car and Sedan.

### Note

Earlier versions of the VisiBroker-RT for C++ ORB used IDL interface names to identify interfaces, but the Location Service uses the repository id instead. To illustrate the difference, if an interface name is `::module1::module2::interface`, the equivalent repository id is `IDL:module1/module2/interface:1.0`. For the example shown in [Figure 21](#), the repository ID for Car would be `IDL:Automobile/Car:1.0`, and the repository ID for Sedan would be `IDL:Automobile/Sedan:1.0`.

## Obtaining references to instances of an interface

You can query VisiBroker-RT for C++ Smart Agents on a network to find all available instances of a particular interface. When performing the query, you can use either of these methods:

**Table 14** Obtaining references to objects that implement a given interface

Method	Description
<code>ObjSeq all_instances(in string repository_id)</code>	Use this method to return object references to instances of the interface.
<code>DescSeq all_instance_descs(in string repository_id)</code>	Use this method to return an instance description for instances of the interface.

In the example shown in [Figure 22](#), a call to either method with the request `IDL:Automobile/Car:1.0` would return three instances of the Car interface: Tom's Car on Athena, Tom's Car on Zeus, and Kerri's Car. The Tom's Car instance is returned twice because there are occurrences of it with two different Smart Agents.

## Obtaining references to like-named instances of an interface

Using one of the following methods, you can query VisiBroker-RT for C++ Smart Agents on a network to return all occurrences of a particular instance name.

**Table 15** References to like-named instances of an interface

Method	Description
<code>ObjSeq all_replica(in string repository_id, in string instance_name)</code>	Use this method to return object references to like-named instances of the interface.
<code>DescSeq all_replica_descs(in string repository_id, in string instance_name)</code>	Use this method to return an instance description for like-named instances of the interface.

In the example shown in [Figure 22](#), a call to either method specifying the repository ID `IDL:Automobile/Sedan:1.0` and instance name Tom's Car would return two instances because there are occurrences of it with two different Smart Agents.

## What is a trigger?

A trigger is essentially a callback mechanism that lets you determine changes to the availability of a specified instance. It is an asynchronous alternative to polling an `Agent`, and is typically used to recover after the connection to an object has been lost. Whereas queries can be employed in many ways, triggers are special-purpose.

## Looking at trigger methods

The trigger methods in the `Agent` class are described in the following table:

**Table 16** Trigger methods

Method	Description
<code>void reg_trigger(in TriggerDesc desc, in TriggerHandler handler)</code>	Use this method to register a trigger handler.
<code>void unreg_trigger(in TriggerDesc desc, in TriggerHandler handler)</code>	Use this method to unregister a trigger handler.

Both of the `Agent` trigger methods can raise the `Fail` exception, which provides a reason for the failure.

The `TriggerHandler` interface consists of the methods described in the following table:

**Table 14.4** TriggerHandler interface method

Method	Description
<code>void impl_is_ready(in Desc desc)</code>	This method is called by the Location Service when an instance matching the <code>desc</code> becomes accessible.
<code>void impl_is_down(in Desc desc)</code>	This method is called by the Location Service when an instance becomes unavailable.

## Creating triggers

A `TriggerHandler` is a callback object. You implement a `TriggerHandler` by deriving from the `TriggerHandlerPOA` class (or the `TriggerHandlerImpl` class if using BOA), and implementing its `impl_is_ready()` and `impl_is_down()` methods. To register a trigger with the Location Service, you use the `reg_trigger()` method in the `Agent` interface. This method requires that you provide a description of the instance you want to monitor, and the `TriggerHandler` object you want invoked when the availability of the instance changes. The instance description (`TriggerDesc`) can contain combinations of the following instance information: repository ID, instance name, and host name. The more instance information you provide, the more particular your specification of the instance.

**IDL sample 9** IDL for `TriggerDesc`

```
struct TriggerDesc {
    string repository_id;
    string instance_name;
    string host_name;
};
```

### Note

If a field in the `TriggerDesc` is set to the empty string (`""`), it is ignored. The default for each field value is the empty string.

For example, a `TriggerDesc` containing only a repository ID matches any instance of the interface. Looking back to our example in [Figure 22](#), a trigger for any instance of `IDL:Automobile/Car:1.0` would occur when one of the following instances becomes available or unavailable: Tom's Car on Athena, Tom's Car on Zeus, or Kerri's Car. Adding an instance name of "Tom's Car" to the `TriggerDesc` tightens the specification so that the trigger only occurs when the availability of one of the two "Tom's Car" instances changes. Finally, adding a host name of Athena refines the trigger further so that it only occurs when the instance of Tom's Car on Athena becomes available or unavailable.

## Looking at only the first instance found by a trigger

Triggers are "sticky." A `TriggerHandler` is invoked every time an object satisfying the trigger description becomes accessible. You may only be interested in learning when the first instance becomes accessible. If this is the case, invoke the `Agent's unreg_trigger()` method to unregister the trigger after the first occurrence is found.

## Querying an agent

This section contains two examples of using the Location Service to find instances of an interface. The first example uses the `Account` interface shown in the following IDL excerpt:

### IDL sample 10 Account example interface definition

```
// Bank.idl
module Bank {
    interface Account {
        float balance();
    };
    interface AccountManager {
        Account open (in string name);
    };
};
```

## Finding all instances of an interface

The following code sample uses the `all_instances()` method to locate all instances of the `Account` interface. Notice that the Smart Agents are queried by passing "LocationService" to the `ORB::resolve_initial_references()` method, then narrowing the object returned by that method to an `ObjLocation::Agent`. Notice, as well, the format of the `Account` repository id—`IDL:Bank/Account:1.0`.

### Code example 73 Finding all instances satisfying the AccountManager interface

```
...

void account_finder()
{
    VISTRY
    {
        // Obtain a reference to the Location Service
        CORBA::Object_var obj =
            orb->resolve_initial_references("LocationService");

        if ( CORBA::is_nil(obj) )
        {
            cout << "Unable to locate initial LocationService" <<
            endl;
            return;
        }
    }
}
```

```

ObjLocation::Agent_var the_agent =
    ObjLocation::Agent::_narrow(obj);

// Query the Location Service for all implementations of
// the Account interface
ObjLocation::ObjSeq_var accountRefs;
VISIFNOT_EXCEP
accountRefs = the_agent->
    all_instances("IDL:Bank/AccountManager:1.0");
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
    cout << "Obtained " << accountRefs->length()
        << " Account objects" << endl;
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
    for (CORBA::ULong i=0; i < accountRefs->length(); i++)
    {
        cout << "Stringified IOR for account #" << i << ":" <<
endl;
        CORBA::String_var
stringified_ior(orb->object_to_string(accountRefs[i]));
        cout << stringified_ior << "\n" << endl;
    } VISEND_IFNOT_EXCEP
}
VISCATCH (CORBA::Exception, e)
{
    cout << "Caught exception: " << e << endl; return;
}
VISEND_CATCH
return;
}

```

## Finding everything known to Smart Agents

The following code sample shows how to find everything known to Smart Agents. It does this by invoking the `all_repository_ids()` method to obtain all known interfaces. Then it invokes the `all_instances_descs()` method for each interface to obtain the instance descriptions.

### Code example 74 Finding everything known to a Smart Agent

```

...

void finder()
{
    VISTRY
    {
        CORBA::Object_var obj =
            orb->resolve_initial_references("LocationService");

        if (CORBA::is_nil(obj) )
        {
            cout << "Unable to locate initial LocationService" <<
endl;
            return;
        }

        ObjLocation::Agent_var the_agent =
            ObjLocation::Agent::_narrow(obj);

        //Report all hosts running osagents
        ObjLocation::HostnameSeq_var HostsRunningAgents;
        VISIFNOT_EXCEP
    }
}

```

```

        HostsRunningAgents = the_agent->all_agent_locations();
    VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
    cout << "Located " << HostsRunningAgents->length()
        << " Hosts running Agents" << endl;
    VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
    for (CORBA::ULong k=0; k<HostsRunningAgents->length();
k++)
    {
        cout << "\tHost #" << (k+1) << ": "
            << (const char*) HostsRunningAgents[k] << endl;
    }
    cout << endl;
    VISEND_IFNOT_EXCEP

    // Find and display all Repository Ids
    ObjLocation::RepositoryIdSeq_var repIds;
    VISIFNOT_EXCEP
    repIds = the_agent->all_repository_ids();
    VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
    cout << "Located" << repIds->length() << " Repository Ids"
<< endl;
    VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
    for (CORBA::ULong j=0; j<repIds->length(); j++)
        cout << "\tRepository ID #" << (j+1) << ": " << repIds[j]
<< endl;
    VISEND_IFNOT_EXCEP

    // Find all Object Descriptors for each Repository Id
    VISIFNOT_EXCEP
    for (CORBA::ULong i=0; i < repIds->length(); i++)
    {
        ObjLocation::DescSeq_var descriptors =
            the_agent->all_instances_descs(repIds[i]);
        VISIF_EXCEP(break;);

        cout << endl;
        cout << "Located " << descriptors->length() << " objects
for "
            << (const char*) (repIds[i]) << " (Repository Id #" <<
(i+1)
            << "):" << endl;

        for (CORBA::ULong j=0; j < descriptors->length(); j++)
        {
            cout << endl;
            cout << (const char*) repIds[i] << " #" << (j+1) <<
":" << endl;

            cout << "\tInstance Name \t= "
                << descriptors[j].instance_name << endl;
            cout << "\tHost\t= "
                << descriptors[j].iiop_locator.host <<endl;
            cout << "\tPort\t= "
                << descriptors[j].iiop_locator.port <<endl;
            cout << "\tAgent Host \t= "
                << descriptors[j].agent_hostname <<endl;
            cout << "\tActivable \t= " <<
                (descriptors[j].activable?"YES":"NO")
                << endl;
        }
    }
}

```

```

        VISEND_IFNOT_EXCEP
    }
    VISCATCH(CORBA::Exception, e)
    {
        cout << "CORBA Exception during execution of find_all: " << e
        << endl;
        return;
    }
    VISEND_CATCH
return;
}

```

## Writing and registering a trigger handler

The following section illustrates how a trigger is implemented and registered.

### Implementing and registering a trigger handler

The following code sample implements and registers a `TriggerHandler`. The `TriggerHandlerImpl`'s `impl_is_ready()` and `impl_is_down()` methods display the description of the instance that caused the trigger to be invoked, and optionally unregister itself. If it unregisters itself, the method calls the `CORBA::BOA::deactivate_obj()` method followed by `CORBA::release()`. This will remove the Trigger from the BOA's Active Object Map and finally call the Triggers destructor to finish cleanup of the Trigger object instance.

Notice that the `TriggerHandlerImpl` class keeps a copy of the `desc` and `Agent` parameters with which it was created. The `unreg_trigger()` method requires the `desc` parameter. The `Agent` parameter is duplicated in case the reference from the main program is released.

#### Code example 75 Implementing a trigger handler

```

...

// Instances of this class will be called back by the Agent when
// the
// event for which it is registered happens.

class TriggerHandlerImpl : public
_sk_ObjLocation::_sk_TriggerHandler
{
public:

    TriggerHandlerImpl(ObjLocation::Agent_ptr agent,
        const ObjLocation::TriggerDesc& initial_desc)
        : _agent(ObjLocation::Agent::_duplicate(agent)),
          _initial_desc(initial_desc) {}

    void impl_is_ready(const ObjLocation::Desc& desc)
    {
        notification(desc, 1);
    }

    void impl_is_down(const ObjLocation::Desc& desc)
    {
        notification(desc, 0);
    }

private:
    ObjLocation::Agent_var _agent;
    ObjLocation::TriggerDesc _initial_desc;
}

```

```

void notification(
    const ObjLocation::Desc& desc, CORBA::Boolean isReady)
{
    if (isReady)
    {
        cout << "Implementation is ready:" << endl ;
    }
    else
    {
        cout << "Implementation is down:" << endl ;
    }

    cout << "\tRepository Id = " << desc.repository_id << endl;
    cout << "\tInstance Name = " << desc.instance_name << endl;
    cout << "\tHost Name = " << desc.iiop_locator.host << endl;
    cout << "\tPort = " << desc.iiop_locator.port << endl;
    cout << "\tAgent Host = " << desc.agent_hostname << endl;
    cout << "\tActivable= " << (desc.activable? "YES" : "NO") <<
endl;
    cout << endl;
    cout << "Unregister this handler and exit (y/n)? " << endl;

    char prompt[256];
    cin >> prompt;

    if ((prompt[0] == 'y') || (prompt[0] == 'Y'))
    {
        VISTRY
        {
            agent->unreg_trigger(_initial_desc, this);
        }

        VISCATCH(ObjLocation::Fail, e)
        {
            cout << "Failed to unregister trigger with reason=["
                << (int) e.reason << "]" << endl;

            return;
        }
        VISEND_CATCH

        cout << "Deactivate and release account trigger...." <<
endl ;

        VISTRY
        {
            boa->deactivate_obj(trig);
        }
        VISCATCH(ObjLocation::Fail, e)
        {
            cout << "Failed to deactivate trigger" << endl ;
            cerr << e << endl;
            return;
        }
        VISEND_CATCH

        CORBA::release(trig);
    }
}

};

void account_trigger(void)
{
    VISTRY
    {
        int argc = 1;
        char*argv[] ={"DO_CORBA"};

        VISIFNOT_EXCEP
        // Initialize the BOA.

```

```

        boa = orb->BOA_init(argc, argv);
        VISEND_IFNOT_EXCEP

        CORBA::Object_ptr obj = orb->
            resolve_initial_references("LocationService");

    if ( CORBA::is_nil(obj) )
    {
        cout << "Unable to locate initial LocationService" <<
endl;
    }
    else
    {
        ObjLocation::Agent_var the_agent =
ObjLocation::Agent::_narrow(obj);

        // Create the trigger descriptor to notify us about
        // OSAgent changes with respect to Account objects
        ObjLocation::TriggerDesc *desc = new
ObjLocation::TriggerDesc;

        desc->repository_id = (const char*) "IDL:Bank/
AccountManager:1.0";
        desc->instance_name = (const char*)"";
        desc->host_name = (const char*)"";

        trig = new TriggerHandlerImpl(the_agent, *desc);

        boa->obj_is_ready(trig);

        VISIFNOT_EXCEP
        the_agent->reg_trigger(*desc, trig); VISEND_IFNOT_EXCEP
    }
}
VISCATCH ( CORBA::Exception, e)
{
    cout << "account trigger caught Exception:" << endl ;
    cerr << e << endl;
    return;
}
VISEND_CATCH

return;
}

```



# Using the Naming Service

This chapter describes how to use the VisiBroker-RT for C++ Naming Service which is a complete implementation of the Interoperable Naming Specification document (*orbos/98-10-11*) from the OMG.

## NOTE

The libraries:

- `libservicesupport.o`
- `libname_c_s.o`
- `libname.o`

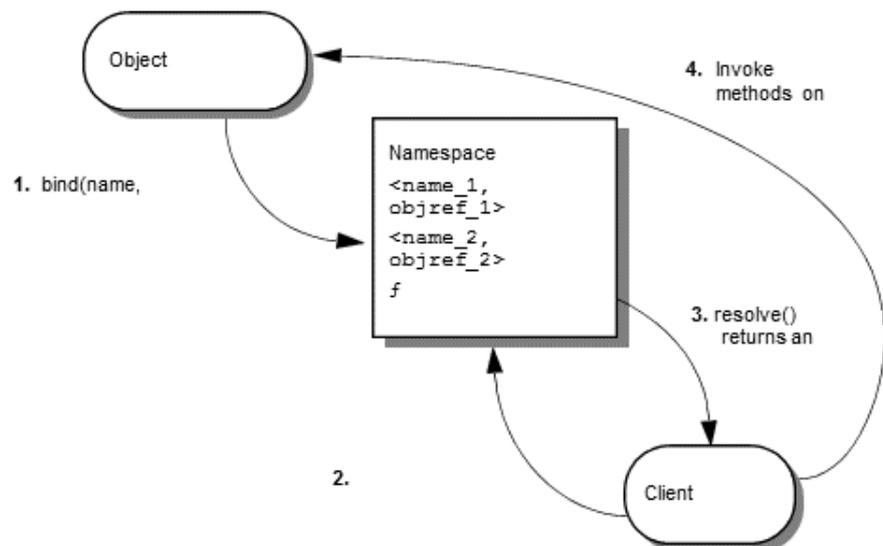
are required when building a VisiBrokerRT60 application to support use of the VisiBroker Naming Service. For a description of all the libraries provided by the VisiBroker-RT for C++ product please refer to “[Step 6: Integrating VisiBroker-RT with Tornado/VxWorks](#)”.

## Overview

The Naming Service allows you to associate one or more *logical* names with an object reference and store those names in a *namespace*. It also allows your client applications to use the Naming Service to obtain an object reference by using the logical name assigned to that object.

Figure 23 contains a simplified view of the Naming Service that shows how:

- 1 An object implementation can *bind* a name to one of its objects within a namespace.
- 2 Client applications can then use the same namespace to *resolve* a name which returns an object reference to a naming context or an object.



**Figure 23** Binding, resolving, and using an object name from a naming context within a namespace

There are some important differences to consider between locating an object implementation with the VisiBroker-RT for C++ Naming Service as opposed to the Smart Agent:

- Smart Agent uses a flat namespace, while the Naming Service uses a hierarchical one.
- Object's interface name is defined at the time you compile your client and server applications. Changing an interface name requires that you recompile your applications. In contrast, the Naming Service allows object implementations to bind logical names to its objects at runtime.
- Object may implement only one interface name, but the Naming Service allows you to bind more than one logical name to a single object.

## Understanding the namespace

Figure 24 shows how the Naming Service might be used to name objects that make up an order entry system. This hypothetical order entry system organizes its namespace by geographic region, then by department, and so on. The Naming Service allows you to organize the namespace in a hierarchical structure of `NamingContext` objects that can be traversed to locate a particular name. For example, the logical name `NorthAmerica/ShippingDepartment/Orders` could be used to locate an `Order` object.

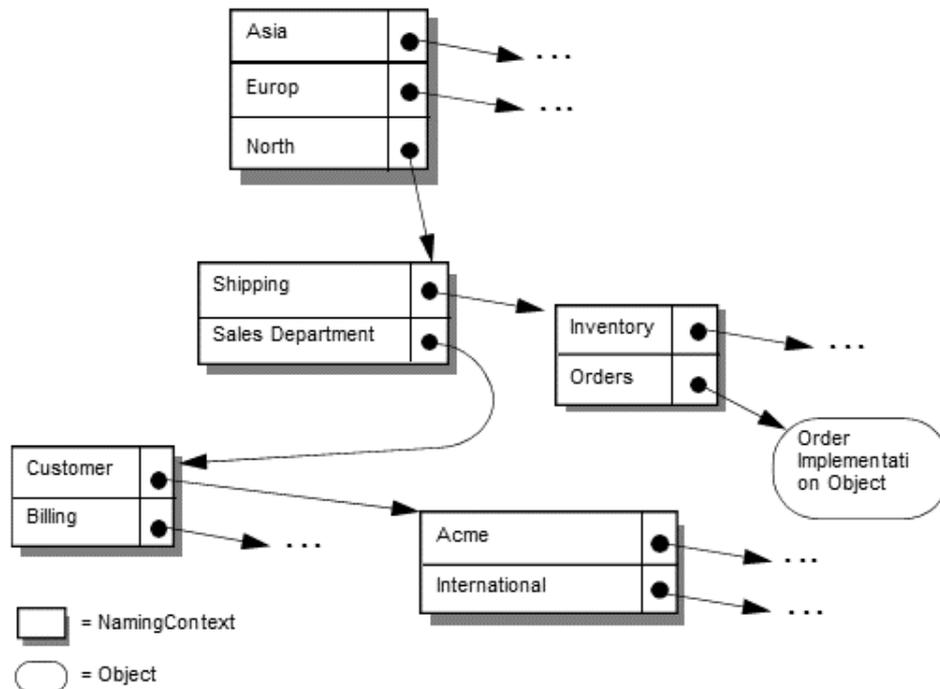


Figure 24 Naming scheme for an order entry system

## Naming contexts

To implement the namespace shown in Figure 24 with the VisiBroker-RT for C++ Naming Service, each of the shadowed boxes would be implemented by a `NamingContext` object. A `NamingContext` object contains a list of Name structures that have been bound to object implementations or to other `NamingContext` objects. Though a logical name may be bound to a `NamingContext`, it is important to realize that a `NamingContext` does not, by default, have a logical name associated with it nor is such a name required.

Object implementations use a `NamingContext` object to *bind* a name to an object that they offer. Client applications use a `NamingContext` to *resolve* a bound name to an object reference.

A `NamingContextExt` interface is also available which provides methods necessary for using stringified names.

## Names and NameComponent

A `CosNaming::Name` represents an identifier that can be bound to an object implementation or a `CosNaming::NamingContext`. A `Name` is not simply a string of alphanumeric characters; it is a sequence of one or more `NameComponent` structures.

Each `NameComponent` contains two attribute strings, `id` and `kind`. The naming service does not interpret or manage these strings, except to ensure that each `id` and `kind` is unique within a given `NamingContext`.

The `id` and `kind` attributes are strings which uniquely identify the object to which the name is bound. The `kind` member adds a descriptive quality to the name. For example, the name "Inventory.RDBMS" has an `id` member of "Inventory" and a `kind` member of "RDBMS."

### IDL sample 11 IDL Specification for the NameComponent structure

```
module CosNaming
{
    typedef string Istring;
    struct NameComponent {
        Istring id;
        Istring kind;
    };
    typedef sequence<NameComponent> Name;
};
```

The `id` and `kind` attributes of a `NameComponent` must be a character from the ISO 8859-1 (Latin-1) character set, excluding the null character (0x00) and other non-printable characters. Neither of the strings in a `NameComponent` can exceed 255 characters.

Furthermore, the Naming Service does not support `NameComponent` which use wide strings.

### Note

The `id` attribute of a `Name` cannot be an empty string, but the `kind` attribute can be an empty string.

## Name resolution

Your client applications use the `NamingContext` method `resolve` to obtain an object reference, given a logical `Name`. Because a `Name` consists of one or more `NameComponent` objects, the resolution process requires that all of the `NameComponent` structures that make up the `Name` be traversed.

## Stringified names

Because the representation of `CosNaming::Name` is not in a form that is readable or convenient for exchange, a stringified name has been defined to resolve this problem. A stringified name is a one-to-one mapping between a string and a `CosNaming::Name`. If two `CosNaming::Name` objects are equal, then their stringified representations are equal and vice versa. In a stringified name, a forward slash "/" serves as a name component separator; a period "." serves as the `id` and `kind` attributes separator; and a backslash "\" serves as an escape character. By convention a

NameComponent with an empty kind attribute does not use a period (for example, Order).

**Code example 76** Stringified name example

```
"Inprise.Company/Engineering.Department/Printer.Resource"
```

**Note**

In the following examples, NameComponent structures are given in their stringified representations.

## Simple and complex names

A *simple name*, such as Billing, has only a single NameComponent and is always resolved relative to the target naming context. A simple name may be bound to an object implementation or to a NamingContext.

A *complex name*, such as NorthAmerican/ShippingDepartment/Inventory, consists of a sequence of three NameComponent structures. If a complex name consisting of  $n$  NameComponent objects has been bound to an object implementation, then the first  $(n-1)$  NameComponent objects in the sequence must each resolve to a NamingContext, and the last NameComponent object must resolve to an object implementation.

If a Name is bound to a NamingContext, each NameComponent structure in the sequence must refer to a NamingContext.

Code example 77 shows a complex name, consisting of three components and bound to a CORBA object. This name corresponds to the stringified name, NorthAmerica/ SalesDepartment/Order. When resolved within the topmost naming context, the first two components of this complex name resolve to NamingContext objects, while the last component resolves to an object implementation with the logical name "Order."

**Code example 77** Example of a complex name bound to an ORB object

```
...
// Name stringifies to "NorthAmerica/SalesDepartment/Order"
CosNaming::Name_var continentName =
rootNamingContext->to_name("NorthAmerica");
CosNaming::NamingContext_var continentContext =
    rootNamingContext->bind_new_context(continentName);

CosNaming::Name_var departmentName =
continentContext->to_name("SalesDepartment");
CosNaming::NamingContext_var departmentContext =
    rootNamingContext->bind_new_context(departmentName);
CosNaming::Name_var objectName =
departmentContext->to_name("Order");
departmentContext->rebind(objectName,
myPOA->servant_to_reference(managerServant));
...
```

## Running the Naming Service

The VisiBroker-RT for C++ Naming Service is comprised of a set of libraries, header files and sample applications, which are delivered as part of the base VisiBroker distribution.

The Naming Service is a CORBA service which provides a set of interfaces (that is, APIs) to support:

- Creation of Naming Context Servants

A Naming Context Servant is created and then activated on the users POA, to create a Naming Context object reference.

- Registration of a Naming Context Object as the "NameService" root context
- Interfaces for Binding Names to Objects
- Interfaces for Iterating through a naming tree

Installing, configuring and running the Naming Service is described below. Once you have created a Naming Tree, you may browse its contents by using the VisiBroker-RT for C++ Console. For more details see ["Naming Services"](#).

## Integrating the Naming Service into your application

The steps required to integrate the Naming Service with your VisiBrokerRT60 application are very similar to the steps for integrating the VisiBroker ORB libraries. Please refer to ["Step 6: Integrating VisiBroker-RT with Tornado/VxWorks"](#) for details on the process to follow when adding additional libraries to your VisiBrokerRT60 application.

### VisiBrokerRT60 Naming Service libraries

The VisiBroker-RT for C++ Naming Service consist of the following libraries:

**Table 17** VisiBroker Name Service Libraries

Library	Description	Dependencies
libname_c_s.o	This library provides the interfaces for the clients which intend to ONLY use the VisiBroker-RT for C++ Naming Service. If a one of your VisiBrokerRT60 nodes intends to start a Naming Service factory it must include both this library as well as the library libname_fact.o (described below).	liborb_dyn.o and libservicesupport.o  or  liborb_min.o and libservicesupport.o
libname.o	This library provides the interfaces for creating and start ing a VisiBroker-RT for C++ Naming Service Cos Extended Factory Server.  Inclusion of this library also requires the inclusion of the client interface library libname_c_s.o.	liborb_dyn.o and libservicesupport.o  or  liborb_min.o and libservicesupport_min.o and libname_c_s.o

## Compiling and linking programs

- C++ applications that use the Naming Service  
C++ applications that use the naming service need to include the generated file "CosNaming\_c.hh".
- C++ applications that start a Naming Service need to include the following files:
  - CosNaming\_c.hh
  - CosNaming.hh
  - NamingLib.h

## Sample programs

Several example programs that illustrate the use of the Naming Service are provided with VisiBroker-RT for C++. They illustrate the new INS features that are now available with the Naming Service and they can be found in the `<install_location>/VisiBrokerRT60/examples/ins` directory. In addition, a Bank Naming example that illustrates basic usage of the Naming Service can be found in the `<install_location>/VisiBrokerRT60/examples/basic/bank_naming` directory.

Before running the example programs, you must first start the naming service, as described in ["Starting the Naming Service"](#). This will automatically create the initial *"root context"*.

### Warning

If no naming context has been created, a `CORBA::NO_IMPLEMENT` exception will be raised when the client attempts to issue a `CosNaming::NamingContext::_bind`.

## Starting the Naming Service

Starting a VisiBroker-RT for C++ Naming Service, involves performing the following steps.

- Create a Naming Service Servant
- Activate that Servant on your Application POA to Establish the "Initial Naming Service Context" CORBA Object.
- Register the *"Initial Naming Service Context" CORBA Object* with the VisiBroker ORB as the "NameService" service.

These steps are demonstrated as part of the VisiBroker Sample application `<install_location>/VisiBrokerRT60/examples/basic/bank_naming`. The file `start_namingservice.cpp`, shows how to start a VisiBroker Naming Service.

### Code example 78 Starting a VisiBroker Naming Service on an Application POA

```
....

void naming_service(char* objectKey, CORBA::Boolean debug, char *
ORB_options_string)
{
    // -----
    // Initialize the ORB
    // -----

    PortableServer::POA_var application_POA;

    VISTRY
    {
        int default_argc = 2;
        char *default_argv[] = {"-ORBagentport", OSAGENT_PORT};
        char **new_argv;
        int new_argc = VISUtil::stringToArgv(&new_argv,
            default_argv, default_argc,
ORB_options_string);

        // Initialize the ORB
        orb = CORBA::ORB_init(new_argc, new_argv);

        // -----
        // Create an application POA
        // -----
        CORBA::Object_var obj;
        PortableServer::POA_var root_POA;
        PortableServer::POAManager_var POA_mgr;
```

```

    VISIFNOT_EXCEP
    // get a reference to the root POA
    obj = orb->resolve_initial_references("RootPOA");
VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
    root_POA = PortableServer::POA::_narrow(obj);
VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
    // Create our POA manager from the root POA Manager. POA_mgr
    = root_POA->the_POAManager();
    VISEND_IFNOT_EXCEP

// Create POA policies.
    CORBA::PolicyList policies;
    policies.length(2);
VISENFNOT_EXCEP
    policies[(CORBA::ULong)0] =

root_POA->create_lifespan_policy(PortableServer::PERSISTENT);
    VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
    policies[(CORBA::ULong) 1] = root_POA->
create_id_assignment_policy(PortableServer::USER_ID);
    VISEND_IFNOT_EXCEP

// Create a application poa.
// exceptions: AdapterAlreadyExist Invalid Policy

// Thread Policy: ORB_CTRL_MODEL
// Lifespan Policy: PERSISTENT
// Object Id Uniqueness Policy: UNIQUE_ID
// Id Assignment Policy: USER_ID
// Server Retention Policy: RETAIN
// Request Processing Policy: USE_ACTIVE_OBJECT_MAP_ONLY
// Implicit Activation Policy NO_IMPLICIT_ACTIVATION

VISENFNOT_EXCEP
application_POA = root_POA->
    create_POA("application_POA",POA_mgr, policies);
VISEND_IFNOT_EXCEP

VISENFNOT_EXCEP
// Activate the Market POA
POA_mgr->activate();
    VISEND_IFNOT_EXCEP

// -----
// Create and start Name Service
// -----

PortableServer::ObjectId_var namingContextId;
POA_CosNaming::NamingContext* namingContextServant;

// Create a namingContext servant. namingContextServant =
NamingLib::create_RootContextNamingServant(objectKey,
application_POA);

    VISIFNOT_EXCEP
    // Create a object Id for naming context servant.
namingContextId =
PortableServer::string_to_ObjectId(objectKey);
VISEND_IFNOT_EXCEP

    CosNaming::NamingContext_var root_context;

// Activate the servant with the ID on Event Channel POA.
// exceptions: ServantAlreadyActive, ObjectAlreadyActive,
// and WrongPolicy

```

```

        VISIFNOT_EXCEP
        application_POA->activate_object_with_id(namingContextId,
namingContextServant); VISEND_IFNOT_EXCEP

        VISIFNOT_EXCEP
        obj = application_POA->id_to_reference(namingContextId);
VISEND_IFNOT_EXCEP

        VISIFNOT_EXCEP
        root_context = CosNaming::NamingContext::_narrow(obj);
VISEND_IFNOT_EXCEP

        CORBA::String_var str;
        VISIFNOT_EXCEP
        str = orb->object_to_string(obj);
        VISEND_IFNOT_EXCEP

        cout << "\n\nInitial NamingContext Registered\n" << str <<
endl;

        VISIFNOT_EXCEP
        // Register a namingContext servant with URL Locator.

        SupportServices::instance()-
>register_service_object(objectKey, obj);
        VISEND_IFNOT_EXCEP
    }

    VISCATCH(CORBA::Exception, e)
    {
        cerr << "exception is " << e << endl;

        taskSuspend(0);
    }
    VISEND_CATCH

    return;
}

```

## Bootstrapping a Naming Service

There are three ways to start a client application so that it can obtain an initial object reference to a specified Naming Service. You can use the following two options when calling `ORB_init` from your Naming Service Client application:

- `ORBInitRef`
- `ORBDefaultInitRef`

### Calling `resolve_initial_references`

The new Naming Service provides a simple mechanism by which the `resolve_initial_references` method can be configured to return a common naming context. You use the `resolve_initial_references` method which returns the root context of the Naming Server to which the client program connects.

Three simple examples will illustrate how to use these three options. Suppose there are three VisiBroker-RT for C++ Naming Services running on the host `TestHost`: `ns1`, `ns2`, and `ns3`. And there are three server applications: `sr1`, `sr2`, `sr3`, each running on a different port (20001, 20002, and 20003) on the host `TestHost`. Server `sr1` binds itself in `ns1`, `sr2` in `ns2`, and `sr3` in `ns3`. Additionally `sr3` has a naming tree hierarchy of `<NorthAmerica/ShippingDepartment/Inventory>`.

**Code example 79** Code snippet showing how to obtain the root naming context

```
...
CORBA::ORB_ptr orb = CORBA::ORB_init(argv, argc, NULL);
CORBA::Object var rootObj =
orb->resolve_initial_references("NameService");
...
```

## Using -ORBInitRef

You can use either the `corbaloc` or `corbaname` **URL naming schemes** to specify which VisiBroker-RT for C++ Naming Service you want to bootstrap.

## Using a corbaloc URL

If you want to bootstrap into Naming Service `ns2` by using the `corbaloc locator`, then you should start your client application and obtain the root context of `ns2` by calling the `resolve_initial_references` method on an ORB reference inside your client application as illustrated in [Code example 80](#).

**Code example 80** BootStrapping to the Naming Service Using `corbaloc` URL

```
...
void do_corba(char * ORB_options_string)
{
    char *nameServiceLocator_URL =
        "NameService=corbaloc::TestHost:20002/ns2";
    // OR
    // char *nameServiceLocator_URL =
        "NameService=corbaloc:iiop:TestHost:20002/ns2";

    int default_argc = 2;
    char *default_argv[] ={"-ORBInitRef",nameServiceLocator};

    char **new_argv;
    int new_argc = VISUtil::stringToArgv(&new_argv, default_argv,
        default_argc, ORB_options_string);

    /* -----*/
    /* Call ORB_init, obtain the Root POA, and bootstrap to */
    /* Name Service initial root context. */
    /* -----*/

    VISTRY
    {
        // Initialize the ORB.
        orb = CORBA::ORB_init(new_argc, new_argv);

        VISIFNOT_EXCEP
        // Get a reference to the Naming Service root_context obj
        = CosNaming::NamingContext::_narrow(
        orb->resolve_initial_references("NameService"));

        VISEND_IFNOT_EXCEP

        VISIFNOT_EXCEP
        CORBA::String_var str = orb->object_to_string(obj);
        VISEND_IFNOT_EXCEP

        VISIFNOT_EXCEP
        rootContext = CosNaming::NamingContext::_narrow(obj);
        VISEND_IFNOT_EXCEP
    }
    VISCATCH(CORBA::Exception, e)
    {
        cerr << e << endl;
        return;
    }
    VISEND_CATCH
}
```

```

    return;
}

```

## Note

This example will work only if there is a Naming Service Object implementation running on an ORB instance at host "TestHost" and port 20002.

## Using a corbaname URL

If you want to bootstrap into Naming Service ns3 by using the *corbaname locator*, then you should start your client application and obtain the root context of ns3 by calling the *resolve\_initial\_references* method on an ORB reference inside your client application as illustrated in [Code example 81](#).

**Code example 81** Bootstrapping to the Naming Service Using corbaname URL

```

...
void do_corba(char * ORB_options_string)
{
    char *nameServiceLocator_URL =
        "NameService=corbaname:TestHost:20003#NorthAmerica/
        ShippingDepartment'";
    // OR
    // char *nameServiceLocator_URL =
    //     "NameService=corbaname:iiop:TestHost:20003#NorthAmerica/
    //     ShippingDepartment'";

    int default_argc = 2;
    char *default_argv[] ={"-ORBInitRef",nameServiceLocator};

    char **new_argv;
    int new_argc = VISUtil::stringToArgv(&new_argv, default_argv,
        default_argc, ORB_options_string);

    /*-----*/
    /* Call ORB_init, obtain the Root POA, and bootstrap to */
    /* Name Service initial root context.*/
    /*-----*/
    VISTRY
    {
        // Initialize the ORB.
        orb = CORBA::ORB_init(new_argc, new_argv);

        VISIFNOT_EXCEP
        // Get a reference to the Naming Service root_context
        obj = CosNaming::NamingContext::_narrow(
            orb->resolve_initial_references("NameService"));

        CORBA::String_var str = orb->object_to_string(obj);

        rootContext = CosNaming::NamingContext::_narrow(obj);

        VISEND_IFNOT_EXCEP
    }
    VISCATCH(CORBA::Exception, e)
    {
        cerr << e << endl;
        return;
    }
    VISEND_CATCH
    return;
}

```

## Note

This example will work only if there is a Naming Service Object implementation running on an ORB instance at port 20003.

## -ORBDefaultInitRef

You can use either an `corbaloc` or `corbaname` URL to specify which VisiBroker-RT for C++ Naming Service you want to bootstrap.

### Using -ORBDefaultInitRef with an corbaloc URL

If you want to bootstrap into `ns2`, then start your client program as:

**Code example 82** BootStrapping to the Naming Service Using -ORBDefaultInitRef with the corbaloc URL

```
...
void do_corba()
{
    char *nameServiceLocator =
        "NameService=corbaloc://TestHost:20002/";

    int default_argc = 2;
    char *default_argv[]
    = {"-ORBDefaultInitRef", nameServiceLocator};

    char **new_argv;
    int new_argc = VISUtil::stringToArgv(&new_argv, default_argv,
        default_argc, ORB_options_string);

    /*-----*/
    /* Call ORB_init, obtain the Root POA, and bootstrap to Name*/
    /* Service initial root context. */
    /*-----*/
    VISTRY
    {
        // Initialize the ORB.
        orb = CORBA::ORB_init(argc, argv);

        VISIFNOT_EXCEP
        // Get a reference to the Naming Service root_context
        obj = CosNaming::NamingContext::narrow(
            orb->resolve_initial_references("NameService"));
    }
    ...
}
```

### Using -ORBDefaultInitRef with corbaname

The combination of `-ORBDefaultInitRef` and `corbaname` works differently from what is expected. If `-ORBDefaultInitRef` is specified, a slash and the stringified object key `"NameService"` is always appended to the `corbaname`. For example, if the URL is `corbaname://TestHost:20002`, then by specifying `-ORBDefaultInitRef`, `resolve_initial_references` will result in a new URL: `corbaname://TestHost:20002/NameService`.

## NamingContext

This object is used to contain and manipulate a list of names that are bound to ORB objects or to other `NamingContext` objects. Client applications use this interface to `resolve` or `list` all of the names within that context. Object implementations use this object to `bind` names to object implementations or to bind a name to a `NamingContext` object.

The following sample shows the IDL specification for the `NamingContext`.

**IDL sample 12** Specification for the `NamingContext` interface

```
module CosNaming {
    interface NamingContext {
```

```

void bind(in Name n, in Object obj)
    raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
void rebind(in Name n, in Object obj)
    raises(NotFound, CannotProceed, InvalidName);
void bind_context(in Name n, in NamingContext nc)
    raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
void rebind_context(in Name n, in NamingContext nc)
    raises(NotFound, CannotProceed, InvalidName);
Object resolve(in Name n)
    raises(NotFound, CannotProceed, InvalidName);
void unbind(in Name n)
    raises(NotFound, CannotProceed, InvalidName); NamingContext
new_context();
NamingContext bind_new_context(in Name n)
    raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
void destroy()
    raises(NotEmpty);
void list(in unsigned long how_many,
    out BindingList bl,
    out BindingIterator bi);
};
};

```

## NamingContextExt

The `NamingContextExt` interface, which extends `NamingContext`, provides the operations required to use stringified names and URLs.

**IDL sample 13** Specification for the `NamingContextExt` interface

```

module CosNaming {
    interface NamingContextExt : NamingContext {
        typedef string StringName;
        typedef string Address;
        typedef string URLString;

        StringName to_string(in Name n)
            raises(InvalidName);
        Name to_name(in StringName sn)
            raises(InvalidName);

        exception InvalidAddress {};
        URLString to_url(in Address addr, in StringName sn)
            raises(InvalidAddress, InvalidName);
        Object resolve_str(in StringName n)
            raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
    };
};

```

## Default naming contexts

A client application can specify a *default naming context*, which is the naming context that the application will consider to be its *root* context. Note that the default naming context is the *root* only in relation to this client application and, in fact, it may be contained by another context.

### Obtaining the default context

The ORB method `resolve_initial_references` can be used by a client application to obtain the default naming context. The default naming context must have been specified by passing the `ORBInitRef` command-line argument when the client application was started. [Code example 79](#) shows how a C++ client application could invoke this method.

## Binding a name in C++

The bind example uses two simple interfaces, shown in [IDL sample 14](#), which offer two methods `Bank::Account::balance` and `Bank::AccountManager::open`. This definition is found in the `Bank.idl` file.

**IDL sample 14** IDL specification for the `Bank::Account` and `Bank::AccountManager` interfaces

```
module Bank {
    interface Account {
        float balance();
    };
    interface AccountManager {
        Account open(in string name);
    };
};
```

The `naming_server.cpp` file for the `../examples/basic/bank_naming` example contains the server-side code, which create a `AccountManager` CORBA Object and binds a name to the Object.

**Code example 83** `bank_naming/naming_server.cpp`, object server binding a name to an ORB object

```
#include <vxWorks.h>
#include "CosNaming_c.hh"
#include "bankImpl.h"

extern CORBA::ORB_var orb;

extern PortableServer::POA_var rootPOA;
extern CosNaming::NamingContext_var rootContext;

...

void bank_server()
{
    VISTRY {

        CORBA::PolicyList policies;
        policies.length(1);

        policies[(CORBA::ULong)0] =
            rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);

        // get the POA Manager
        PortableServer::POAManager_var poa_manager;

        VISIFNOT_EXCEP
        poa_manager = rootPOA->the_POAManager();
        VISEND_IFNOT_EXCEP

        PortableServer::POA_var myPOA;

        VISIFNOT_EXCEP
        // Create myPOA with the right policies
        myPOA = rootPOA->create_POA("bank_agent_poa", poa_manager,
            policies);
        VISEND_IFNOT_EXCEP

        // Create the servant
        AccountManagerImpl *managerServant = new AccountManagerImpl;

        PortableServer::ObjectId_var managerId;

        VISIFNOT_EXCEP
        // Decide on the ID for the servant
        managerId = PortableServer::string_to_ObjectId("BankManager");
        VISEND_IFNOT_EXCEP

        VISIFNOT_EXCEP
```

```

// Activate the servant with the ID on myPOA
myPOA->activate_object_with_id(managerId, managerServant);
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
// Activate the POA Manager
poa_manager->activate();
VISEND_IFNOT_EXCEP

CORBA::Object_var reference;

VISIFNOT_EXCEP
reference = myPOA->servant_to_reference(managerServant);
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
// Associate the bank manager with the name at the root
context
CosNaming::Name name;
name.length(1);

name[0].id = (const char *) "BankManager";
name[0].kind = (const char *) "";
rootContext->rebind(name, reference);
VISEND_IFNOT_EXCEP

cout << reference << " is ready" << endl;

}
VISCATCH(CORBA::Exception, e) {
    cerr << e << endl;
    taskSuspend(0);
}
VISEND_CATCH

return;
}

```

## Resolving a name in C++

The following code sample shows the client program that use the Naming Service to resolve a Name Binding to an Object Reference.

**Code example 84** banke\_naming/client.C file for resolving an ORB object by name

```

#include <vxWorks.h>
#include "CosNaming_c.hh"
#include "bank_c.hh"

extern CosNaming::NamingContext_var rootContext;

....

void bank_client(void)
{
    VISTRY {

        // Locate an account manager through the Naming Service
        CosNaming::Name name;
        name.length(1);
        name[0].id = (const char *) "BankManager";
        name[0].kind = (const char *) "";
        Bank::AccountManager_var manager =
            Bank::AccountManager::_narrow(rootContext->resolve(name));

        // Set the account name
        const char* account_name = "Jack B. Quick";
        Bank::Account_var account;
        VISIFNOT_EXCEP
        // Request the account manager to open a named account
        account = manager->open(account_name); VISEND_IFNOT_EXCEP
    }
}

```

```
CORBA::Float balance;
VISIFNOT_EXCEP
// Get the balance of the account
balance = account->balance(); VISEND_IFNOT_EXCEP

// Print out the balance
cout << "The balance in " << account_name << "'s account is $"
<< balance << endl;
}
VISCATCH(CORBA::Exception, e) {
    cerr << e << endl;
    taskSuspend(0);
}
VISEND_CATCH

return;
}
```



# Using the Event Service

This chapter describes the VisiBroker-RT for C++ Event Service.

## NOTE

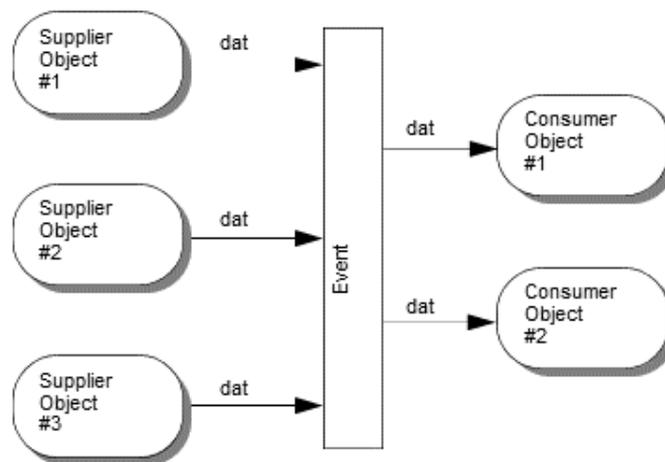
The libraries:

- libservicesupport.a
- libevchn\_c\_s.a
- libevchn.a

are required when building a VisiBrokerRT60 application to support use of the VisiBroker Event Service. For a description of all the libraries provided by the VisiBroker-RT for C++ product please refer to “[Step 6: Integrating VisiBroker-RT with Tornado/VxWorks](#)”.

## Overview

The Event Service package provides a facility that de-couples the communication between objects. It provides a *supplier-consumer* communication model that allows multiple *supplier objects* to send data asynchronously to multiple *consumer objects* through an event channel. The supplier-consumer communication model allows an object to communicate an important change in state, such as a disk running out of free space, to any other objects that might be interested in such an event.



**Figure 25** Supplier-Consumer communication model

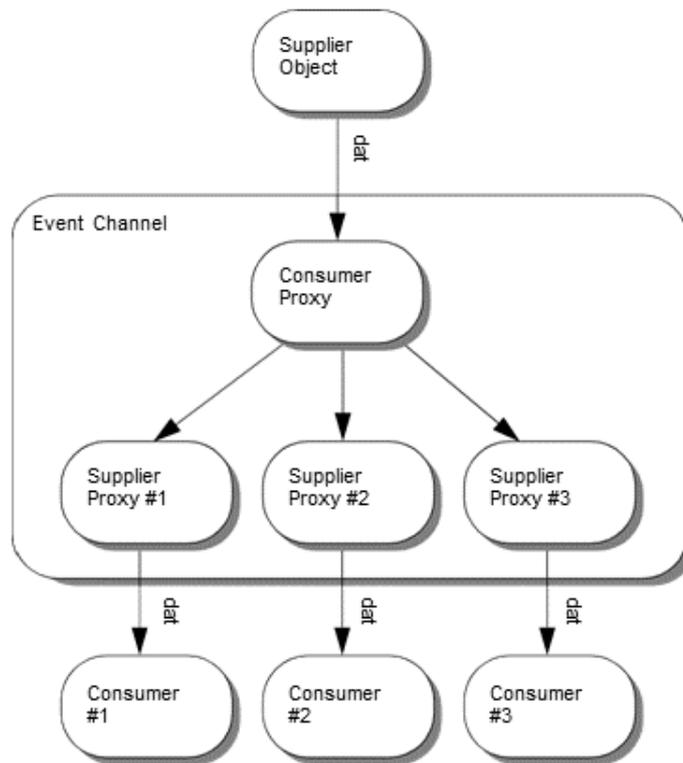
Figure 25 shows three supplier objects communicating through an event channel with two consumer objects. The flow of data into the event channel is handled by the supplier objects, while the flow of data out of the event channel is handled by the consumer objects. If the three suppliers shown in Figure 25 each send one message every second, then each consumer will receive three messages every second and the event channel will forward a total of six messages per second.

The event channel is both a consumer of events and a supplier of events. The data communicated between suppliers and consumers are represented by the `Any` class, allowing any CORBA type to be passed in a type safe

manner. Supplier and consumer objects communicate through the event channel using standard CORBA requests.

## Proxy consumers and suppliers

Consumers and suppliers are completely de-coupled from one another through the use of *proxy objects*. Instead of directly interacting with each other, they obtain a proxy object from the `EventChannel` and communicate with it. Supplier objects obtain a *consumer proxy* and consumer objects obtain a *supplier proxy*. The `EventChannel` facilitates the data transfer between consumer and supplier proxy objects. Figure 26 shows how one supplier can distribute data to multiple consumers.



**Figure 26** Consumer and supplier proxy objects

### Note

The event channel is shown in Figure 26 as a separate process, but it may also be implemented as part of the supplier object's process. See "Starting the Event Service" for more information.

## OMG common object services specification

The VisiBroker-RT for C++ Event Service implementation conforms to the OMG Common Object Services Specification, with the exception of these two items,

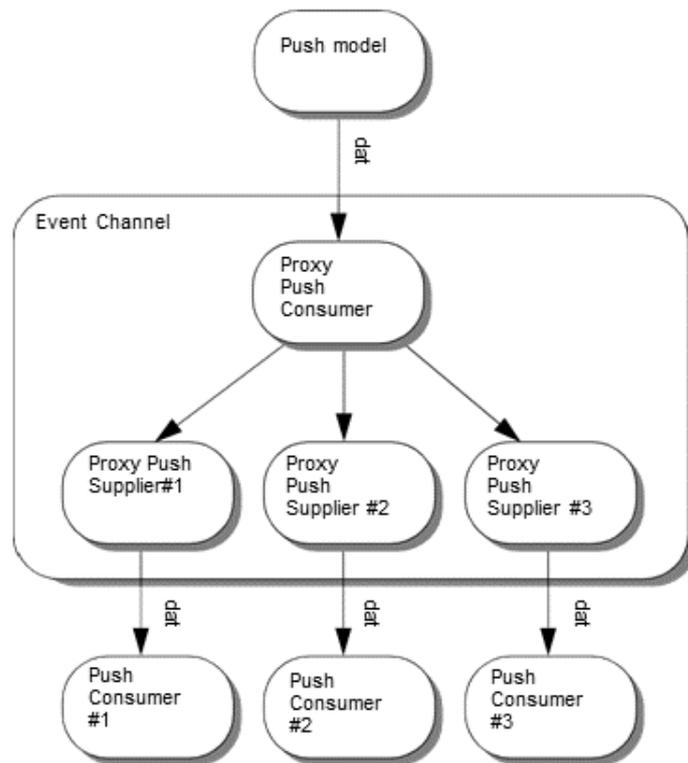
- The VisiBroker-RT for C++ Event Service only supports generic events. There is currently no support for typed events in the VisiBroker-RT for C++ Event Service.

- The VisiBroker-RT for C++ Event Service offers no confirmation of the delivery of data to either the event channel or to consumer applications. TCP/IP is used to implement the communication between consumers, suppliers and the event channel and this provides reliable delivery of data to both the channel and the consumer. However, this does not guarantee that all of the data that is sent will actually be processed by the receiver.

## Communication models

The event service provides both a *pull* and *push* communication model for suppliers and consumers. In the *push model*, supplier objects control the flow of data by *pushing* it to consumers. In the *pull model*, consumer objects control the flow of data by *pulling* data from the supplier.

The `EventChannel` insulates suppliers and consumers from having to know which model is being used by other objects on the channel. This means that a pull supplier can provide data to a push consumer and a push supplier can provide data to a pull consumer.



**Figure 27** Push model

### Note

The event channel is shown in [Figure 27](#) as a separate process, but it may also be implemented as part of the supplier object's process. See "[Starting the Event Service](#)" for more information.

## Push model

The *push model* is the more common of the two communication models. An example use of the push model is a supplier that monitors available free

space on a disk and notifies interested consumers when the disk is filling up. The push supplier sends data to its `ProxyPushConsumer` in response to events that it is monitoring.

The `PushConsumer`'s `push` method is invoked by the `EventChannel` upon the `ProxyPushSupplier` retrieving data from the `EventChannel`. The `EventChannel` facilitates the transfer of data from the `ProxyPushSupplier` to the `ProxyPushConsumer`.

Figure 27 shows a push supplier and its corresponding `ProxyPushConsumer` object. It also shows three push consumers and their respective `ProxyPushSupplier` objects.

## Pull model

In the *pull* model, the event channel regularly pulls data from a supplier object, puts the data in a queue, and makes it available to be *pulled* by a consumer object. An example of a pull consumer would be one or more network monitors that periodically poll a network router for statistics.

The pull supplier implements a "pull" method which is invoked by the `ProxyPullConsumer`. The `ProxyPullConsumer` spends most of its time in an event loop invoking the pull supplier's "pull" method. The pull consumer requests data from the `ProxyPullSupplier` when it is ready for more data. The `EventChannel` pulls data from the supplier to a queue and makes it available to the `ProxyPullSupplier`.

Figure 28 shows a pull supplier and its corresponding `ProxyPullConsumer` object. It also shows three pull consumers and their respective `ProxyPullSupplier` objects.

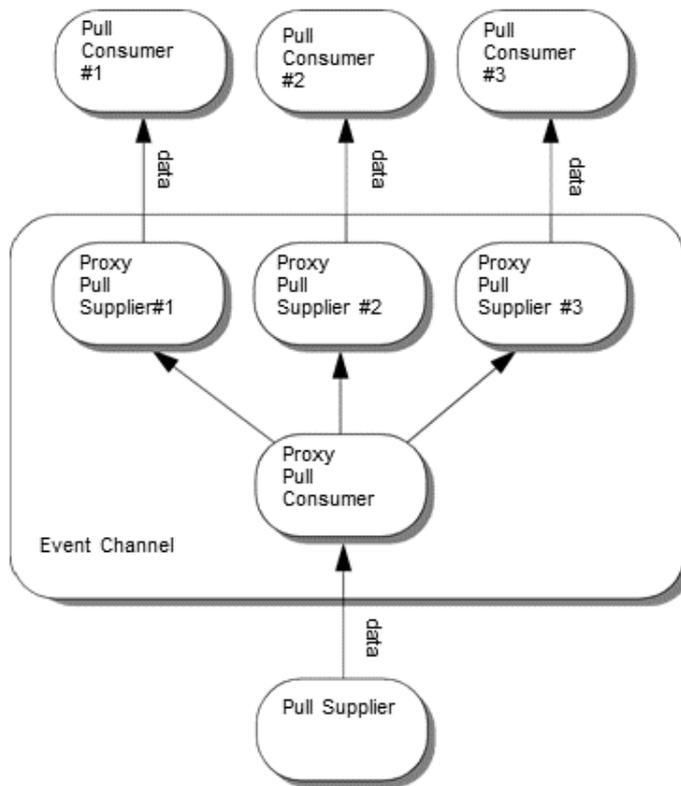


Figure 28 Pull model

## Note

The event channel is shown in [Figure 28](#) as a separate process, but it may also be implemented as part of the supplier object's process.

# Using event channels

When an `EventChannel` is first created, it has no suppliers or consumers. A supplier or consumer connects to and uses an event channel by following these steps:

- 1 Connect to the `EventChannel`.
- 2 Obtain an administrative object from the channel and use it to obtain a proxy object.
- 3 Connect to the proxy object.
- 4 Begin transferring or receiving data.

The methods used for these steps vary, depending on whether the object being connected is a supplier or a consumer, and on the communication model being used. [Table 18](#) shows the appropriate methods for suppliers and [Table 19](#) shows the methods for consumers.

**Table 18** Connecting Suppliers to an EventChannel

Steps	Push supplier	Pull supplier
Bind to the Event Channel.	<code>CosEventChannelAdmin::EventChannel::_narrow(orb::resolve_initial_references("EventService"))</code>	<code>CosEventChannelAdmin::EventChannel::_narrow(orb::resolve_initial_references("EventService"))</code>
Get a Supplier Admin.	<code>EventChannel::for_suppliers()</code>	<code>EventChannel::for_suppliers()</code>
Get a consumer proxy.	<code>SupplierAdmin::obtain_push_consumer()</code>	<code>SupplierAdmin::obtain_pull_consumer()</code>
Add the supplier to the Event Channel.	<code>ProxyPushConsumer::connect_push_supplier()</code>	<code>ProxyPullConsumer::connect_pull_supplier()</code>
Data transfer	<code>ProxyPushConsumer::push()</code>	Implements <code>pull()</code> and <code>try_pull()</code>

**Table 19** Connecting Consumers to an EventChannel

Steps	Push consumer	Pull consumer
Bind to the Event Channel.	<code>CosEventChannelAdmin::EventChannel::_narrow(orb::resolve_initial_references("EventService"))</code>	<code>CosEventChannelAdmin::EventChannel::_narrow(orb::resolve_initial_references("EventService"))</code>
Get a Consumer Admin.	<code>EventChannel::for_consumers()</code>	<code>EventChannel::for_consumers()</code>
Obtain a supplier proxy.	<code>ConsumerAdmin::obtain_push_supplier()</code>	<code>ConsumerAdmin::obtain_pull_supplier()</code>
Add the consumer to the Event Channel.	<code>ProxyPushSupplier::connect_push_consumer()</code>	<code>ProxyPushSupplier::connect_pull_consumer()</code>
Data transfer	Implements <code>push()</code>	<code>ProxyPushSupplier::pull()</code> and <code>try_pull()</code>

## Example push supplier and consumer

This section describes the example push supplier and consumer applications. When executed, the supplier application prompts the user to enter data and then *pushes* the data to the consumer application. The consumer application receives the data and writes it to the screen.

The push supplier application is implemented in the `factoryPushSupplier.cpp` file and the push consumer is implemented in the `factoryPushConsumer.cpp` file. These files can be found in the `<install_location>/VisiBrokerRT60/examples/events` directory on your system.

### Deriving a PushSupplier class

The first step in implementing a supplier is to derive our own `PushModel` class from the `PushSupplier` interface, shown in the following code sample.

#### Code example 85 PushSupplier interface

```
module CosEventComm {
    interface PushSupplier {
        void disconnect_push_supplier();
    };
};
```

This example shows the `PushModel` class, implemented in C++. The `disconnect_push_supplier` method is called by the `EventChannel` to disconnect the supplier when the channel is being destroyed. This implementation simply prints out a message and exits. If the `PushModel` object were persistent, this method might also call `deactivate_obj` to deactivate the object.

#### Code example 86 PushModel class

```
// PushModel.C

#include "CosEventComm_s.hh"
#include "CosEventChannelAdmin_c.hh"

class PushModel : public POA_CosEventComm::PushSupplier, public
VISThread {
public:
    void disconnect_push_supplier() {
        cout << "Model::disconnect_push_supplier()" << endl;
        VISTRY {
            PortableServer::ObjectId var objId =
                PortableServer::string_to_ObjectId("PushModel");
            _myPOA->deactivate_object(objId);
        }
        VISCATCH(const CORBA::Exception, e) {
            cout << e << endl;
        }
    }
};
```

### Implementing the PushSupplier

The Push Supplier performs the following:

- 1 Binds to the Event Channel Factory
- 2 Does a lookup by name for the desired EventChannel
- 3 Creates a Push Supplier
- 4 Connects the Push Supplier to the Event Channel

## 5 Goes into a loop to Push Data to the Event Channel

### Code example 87 Complete implementation for a sample push supplier

```
// factorypushSupplier.C #include <vxWorks.h>

#include "CosEventComm_s.hh"
#include "ChannelLib.h"

/*-----*/
/* Forward Declarations*/
/*-----*/
extern "C" intsysClkRateGet (void);

/*-----*/
/* Export Variable Declarations*/
/*-----*/
extern CORBA::ORB_var orb;
extern PortableServer::POA_var ec_POA;

class PushSupplierImpl : public POA_CosEventComm::PushSupplier
{
public:
    void disconnect_push_supplier()
    {
        cout << "disconnect_push_supplier()" << endl;
        _alive = 0;
    }

    PushSupplierImpl() :
        POA_CosEventComm::PushSupplier() { _alive = 1; }

    CORBA::Boolean Alive() { return _alive; }

private:
    CORBA::Boolean _alive;
};

static void factorypushSupplier(char * name, char*
factoryname);

void start_pushSupplier(char* channelName, char* factName)
{
    char * taskName = "PUSHSUP";
    int Prio = 100;
    int option = VX_FP_TASK;
    int stackSize = 20000;

    taskSpawn(taskName,
        Prio,
        option,
        stackSize,
        (FUNCPTR)factorypushSupplier,
        (int)channelName,
        (int)factName,
        0,0,0,0,0,0,0,0);
}

void factorypushSupplier(char* name, char* factoryname)
{
    CosEventChannelAdmin::EventChannelFactory_var factory;

    VISTRY
    {
        // Get the Channel Id from user's supplied name.
        PortableServer::ObjectId_var factId =
        PortableServer::string_to_ObjectId(factoryname);

        // Bind to Event Factory by giving the full POA name
        // and the Object ID..
        factory = CosEventChannelAdmin::EventChannelFactory::
        _bind("/ef_POA", factId);
    }
}
```

```

CosEventChannelAdmin::EventChannel_var channel;

VISIFNOT_EXCEP
// Bind to the Event Channel
channel = factory->lookup_by_name(name);
VISEND_IFNOT_EXCEP

CosEventChannelAdmin::SupplierAdmin_var for_supplier;
CosEventChannelAdmin::ProxyPushConsumer_var
proxyPushConsumer;

VISIFNOT_EXCEP
// Obtain Supplier Administrator
for_supplier = channel->for_suppliers();
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
// Obtain push_consumer proxy
proxyPushConsumer = for_supplier->obtain_push_consumer();
VISEND_IFNOT_EXCEP

PortableServer::ObjectId_var supplierId;
CosEventComm::PushSupplier_var pushSupplierObject;
PushSupplierImpl* pushSupplier = new PushSupplierImpl();

VISIFNOT_EXCEP
// Create a object Id for Supplier servant.
supplierId =
PortableServer::string_to_ObjectId("mypushSupplier");
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
// Activate the servant with the ID on Event Channel POA.
// exceptions: ServantAlreadyActive, ObjectAlreadyActive,
// and WrongPolicy
ec_POA->activate_object_with_id(supplierId, pushSupplier);
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
pushSupplierObject = CosEventComm::PushSupplier::
_narrow(ec_POA->id_to_reference(supplierId));
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
// Connect my push Supplier to the proxy push Consumer
proxyPushConsumer->
connect_push_supplier(pushSupplierObject);
VISEND_IFNOT_EXCEP

char string[1024]; CORBA::Any any;

while(pushSupplier->Alive())
{
    cout << "(Type 'q' to quit)-> " << flush;
    cin >> string;

    if(!strcmp(string,"q"))
    {
        // Disconnect all suppliers and
        // consumer and destroy channel channel->destroy();
    } else {

        any <<= (const char *)string;
    }

    VISTRY
    {
        proxyPushConsumer->push(any);
    }
    VISCATCH(CosEventComm::Disconnected, e)
    {
        cerr << "Disconnected" << endl;
        break;
    }
}

```

```

        }
        VISEND_CATCH;

        if (!strcmp(string, "q")) { break;}
    }
}
VISCATCH(CORBA::Exception, err)
{
    cerr << "Error: " << err << endl << flush;
    taskSuspend(0);
}
VISEND_CATCH

return;
}

```

## Deriving a PushConsumer class

The following code sample shows the complete implementation of the Push Consumer Class which is derived from the `PushConsumer` interface, shown in [Code example 87](#).

### Code example 88 PushConsumer interface

```

module CosEventComm {
    exception Disconnected();
    interface PushConsumer {
        void push(in any data) raises(Disconnected);
        void disconnect_push_consumer();
    };
};

```

The push method receives an Any type and attempts to convert it to a string and print it. The `disconnect_push_supplier` method is called by the `EventChannel` to disconnect the consumer when the channel is destroying itself.

## Implementing the PushConsumer

Just like the Push Supplier, the Push Consumer needs to acquire a handle to the Event Channel Factory, find the named Event Channel, and connect a PushConsumer to the Event Channel. The `factoryPushConsumer` sample application performs the following:

- 1 Binds to the Event Channel Factory
- 2 Does a lookup by name for the desired EventChannel
- 3 Creates a Push Consumer
- 4 Connects the Push Consumer to the Event Channel

### Code example 89 Complete implementation for a sample push consumer

```

// factorypushConsumer.C
#include <vxWorks.h>

#include "CosEventComm_s.hh"
#include "CosEventChannelAdmin_c.hh"

/*-----*/
/* Forward Declarations */
/*-----*/
extern "C" int sysClkRateGet (void);

/*-----*/
/* Export Variable Declarations */
/*-----*/
extern CORBA::ORB_var orb;

```

```

extern PortableServer::POA_var ec_POA;

class PushConsumerImpl : public POA_CosEventComm::PushConsumer
{
public:

void push(const CORBA::Any& any)
{
char* string;
if(any >>= string)
{
cout << string << endl;
}
else
{
cout << "Non string: " << any << endl;
}
}

void disconnect_push_consumer()
{
cout << "disconnect_push_consumer()" << endl;
}

private:
};

static void factorypushConsumer(char * name, char* factoryname);

void start_pushConsumer(char* channelName, char* factName)
{
char * taskName = "PUSHCONS";
int Prio = 100;
int option = VX_FP_TASK;
int stackSize = 20000;

taskSpawn(taskName,
Prio,
option,
stackSize,
(FUNCPTR)factorypushConsumer,
(int)channelName,
(int)factName,
0,0,0,0,0,0,0,0);
}

void factorypushConsumer(char* name, char* factoryname)
{
CosEventChannelAdmin::EventChannelFactory_var factory;

VISTRY
{
// Get the Channel Id from user's supplied name.
PortableServer::ObjectId_var factId =
PortableServer::string_to_ObjectId(factoryname);

// Bind to Event Factory by giving the full POA name
// and the Object ID..

factory = CosEventChannelAdmin::EventChannelFactory::_bind
("/ ef_POA", factId);

CosEventChannelAdmin::EventChannel_var channel;
VISIFNOT_EXCEP
channel = factory->lookup_by_name(name);
VISEND_IFNOT_EXCEP

CosEventChannelAdmin::ConsumerAdmin_var for_consumer;
CosEventChannelAdmin::ProxyPushSupplier_var
proxyPushSupplier;
VISIFNOT_EXCEP
//obtain Consumer Admin
for_consumer = channel->for_consumers();
}
}

```

```

        VISEND_IFNOT_EXCEPT

        VISIFNOT_EXCEPT
        proxyPushSupplier = for_consumer->obtain_push_supplier();
        VISEND_IFNOT_EXCEPT

        PortableServer::ObjectId_var consumerId;
        CosEventComm::PushConsumer_var pushConsumerObject;
        PushConsumerImpl* pushConsumer = new PushConsumerImpl();

        VISIFNOT_EXCEPT
        // Create a object Id for Supplier servant. consumerId =
        PortableServer::string_to_ObjectId("mypushConsumer");
        VISEND_IFNOT_EXCEPT

        VISIFNOT_EXCEPT
        // Activate the servant with the ID on Event Channel POA.
        // exceptions: ServantAlreadyActive, ObjectAlreadyActive,
        // and WrongPolicy
        ec_POA->activate_object_with_id(consumerId, pushConsumer);
        VISEND_IFNOT_EXCEPT

        VISIFNOT_EXCEPT
        pushConsumerObject = CosEventComm::PushConsumer::

        _narrow(ec_POA->id_to_reference(consumerId));
        VISEND_IFNOT_EXCEPT

        VISIFNOT_EXCEPT
        //connect to push Consumer
        proxyPushSupplier
        ->connect_push_consumer(pushConsumerObject);
        VISEND_IFNOT_EXCEPT
    }
    VISCATCH(CORBA::Exception, err)
    {
        cerr << "Error: " << err << endl << flush;
        taskSuspend(0);
    }
    VISEND_CATCH

    return;
}

```

## Starting the Event Service

The VisiBroker-RT for C++ Event Service is comprised of a set of libraries, header files and sample applications, which are delivered as part of the base VisiBroker distribution.

The Event Service is a CORBA application which provides a set of interfaces (APIs) to support:

- 1 Creation of Named Event Channels
- 2 Creation of Suppliers and Consumers and "connecting" them over a specified Event Channel
- 3 Management of the Event Channel
- 4 Management of Event Channels through an Event Channel Factory interface

## Installing the Event Service

The Event Service is installed automatically when you install VisiBroker-RT for C++ 6.0.

**Note that the existing manual refers to v4.0! I presume this is still correct for 6.0?**

## Integrating the Naming Service into your application

The steps required to integrate the Event Service into your VisiBrokerRT60 application are very similar to the steps for integrating the VisiBroker ORB libraries. Please refer to “[Step 6: Integrating VisiBroker-RT with Tornado/VxWorks](#)” for details on the process to follow when adding additional libraries to your VisiBrokerRT60 application.

### VisiBroker Event Service libraries

The VisiBroker-RT for C++ Event Service consist of the following libraries:

**Table 20** VisiBroker Event Service Libraries

Library	Description	Dependencies
libevchn_c_s.o	This library provides the interfaces for the clients which intend ONLY to use an already existing VisiBroker-RT for C++ Event Service channel and/or factory. If a one of your VisiBrokerRT60 nodes intends to start a Event Service channel and/or factory it must include both this library as well as the library libevchn.o (described below)	liborb_dyn.o o and libservicesupport.o  or  liborb_min.o and libservicesupport.o
libevchn.o	This library provides the interfaces for creating and starting Visi- Broker-RT for C++ Event Service channels and/or factories.  Inclusion of this library also requires the inclusion of the client interface library libevchn_c_s.o.	liborb_dyn.o o and libservicesupport.o  or  liborb_min.o and libservicesupport.o  and libevchn_c_s.o

### VisiBroker Event Service “munched” libraries

Alternatively “**munched**” versions of the Event Service libraries are also delivered as part of the VisiBroker-RT for C++ Event Service distribution. These “**munched**” versions **are** made available for those users who prefer to dynamically load the Event Service libraries from the Tornado WindShell:

**Code example 90** Loading the Event Service libraries from the Tornado WindShell.

```
--> ld < libservicesupport_munched.o  
--> ld < libevchn_c_s_munched.o  
--> ld < libevchn_munched.o
```

## Setting the queue length

In some environments, consumer applications may run slower than supplier applications. The `maxQueueLength` parameter prevents out-of-memory conditions by limiting the number of outstanding messages that will be held for each consumer that cannot keep up with the rate of messages from the supplier.

If a supplier generates 10 messages per second and a consumer can only process one message per second, the queue will quickly fill up. Messages in the queue have a fixed maximum length and if an attempt is made to add a

message to a queue that is full, the channel will remove the oldest message in the queue to make room for the new message.

Each consumer has a separate queue, so a slow consumer may miss messages while another, faster consumer may not lose any. The Consumer message queue length can be configured on a per Event Channel basis. This means that all Consumers for a given Channel will have separate but equal size queues.

If `maxQueueLength` is not specified or if an invalid number is specified, a default queue length of 100 is used.

**Code example 91** Sample use of setting the Queue Length

```
myChannel = myfactory-> create_by_name("MyChannel",
                                     150) // Queue Length
```

## Compiling and linking programs

C++ applications that use the event service need to include the following generated files:

```
#include "CosEventComm_s.hh" #include "CosEventChannelAdmin_c.hh"
```

## Interface reference

The remainder of this chapter provides reference information on all of the Event Service interfaces.

### EventChannel

The `EventChannel` provides the administrative operations for adding suppliers and consumers to the channel and for destroying the channel.

```
ConsumerAdmin for_consumers();
```

This method returns a `ConsumerAdmin` object that can be used to add consumers to this `EventChannel`.

```
SupplierAdmin for_suppliers();
```

This method returns a `SupplierAdmin` object that can be used to add suppliers to this `EventChannel`.

```
void destroy();
```

This method destroys this `EventChannel`.

### ConsumerAdmin

This interface is used by consumer applications to obtain a reference to a proxy supplier object. This is the second step in connecting a consumer application to an `EventChannel`.

**Code example 92** ConsumerAdmin interface

```
module CosEventChannelAdmin {
    interface ConsumerAdmin {
        ProxyPushSupplier obtain_push_supplier();
        ProxyPullSupplier obtain_pull_supplier();
    };
};
```

The `obtain_push_supplier` method is invoked if the calling consumer application is implemented using the push model. If the application is implemented using the pull model, the `obtain_pull_supplier` method should be invoked.

The returned reference is used to invoke either the `connect_push_consumer` or the `connect_pull_consumer` method.

## SupplierAdmin

This interface is used by supplier applications to obtain a reference to the proxy consumer object. This is the second step in connecting a supplier application to an `EventChannel`.

### Code example 93 SupplierAdmin interface

```
module CosEventChannelAdmin {
    interface SupplierAdmin {
        ProxyPushConsumer obtain_push_consumer();
        ProxyPullConsumer obtain_pull_consumer();
    };
};
```

The `obtain_push_consumer` method is invoked if the supplier application is implemented using the push model. If the application is implemented using the pull model, the `obtain_pull_consumer` method should be invoked.

The returned reference is used to invoke either the `connect_push_supplier` or the `connect_pull_supplier` method.

## ProxyPullConsumer

This interface is used by a pull supplier application and provides the `connect_pull_supplier` method for connecting the supplier's `PullSupplier`-derived object to the `EventChannel`. An `AlreadyConnected` exception is raised if `ProxyConsumer` is already connected to a `PullSupplier`.

### Code example 94 ProxyPullConsumer interface

```
module CosEventChannelAdmin {
    exception AlreadyConnected();
    interface ProxyPullConsumer : CosEventComm::PullConsumer {
        void connect_pull_supplier(
            in CosEventComm::PullSupplier pull_supplier)
            raises (AlreadyConnected);
    };
};
```

## ProxyPushConsumer

This interface is used by a push supplier application and provides the `connect_push_supplier` method, used for connecting the supplier's `PushSupplier`-derived object to the `EventChannel`. An `AlreadyConnected` exception is raised if `ProxyConsumer` is already connected to a `PullSupplier`.

### Code example 95 ProxyPushConsumer interface

```
module CosEventChannelAdmin {
    exception AlreadyConnected();
    interface ProxyPushConsumer : CosEventComm::PushConsumer {
        void connect_push_supplier(
            in CosEventComm::PushSupplier push_supplier)
            raises (AlreadyConnected);
    };
};
```

## ProxyPullSupplier

This interface is used by a pull consumer application and provides the `connect_pull_consumer` method, used for connecting the consumer's `PullConsumer`-derived object to the `EventChannel`. An `AlreadyConnected` exception is raised if `ProxyConsumer` is already connected to a `PullConsumer`.

### Code example 96 ProxyPullSupplier interface

```
module CosEventChannelAdmin {
    exception AlreadyConnected();
    interface ProxyPullSupplier : CosEventComm::PullSupplier {
        void connect_pull_consumer(
            in CosEventComm::PullConsumer pull_consumer)
            raises (AlreadyConnected);
    };
};
```

## ProxyPushSupplier

This interface is used by a push consumer application and provides the `connect_push_consumer` method, used for connecting the consumer's `PushConsumer`-derived object to the `EventChannel`. An `AlreadyConnected` exception is raised if `ProxyConsumer` is already connected to a `PullSupplier`.

### Code example 97 ProxyPushSupplier interface

```
module CosEventChannelAdmin {
    exception AlreadyConnected();
    interface ProxyPushSupplier : CosEventComm::PushSupplier {
        void connect_push_consumer(
            in CosEventComm::PushConsumer push_consumer)
            raises (AlreadyConnected);
    };
};
```

## PullConsumer

This interface is used to derive consumer objects that use the pull model of communication. The `pull` method is called by a consumer whenever it wants data from the supplier. A `Disconnected` exception will be raised if the supplier has disconnected.

The `disconnect_pull_consumer` method is used to deactivate this consumer if the channel is destroyed.

### Code example 98 PullConsumer interface

```
module CosEventComm {
    exception Disconnected {};
    interface PullConsumer {
        void disconnect_pull_consumer();
    };
};
```

The only method that must be implemented in the derived classes of `PullConsumer` is `disconnect_pull_consumer`, which is used to disconnect the `PullConsumer` from the `EventChannel`. For instance, in the `PullModel` example, the `PullSupplier` uses it to disconnect the pull consumer.

## PushConsumer

This interface is used to derive consumer objects that use the push model of communication. The `push` method is used by a supplier whenever it has

data for the consumer. It raises a `Disconnected` exception if the consumer has already been disconnected.

#### Code example 99 PushConsumer interface

```
module CosEventComm {
    exception Disconnected();
    interface PushConsumer {
        void push(in any data) raises(Disconnected);
        void disconnect_push_consumer();
    };
};
```

The `PushConsumer` implements the `push(in any data)` method. This method is called by the `PushSupplier` continuously to receive data until the `PushSupplier` is explicitly disconnected from the `PushConsumer` by a call to `disconnect_push_supplier` on the `ProxyPushSupplier` object.

## PullSupplier

This interface is used to derive supplier objects that use the pull model of communication.

#### Code example 100 PullSupplier interface

```
module CosEventComm {
exception Disconnected{}; interface PullSupplier {
any pull() raises(Disconnected); any try_pull() raises(Disconnected); void
disconnect_pull_supplier();
};
};
```

The `PullConsumer` pulls data from a `PullSupplier`. Once connected to a `ProxyPullSupplier`, `PullConsumer` can `pull()` or `try_pull()` on the `ProxyPullSupplier` object.

`try_pull()` is for asynchronous pull (returns immediately, even if the data is not yet available) and `pull()` is for synchronous pull (returns when the data is available).

`PullConsumer` calls `disconnect_pull_supplier()` on `ProxyPullServer` when the consumer wants to disconnect from the `ProxyPullSupplier`. The `pull()` and `try_pull()` methods return `CORBA::Any` objects. In the example, the returned `Any` object contains a numbered string that contains the value "Hello".

## PullSupplier methods

```
any pull();
```

This method blocks until there is data available from the supplier. The data is returned an `Any` type. If the consumer has disconnected, this method raises a `Disconnected` exception.

```
any try_pull(out boolean has_event);
```

This non-blocking method attempts to retrieve data from the supplier. When this method returns, `has_event` is set to `CORBA::TRUE` and the data is returned as an `Any` type if there was data available. If `has_event` is set to `CORBA::FALSE`, then no data was available and the return value will be `NULL`.

```
void disconnect_pull_supplier();
```

This method deactivates this pull server if the channel is destroyed.

## PushSupplier

This interface is used to derive supplier objects that use the push model of communication. The `disconnect_push_supplier` method is used by the `EventChannel` to disconnect supplier when it is destroyed.

### Code example 101 PushSupplier interface

```
module CosEventComm {  
    exception Disconnected();  
    interface PushSupplier {  
        void disconnect_push_supplier();  
    };  
};
```

`PushSupplier` should be implemented so that it constantly “pushes” data to the consumer. In the `PushModel` example, once a `PushModel` object (a `PushSupplier`-derived object) is created, it starts a new `Thread` that keeps calling `push(CORBA.Any)` on the `ProxyPushConsumer` at intervals. The pushed data is an `Any` with a message string (numbered Hello string) inserted.

The only method that must be implemented in the derived classes of `PushSupplier` is `disconnect_pull_consumer`, which is used to disconnect the `PullConsumer` from the `EventChannel`. For instance, in the `PushView` example, the `PushConsumer` uses it to disconnect the `ProxyPushSupplier`.



# Part 6

## Advanced Concepts

### **In this part**

This part contains the following chapters:

<a href="#">Real-Time CORBA Extensions</a>	<a href="#">page 199</a>
<a href="#">Listening and Dispatch Configuration</a>	<a href="#">page 223</a>
<a href="#">Connection Management</a>	<a href="#">page 235</a>
<a href="#">Bidirectional Communication</a>	<a href="#">page 239</a>
<a href="#">VisiBroker Pluggable Transport Interface</a>	<a href="#">page 243</a>
<a href="#">Using Portable Interceptors</a>	<a href="#">page 255</a>
<a href="#">Using VisiBroker Interceptors</a>	<a href="#">page 281</a>
<a href="#">Using Object Wrappers</a>	<a href="#">page 295</a>
<a href="#">Using Valuetypes</a>	<a href="#">page 313</a>
<a href="#">VisiBroker Logging</a>	<a href="#">page 319</a>















# Real-Time CORBA Extensions

*VisiBroker-RT for C++ supports a number of Real-Time CORBA extensions, as defined in the Real-Time CORBA 1.0 Specification. This chapter describes these extensions and how to apply them in application code.*

## Overview

VisiBroker-RT for C++ provides the following Real-Time CORBA extensions:

- **Real-Time CORBA Priority**

A platform-independent priority scheme, that is used to control the priority of threads related to the VisiBrokerRT60 application. Specifying priorities in terms of the Real-Time CORBA priority scheme, instead of the priority scheme of the particular RTOS, allows applications to be developed that schedule real-time activities consistently across machines running different RTOSs and even non-Real-Time Operating Systems. It also aids the porting and/or extension of applications to different Operating Systems at a later date.
- **Priority Mappings**

The means by which the Real-Time CORBA Priority scheme is 'mapped' onto the priority scheme of the underlying RTOS. The user may install a Priority Mapping, to control the way the priorities are mapped, or use the default mapping that is provided by the ORB.
- **Threadpools**

Real-Time CORBA entities that allow an application to control the threads used by the ORB to execute CORBA invocations.
- **Real-Time Object Adapters**

Enhanced Portable Object Adapters (POAs), that work with Threadpools and have a number of configurable Real-Time CORBA properties.
- **Real-Time CORBA Current interface**

An extension of the `CORBA::Current` interface, that allows Real-Time CORBA priority values to be assigned to application threads.
- **Real-Time CORBA Priority Models**

Two alternate models for deciding the priority at which CORBA invocations are executed.
- **Real-Time CORBA Mutex API**

An IDL-defined mutex interface, that gives applications access to the same mutex implementation as that used internally by the ORB. This guarantees consistent priority inheritance behavior, as well as improving application portability.
- **Real-Time ORB**

Used to manage the creation and destruction of other Real-Time CORBA entities, such as Threadpools and Mutexes.
- **Control of Internal ORB Thread Priorities**

Mechanisms to allow range limitation and explicit control of the priorities of all additional threads used internally within the ORB.

These features are explained in the sections that follow.

## Using the Real-Time CORBA Extensions

Applications that want to make use of the Real-Time CORBA extensions must include the C++ header file `rtcorba.h`, that is provided in the VisiBroker `include` directory.

Many of the Real-Time CORBA features have interfaces that are defined in IDL. The IDL for these features is specified in a new `RTCORBA` module. This IDL is available for inspection in the file `RTCORBA.idl`, which can be found in the `idl` directory of the VisiBroker installation.

However, there is no need to compile the IDL in `RTCORBA.idl` to make use of the Real-Time CORBA features. Applications need only to include the `rtcorba.h` header file that is provided with the other VisiBroker header files.

This is because all of the interfaces in the module are specified as 'locality constrained'. That is, their object references cannot be passed off-node or used to invoke operations on instances remotely. All manipulation of Real-Time CORBA interfaces must be performed locally, as is the case with other CORBA entities, such as `CORBA::ORB` and `PortableServer::POA`.

## Real-Time CORBA ORB

The Real-Time CORBA extensions include a Real-Time ORB interface, which is used to manage other Real-Time CORBA entities. The interface is named `RTCORBA::RTORB`, and has the following definition.

**Code example 102** Real-Time CORBA ORB IDL : interface `RTCORBA::RTORB`.

```
module RTCORBA {  
  
    // locality constrained interface  
    interface RTORB {  
  
        Mutex create_mutex();  
        void destroy_mutex( in Mutex the_mutex );  
        exception InvalidThreadpool {};  
  
        ThreadpoolId create_threadpool(  
            in unsigned long stacksize,  
            in unsigned long static_threads,  
            in unsigned long dynamic_threads,  
            in Priority default_priority,  
            in boolean allow_request_buffering,  
            in unsigned long max_buffered_requests,  
            in unsigned long max_request_buffer_size );  
  
        void destroy_threadpool( in ThreadpoolId threadpool )  
            raises (InvalidThreadpool);  
  
        void threadpool_idle_time(  
            in ThreadpoolId threadpool,  
            in unsigned long seconds )  
            raises (InvalidThreadpool);  
    };  
};
```

The operations shown in the IDL are described below, in the sections “[Threadpools](#)” and “[Real-Time CORBA Mutex API](#)”.

The Real-Time ORB does not need to be explicitly initialized - it is initialized implicitly as part of the regular `CORBA::ORB_init` call. Any Real-Time ORB initialization arguments are passed in to the call to `CORBA::ORB_init`, along with non-Real-Time arguments. If any Real-Time initialization argument is invalid, the `ORB_init` call will fail, and a system exception will be thrown.

To use the Real-Time ORB operations, the application must have a reference to the `RTCORBA::RTORB` instance. This reference can be obtained any time after the call to `ORB_init`, and is obtained by calling the `resolve_initial_references` operation on `CORBA::ORB`, with the object id "RTORB" as the parameter. Because `resolve_initial_references` returns the reference as a `CORBA::Object_ptr`, it must then be narrowed to a `RTCORBA::RTORB_ptr` before it can be used.

The code example below shows how to obtain the `RTCORBA::RTORB` reference. Similar code can be found in the Real-Time CORBA examples included with the VisiBroker release : `threadpool`, `priority_models` and `rtmutex`.

**Code example 103** Obtaining a reference for the Real-Time ORB via `resolve_initial_references`

```
#include "corba.h"
#include "rtcorba.h"

// First initialize the ORB
CORBA::ORB_ptr orb;
VISTRY
{
    orb = ORB_init(argc, argv);
}
VISCATCH(CORBA::Exception, e)

{
    cerr << "Exception initializing ORB" << endl << e << endl;
    // handle error here
}
VISEND_CATCH

// Then obtain the RTORB reference
CORBA::Object_var ref;
// Note use of _var, so ref will be automatically released
VISTRY
{
    ref = orb->resolve_initial_references("RTORB");
}
VISCATCH
{
    cerr << "Exception obtaining RTORB reference" << endl << e <<
endl;
    // handle error here
}
VISEND_CATCH

// Finally, narrow the RTORB reference
RTCORBA::RTORB_ptr rtorb;
VISTRY
{
    rtorb = RTCORBA::RTORB::_narrow(ref);
    // ref is no longer needed. Will be automatically released as
is a _var
}
VISCATCH(CORBA::Exception, e)
{
    cerr << "Error narrowing RTORB reference" << endl << e <<
endl;
    // Handle error here
}
VISEND_CATCH
```

# Real-Time Object Adapters

In Real-Time CORBA, all Object Adapters are Real-Time Object Adapters. This means that all Object Adapters are aware of priorities and handle CORBA invocations according to rules defined by Real-Time CORBA. It is necessary for all Object Adapters on a node to be Real-Time: if some Object Adapters in the CORBA application were non-Real-Time, their operation would interfere with the behavior of the Real-Time Object Adapters (because threads associated with all Object Adapters must be scheduled together by the OS.)

As all Object Adapters are Real-Time, the normal Portable Object Adapter (POA) interface is used to manage them.

Real-Time Object Adapters are created in the normal way, through a call to `create_POA`. Configuration of the extra, Real-Time properties is achieved through the passing of new Real-Time policies in the policy list parameter. An example of POA creation specifying one such new policy (and its associated value) is shown below:.

## Code example 104 Configuration of a Real-Time Policy at time of POA creation

```
// Create Real-Time CORBA Priority Model Policy
// (Already obtained RTORB reference)
RTCORBA::PriorityModelPolicy_ptr priority_model_policy =
    rtorb->create_priority_model_policy(
        RTCORBA::SERVER_DECLARED, 25 );

// Create Policy List containing this RT CORBA Policy
// (Include any required non-Real-Time policies in the same list)
CORBA::PolicyList policies;
policies.length(1);
policies[0] = priority_model_policy;

// Create POA, using the Policy List
// (Associate POA with the Root POA's POA manager, if none other)
// (Already obtained Root POA reference)
PortableServer::POAManager_var poa_manager =
    rootPOA->the_POAManager();
VISTRY
{
    poa = rootPOA->create_POA("myPOA", poa_manager, policies);
}

VISCATCH(CORBA::Exception, e)
{
    // handle exceptions here
}
VISEND_CATCH
```

The Real-Time policies that can be configured at the time of POA creation are concerned with the Priority Model that the POA supports and which Threadpool it will be associated with.

The configuration of these properties is described in the sections [“Threadpools”](#) and [“Real-Time CORBA Mutex API”](#).

If any of these Real-Time properties is not configured by the application at the time of POA creation, the ORB will initialize that property with a default value. The default Priority Model behavior is for the POA to support the Server Declared Priority Model, and the default Threadpool behavior is for the POA to be associated with the General Threadpool. These defaults are explained in the two sections on Priority Models and Threadpools.

# Real-Time CORBA Priority

Real-Time CORBA defines a universal, platform independent priority scheme called *Real-Time CORBA Priority*. It allows Real-Time CORBA applications to make prioritized CORBA invocations in a consistent fashion between nodes running different Operating Systems. Even if all nodes in the existing system are running the same Operating System, its use aids the configuration of priorities in the system, and will improve application portability and simplify future extension to a mixed OS environment.

For consistency and portability, Real-Time CORBA applications are obliged to use Real-Time CORBA Priority to express the priorities in the (CORBA part of the) application, even if all nodes in a system use the same OS, and hence the same priority scheme.

The `RTCORBA::Priority` type is used to represent Real-Time CORBA Priority:

```
module RTCORBA {
    typedef short Priority;
    const Priority minPriority = 0;
    const Priority maxPriority = 32767;
};
```

A signed short is used in order to accommodate the Java language mapping. However, only values in the range 0 (minPriority) to 32767 (maxPriority) are valid.

## NOTE

Numerically higher `RTCORBA::Priority` values are defined to be of higher priority. This is the reverse of the priority scheme used by VxWorks, where 0 is the highest priority.

In practice, an application does not need to use the entire range of valid `RTCORBA::Priority` values (0 to 32767.) A smaller range, that suits the needs of the application, can be defined as the only admissible range. This is achieved through control of the *Priority Mapping*. Priority Mappings are described in the next section.

By default, VisiBroker-RT for C++ installs a Priority Mapping that only allows `RTCORBA::Priority` values in the range 0 to 31. (The POSIX threading range of priorities.) See the next section for details.

# Priority Mappings

A given Real-Time Operating System has a particular priority 'scheme': the range and direction of priority values that it uses. The VxWorks priority scheme is priorities in the range 0 to 255, ranging from 255 as the lowest priority to 0 as the highest priority. In Real-Time CORBA, this is referred to as the *Native* priority scheme of VxWorks, and the VxWorks priority values are referred to as *Native Priority* values.

As the Real-Time CORBA application will describe its priorities in terms of `RTCORBA::Priority` values, and the OS works in terms of *Native Priority* values, a mapping must be defined between these two priority schemes. The mapping is used by the ORB, to obtain the Native Priority corresponding to a given `RTCORBA::Priority` value, and vice versa, as is required. This is done, for example, when an application specifies that it wants a Threadpool to have threads that are created with a particular `RTCORBA::Priority`, and the ORB needs to know what Native Priority to tell the OS to use when it actually creates the threads.

The Priority Mapping may also be used directly by the application. But this should only occur in special circumstances. This is discussed further in section [“Using Native Priorities in VisiBroker Application Code”](#).

The ORB comes with a default Priority Mapping, which is sufficient for experimenting with the Real-Time CORBA features and may be sufficient for many Real-Time applications (since it is based on the POSIX priority scheme.) Therefore, when first becoming familiar with the Real-Time features of VisiBroker-RT for C++, it may be appropriate to skip the rest of this section, and learn about the rest of the Real-Time CORBA features (beginning in the section [“Threadpools”](#)), before returning to this section to understand the details of Priority Mappings and the reasons for installing one that is different from the default.

## Priority Mapping Types

To support Priority Mappings, a `RTCORBA::NativePriority` type and `RTCORBA::PriorityMapping` type are defined :

```
module RTCORBA {
    typedef short NativePriority;
    native PriorityMapping
};
```

`RTCORBA::NativePriority` values must be integers in the range -32768 to +32767. However, for a particular RTOS, the valid range will be a sub-range of this range. For VxWorks, the valid range is 0 to 255.

The `RTCORBA::PriorityMapping` type is defined as an IDL *native* interface. This means that the interface is defined directly in each implementation language, rather than being defined in IDL and mapped automatically to each language according to the rules of the particular CORBA language mapping. This is done for reasons of efficiency.

The C++ mapping of the `RTCORBA::PriorityMapping` interface is:

```
//C++
class PriorityMapping {
public:
    virtual CORBA::Boolean to_native(
        RTCORBA::Priority corba_priority,
        RTCORBA::NativePriority &native_priority );

    virtual CORBA::Boolean to_CORBA(
        RTCORBA::NativePriority native_priority,
        RTCORBA::Priority &corba_priority );

    virtual RTCORBA::Priority max_priority();

    PriorityMapping();
    virtual ~PriorityMapping() {}
    static RTCORBA::PriorityMapping * instance();
};
```

The methods that define the behavior of a particular Priority Mapping are `to_native`, `to_CORBA` and `max_priority`. Their purpose is as follows:

### `to_native`

This method takes a `RTCORBA::Priority` value from the `corba_priority` parameter and either maps it to a `RTCORBA::NativePriority` value or fails to map it. If the value is mapped, the resulting Native Priority value is stored in the location referenced by the parameter `native_priority` (which is a C++ reference parameter) and a true value is returned to indicate that the mapping was successful. If the value is not mapped, the contents of the `native_priority` parameter are not altered, and a false value is returned to indicate that the mapping operation failed.

`to_CORBA`

The converse of `to_native`, this method takes a `RTCORBA::NativePriority` value from the `native_priority` parameter, and either maps it to a `RTCORBA::Priority` value or fails to map it. If the value is mapped, the resulting `RTCORBA::Priority` value is stored in the location referenced by the `corba_priority` parameter (which is a C++ reference parameter) and a true value is returned to indicate that the mapping was successful. If the value is not mapped, the contents of the `corba_priority` parameter are not altered, and a false value is returned to indicate that the mapping operation failed.

`max_priority`

This method just returns the highest `RTCORBA::Priority` value that is valid in this mapping. The ORB needs to be explicitly told the highest value as there is no efficient way for it to determine it by examining the behavior of the `to_native` and `to_CORBA` methods given different input values.

The implementation of these methods must conform to certain rules, that are described below.

## Rules for Priority Mappings

Any Priority Mapping that is installed (including the default Priority Mapping) must conform to the following rules:

- The `to_native` and `to_CORBA` methods should be able to handle all values of their input parameter, in the range -32768 to +32767.
- `to_native` must definitely fail to map values outside the range 0 to 32767, and may fail to map values within that range as well. (For example the default Priority Mapping fails to map all values outside the range 0 to 31.)
- `to_CORBA` must definitely fail to map values outside the range of the Native Priority scheme and may fail to map values within that range as well. (The default Priority Mapping chooses to only map from VxWorks Native Priorities in the range 100 to 131.)
- Lower `RTCORBA::Priority` values should always map to/from lower importance Native Priority values, and higher to higher. Note that in the case of a VxWorks based operating system, this means mapping numerically lower `RTCORBA::Priority` values to/from numerically higher Native Priorities. This follows the convention used by the majority of Real Time Operations Systems. VxWorks is at odds with this convention, in making 0 the highest importance priority. The reason for following the convention is to maintain consistency with Real-Time CORBA applications developed on other RTOSs. Otherwise future porting and interworking with other Real-Time applications will be greatly complicated
- `RTCORBA::Priority 0` should always be mapped, and always be mapped to the lowest importance Native Priority value in the range of Native Priority values that is mapped to/ from. (The default Priority Mapping maps `RTCORBA::Priority 0` to VxWorks Priority 131, which is the lowest importance priority in the default mapping.)
- `max_priority` must return the highest `RTCORBA::Priority` value that is mapped by the mapping. (That is, the highest value for which a Native Priority value is returned.)

The following are not mandated, but will often be the case, unless there is special reason to do otherwise:

- `to_native` and `to_CORBA` will usually return the same value (or fail to map) every time they are called with the same input value.
- `to_native` and `to_CORBA` will usually be reverse mappings of one another.
- The ranges of `RTCORBA::Priority` and Native Priority values that are mapped will usually each be a single contiguous range of priority values.

## Default Priority Mapping

VisiBroker-RT for C++ provides a default Priority Mapping. This is the Priority Mapping that will be used unless a different one is written by the application developer and installed using the process described below, in the section [“Replacing the Default Priority Mapping”](#).

### Note

Only one Priority Mapping may be installed at any one time on a given VisiBrokerRT60 system. The act of installing one Priority Mapping automatically un-installs the previously installed Priority Mapping (which will usually be the default Priority Mapping.)

The default Priority Mapping has the following characteristics:

- Valid `RTCORBA::Priority` range is 0 to 31 only. This follows the POSIX threading model. All priorities outside of this range are invalid, which means an exception will be thrown if an attempt is made to use them.
- The valid `RTCORBA::Priority` values are mapped one-to-one onto a 32 priority sub-range of the VxWorks Native Priority range. Specifically, they are mapped onto the Native Priority range 100 to 131.
- The valid `RTCORBA::Priority` values are mapped onto the Native Priority range in such a way that `RTCORBA::Priority` value 0 corresponds to the lowest-importance Native Priority in the sub-range used, and `RTCORBA::Priority` 31 corresponds to the highest-importance Native Priority in the sub-range used. Specifically:

```
RTCORBA::Priority 0 maps to VxWorks Native Priority 131; and
RTCORBA::Priority 31 maps to VxWorks Native Priority 100.
```

The default Priority Mapping is defined within the ORB, and hence the source code for it is not included in the VisiBrokerRT60 release. The source code for the mapping is shown here, however, to show exactly how this mapping behaves,

### Code example 105 The default Priority Mapping implementation

```
// VisiBroker for C++ for Tornado Default Priority Mapping
CORBA::Boolean
VISDefaultPriorityMapping::to_native(
    RTCORBA::Priority corba_priority,
    RTCORBA::NativePriority &native_priority )
{
    if ((corba_priority < 0) || (corba_priority > 31))
    {
        return FALSE;
    }

    else
    {
        native_priority = 131 - corba_priority; // 0 -> 131, 31 ->
        100 return TRUE;
    }
}

// Default 'to_corba' mapping CORBA::Boolean
VISDefaultPriorityMapping::to_CORBA(
```

```

RTCORBA::NativePriority native_priority RTCORBA::Priority
&corba_priority )
{
    if ((native_priority < 100) || (native_priority > 131))
    {
        return FALSE;
    }
    else
    {
        corba_priority = 131 - native_priority; // 131 -> 0, 100 -> 31
        return TRUE;
    }
}

// Default max method : returns the max RTCORBA::Priority
// supported by the default priority mapping
RTCORBA::Priority VISDefaultPriorityMapping::max_priority()
{
    return 31;
}

```

## Replacing the Default Priority Mapping

### Note

Only one Priority Mapping may be installed at any one time on a particular machine. The act of installing one Priority Mapping automatically uninstalls the previously installed Priority Mapping (which will usually be the default Priority Mapping.)

The application may wish to replace the default Priority Mapping on some or all nodes in the system. Reasons for doing this include:

- To 'shift' the range of Native Priority values that are mapped to/from higher or lower in the overall Native Priority scheme. For example to take the default Priority Mapping's range of Native Priority 100 to 131, and replace it with the range 50 to 81 (higher importance) or 200 to 231 (lower importance.)
- To have more or less `RTCORBA::Priority` values in the range of valid (i.e. mapped) values. For example, to only map `RTCORBA::Priority` values in the range 0 to 8 or to map values in the range 0 to 128.
- To have more or less Native Priority values in the range of valid (i.e. mapped) values. For example, to map to/from Native Priority values in the range 128 to 256.

Note that the relationship between the ranges of `RTCORBA::Priority` and Native Priority values that are valid in the mapping will determine whether the mapping is a one-to-one mapping or not. The mapping does not have to be a one-to-one mapping, but this may be convenient. The default Priority Mapping is a one-to-one mapping.

### NOTE

Installed Priority Mappings should follow the convention, used in the default Priority Mapping, of making the `RTCORBA::Priority 0` have the lowest importance. On VxWorks, this means ensuring that `RTCORBA::Priority 0` maps to the numerically largest VxWorks Native Priority value (of the sub-range that is being mapped to.) The reason for doing this is to maintain consistency with Real-Time CORBA applications developed on other RTOSs. Otherwise future porting and interworking with other Real-Time applications will be greatly complicated.

A new Priority Mapping is installed by defining a new class, which must inherit from the class `TCORBA::PriorityMapping`, and creating one static

instance of it in the application. When the static instance is initialized (during the execution of static constructors) the base `RTCORBA::PriorityMapping` class' constructor will register the new mapping with the ORB.

For an example of a writing and installing a new Priority Mapping, look at the files `mapping.h` and `mapping.C` in the `threadpool` example included in the `<install_location>/VisiBrokerRT60/examples` directory under the VisiBroker installation. Note the single instance of the new class that is created in global scope in `mapping.C`. When the resulting `mapping.o` is loaded onto a VxWorks target, and static constructor initialization takes place, it is the initialization of this instance that installs the mapping.

(To see the effect of installing the mapping, compare the behavior of loading and running the `corba_init` and `corba_init_mapping` executables. `corba_init_mapping` has `mapping.o` linked in, `corba_init` does not.)

## Using Native Priorities in VisiBroker Application Code

Although applications are obliged to use Real-Time CORBA Priority to reason about the priority of different parts of their CORBA application (and the priority of CORBA invocations between parts of the application), there are cases in which the application will need to deal in terms of Native Priority. For example, to configure some sub-system outside of the CORBA application, that only knows about the Native Priority scheme, or to use some OS call directly, that takes a (Native) priority value as a parameter.

Hence, it may be necessary to translate between Real-Time CORBA and Native Priority in the application. To allow this, VisiBroker-RT for C++ offers a convenience method, that returns a pointer to the currently installed Priority Mapping. The method is the static `instance` method on the class `RTCORBA::PriorityMapping`.

Using this, the application can call the Priority Mapping's methods directly, but is always guaranteed to be working with the installed mapping. This allows the code to continue to work if the mapping is changed.

**Code example 106** An example of using the installed Priority Mapping from application code

```
RTCORBA::Priority corba_priority;

// Priority Mapping methods return boolean flag, rather than
// throwing exceptions
if
(!RTCORBA::PriorityMapping::instance()->to_CORBA(100,
corba_priority
))
{
// Handle failure to map native priority to RT CORBA priority
}

// Use corba_priority value here...
```

## Threadpools

VisiBroker-RT for C++ uses Threadpools to manage the threads of execution on the server-side of the ORB. Threadpools offer the following features:

- Pre-allocation of threads.

This helps guarantee Real-Time system behavior, by allowing the application programmer to ensure that there are enough thread resources to satisfy a certain number of concurrent invocations, and also helps reduce latency and increase predictability, by avoiding the destruction and recreation of threads between invocations.

- **Partitioning of threads.**  
Having multiple Threadpools, associated with different Object Adapters allows one part of the system to be isolated from the thread usage of another, possibly lower priority, part of the application system. This can again be used to help achieve Real-Time behavior of the system as a whole.
- **Bounding of thread usage.**  
A Threadpool can be used to set a maximum limit on the number of threads that a POA or set of POAs may use. In systems where the total number of threads that may be used is constrained, this can be used in conjunction with Threadpool partitioning to avoid thread starvation in a critical part of the system.

## Threadpool API

Threadpools are managed using the following operations of the `RTCORBA::RTORB` interface:

```
module RTCORBA {  
  
    typedef unsigned long ThreadpoolId;  
  
    // locality constrained object  
    interface RTORB {  
  
        exception InvalidThreadpool {};  
  
        ThreadpoolId create_threadpool(  
            in unsigned long stacksize,  
            in unsigned long static_threads,  
            in unsigned long dynamic_threads,  
            in Priority default_priority,  
            in boolean allow_request_buffering,  
            in unsigned long max_buffered_requests,  
            in unsigned long max_request_buffer_size );  
  
        void destroy_threadpool( in ThreadpoolId threadpool )  
            raises (InvalidThreadpool);  
  
        void threadpool_idle_time(  
            in ThreadpoolId threadpool,  
            in unsigned long seconds )  
            raises (InvalidThreadpool);  
    };  
};
```

These operations are described in the sections that follow. Examples of Threadpool creation and their association with POAs can be found in the `threadpool` example included with the VisiBroker installation.

## Threadpool Creation and Configuration

A Threadpool is created by invoking the `create_threadpool` operation on the Real-Time ORB. The arguments to `create_threadpool` have the following significance:

- `stacksize`

The stack size, in bytes, that each thread created for the Threadpool should have.

- `static_threads`  
The number of threads that will be created and assigned to the pool at the time of Threadpool creation. These threads will not be destroyed until the Threadpool itself is destroyed. After they have been used to execute a CORBA invocation, they are returned to the Threadpool, and await another invocation to execute.
- `dynamic_threads`  
The number of threads that may be created dynamically, to execute CORBA invocations received when all the static threads are currently in use. The number may be zero, in which case no threads may be dynamically created after Threadpool creation. (In this case, the number of concurrently executing invocations is limited by the number of static threads.)
- `default_priority`  
The `RTCORBA::Priority` at which idle threads should remain while in the pool waiting for a CORBA invocation to execute. The priority at which the invocation will be executed depends on the Real-Time CORBA Priority Model in use. See the section “[Real-Time CORBA Priority Models](#)” for details. This parameter determines the priority of the threads when they are not handling invocations.
- `allow_request_buffering`, `max_buffered_requests` and `max_request_buffer_size`  
These arguments support the Request Buffering feature from the Real-Time CORBA specification, that allows for invocation requests to be queued once the static and dynamic thread limits of a Threadpool have been reached. This feature is not currently supported in VisiBroker-RT for C++, and the value of these arguments is ignored.

If `dynamic_threads` is greater than zero, so that threads may be created dynamically, the threads are not immediately destroyed after they have completed executing the CORBA invocation that they were created to handle. They are returned to the Threadpool, in the same way that static threads are. However, dynamic threads that remain idle in the Threadpool may eventually be destroyed during garbage collection that occurs from time to time.

The amount of time a dynamically created thread must remain idle in a Threadpool before it is destroyed may be set using the `threadpool_idle_time` operation of `RTCORBA::RTORB`. If the idle time is not set using this operation, it defaults to 300 seconds.

If successful, `create_threadpool` returns an identifier for the new Threadpool. The identifier is of type `RTCORBA::ThreadpoolId` (an unsigned long), and is subsequently used to refer to that Threadpool.

## Association of an Object Adapter with a Threadpool

Every POA created using VisiBroker-RT for C++ is associated with a Threadpool. Each Threadpool, on the other hand, may be associated with any number of POAs. By configuring multiple POAs to use the same or different Threadpools, the application designer can control the use of threads by different sets of CORBA Objects.

Which Threadpool a POA is associated with is determined by passing the `RTCORBA::ThreadpoolId` of the desired Threadpool into the `create_POA` operation as the value of a `RTCORBA::ThreadpoolPolicy` policy.

**Code example 107** Associating a POA with a Threadpool at time of POA initialization

```
// Obtain RTORB reference
CORBA::Object var objref =
orb->resolve_initial_references("RTORB");
RTCORBA::RTORB_var rtorb = RTCORBA::RTORB::_narrow(objref);

// Create a Threadpool
RTCORBA::ThreadpoolId tpool_id = rtorb->create_threadpool(
    30000, // stacksize
    5,    // num static threads
    0,    // num dynamic threads
    20,   // default RT CORBA priority
    0, 0, 0);

// Create Threadpool Policy object for use in POA initialization
RTCORBA::ThreadpoolPolicy_ptr tpool_policy =
    rtorb->create_threadpool_policy( tpool_id );

// Create Policy List for POA initialization
// (Include any required non-Real-Time policies in the same list)
CORBA::PolicyList policies;
    policies.length(1);
    policies[0] = tpool_policy;

// Create POA, using the Policy List
// (Associate POA with the Root POA's POA manager, if none other)
// (Already obtained Root POA reference)
PortableServer::POAManager_var poa_manager =
    rootPOA->the_POAManager();
VISTRY
{
    poa = rootPOA->create_POA("myPOA", poa_manager, policies);
}

VISCATCH(CORBA::Exception, e)
{
    // handle exceptions here
}
VISEND_CATCH
```

The `create_POA` fails if any part of the Real-Time CORBA configuration is invalid. For example, if the `ThreadpoolId` is not for a currently existing Threadpool, a `CORBA::BAD_PARAM` system exception will be thrown.

## The General Threadpool

If a Threadpool is not specified at POA creation time, as described in the previous section, then the new POA that is created is associated with a special Threadpool, called the *General Threadpool*.

The General Threadpool does not have to be created by a call to `RTCORBA::RTORB`'s `create_threadpool` operation. Instead, the General Threadpool is created automatically by the ORB the first time it is required. That is, it is created the first time `create_POA` is called without specifying a Threadpool. Hence, if all POAs are created specifying

application-created Threadpools, the General Threadpool will not be created. The General Threadpool will be created with the following configuration:

- `stacksize = 30000`
- `static_threads = 0`
- `dynamic_threads = 1000`

- `default_priority = 0`
- `max_thread_idle_time = 300`

If this configuration is not appropriate for the application, the General Threadpool should **not** be used, and the application should explicitly associate each POA with an appropriately configured Threadpool at POA creation time.

## Threadpool Destruction

A Threadpool may be destroyed by passing its ThreadpoolId as the argument to a call to the `destroy_threadpool` operation of `RTCORBA::RTORB`:

```
// RTORB reference and Threadpool id obtained previously

// Get RT ORB reference
CORBA::Object_var objref =
    orb->resolve_initial_references("RTORB");

RTCORBA::RTORB_var rtorb = RTCORBA::RTORB::_narrow(objref);

VISTRY
{
    rtorb->destroy_threadpool( pool_id );
}
VISCATCH(CORBA::Exception, e)
{
    // handle error here
}
VISEND_CATCH
```

All POAs that have been associated with a particular Threadpool (i.e. that had this Threadpool specified as the Threadpool to use, at the time of POA creation) must have been destroyed before the `destroy_threadpool` operation will succeed.

If POAs still exist that are associated with the Threadpool, the call fails and a system exception is thrown.

## Real-Time CORBA Current

Real-Time CORBA defines a Real-Time CORBA Current interface to provide access to the CORBA priority of a thread.

**Code example 108** The `RTCORBA::Current` interface

```
module RTCORBA {

    interface Current : CORBA::Current {
        attribute Priority base_priority;
    };
};
```

A Real-Time CORBA Priority may be associated with the current thread, by setting the `base_priority` attribute of the `RTCORBA::Current` object. This has two effects:

- The Native Priority of the current thread will immediately be set to the value mapped from the Real-Time CORBA Priority value given as the parameter to the set attribute operation. Thus setting this attribute has the effect of controlling the priority of CORBA application threads.
- The Real-Time CORBA Priority value is stored, for use with any CORBA invocations made from that thread. The value is only relevant when making invocations on CORBA Objects that were created from POAs that are configured to support the 'Client Priority Propagation' Priority Model.

(See the next section, “Real-Time CORBA Priority Models”.)

The priority value stays in effect (for both of the above purposes) until a new value is set. The current value can also be read, using the corresponding get attribute operation.

A `CORBA::BAD_PARAM` system exception will be thrown by the set attribute operation if an attempt is made to set a priority outside of the valid 0 to 32767 range. A `CORBA::DATA_CONVERSION` exception will be thrown if an attempt is made to set a priority that is in the 0 to 32767 range, but outside of the range supported by the currently installed Priority Mapping.

A `CORBA::INITIALIZE` system exception will be thrown if an attempt is made to get the priority value from a thread that has not yet had a Real-Time CORBA Priority value set on it. (The Native Priority of the current thread is not just mapped to a Real-Time CORBA Priority and returned.)

To use the `RTCORBA::Current` object, a reference to it must be obtained. This is achieved by calling the `CORBA::ORB` operation `resolve_initial_references` with the parameter “RTCurrent”, as in the following example.

**Code example 109** Obtaining the reference of `RTCORBA::Current`

```
// ORB previously initialized CORBA::ORB_ptr orb;

// Obtain the RTCORBA::Current reference
CORBA::Object_var ref;
// Note use of _ptr. The reference will be automatically released

VISTRY
{
    ref = orb->resolve_initial_references("RTCurrent");
}
VISCATCH
{
    // handle error here
}
VISEND_CATCH

// Narrow the RTCORBA::Current reference
RTCORBA::Current_ptr rcurrent;
VISTRY
{
    rcurrent = RTCORBA::Current::_narrow(ref);
    // ref is no longer needed. Will be automatically released
    // as is a _var
}
VISCATCH(CORBA::Exception, e)
{
    // handle error here
}
VISEND_CATCH
```

Note that the `RTCORBA::Current` reference only needs to be obtained once. The same variable may be used by different threads, and will behave as if it is private to each of them (setting and getting their thread-specific priority value.) This behavior is inherited from the base `CORBA::Current` object.

## Real-Time CORBA Priority Models

Real-Time CORBA supports two models for the coordination of priorities across a system. These two models provide two alternate answers to the question: where does the priority at which the CORBA invocation is executed come from? They are:

- Client Propagated Priority Model

In this model, the Real-Time CORBA Priority associated with a client CORBA application thread, using `RTCORBA::Current`, is also used as the priority on the server-side of the invocation. The thread that executes the invocation (which is taken from a Threadpool) runs at a Native Priority that is mapped from the Real-Time CORBA priority set on the client side prior to making the invocation.

- Server Declared Priority Model

In this model the Real-Time CORBA Priority associated with a client CORBA application thread only affects the priority on the client-side of the invocation. The priority that the invocation is handled at on the server-side is determined by the configuration of the CORBA Object and the POA that created it.

Which Priority Model is used is a server-side issue, configured at the POA level. All CORBA Objects created from the same POA will have their invocations processed according to the Priority Model the POA is configured with.

The Priority Model is selected at POA initialization time, by including a `RTCORBA::PriorityModelPolicy` instance in the Policy List passed as a parameter to `create_POA`. The Policy is configured with one or other of the two values:

- `RTCORBA::CLIENT_PROPAGATED`  
To select the Client Priority Propagation Model.
- `RTCORBA::SERVER_DECLARED`  
To select the Server Declared Model.

In either case a `RTCORBA::Priority` value is also specified as part of the Policy. The two models use this priority value differently:

- In the Client Priority Propagation Model, the value is the priority at which to execute invocations from clients that did not set a priority prior to making the invocation. This will include clients from non-Real-Time ORBs (including non-Real-Time ORBs from other vendors), and also invocations from threads that have not yet set a priority value using `RTCORBA::Current`.
- In the Server Declared Model, the value is the priority at which invocations will be executed, unless a different priority is set at the Object level. See the section below for details on the setting of the priority at the Object level.

The Server Declared Model is the default model. If a POA is initialized without specifying which model to use, it will be configured to use the Server Declared Model. However, in this case there is a subtle difference in behavior; because a priority has not been specified, the invocations run at the default priority of the Threadpool that the POA is associated with. (The default priority is a configurable property of Threadpools. It is the priority that threads remain at when idle in the pool. See the section on Threadpools for details.)

The following code demonstrates the setting of the Priority Model Policy at the time of POA creation. In this case, the Client Priority Propagation Model is selected, with a default priority of 7 (for invocations from non-Real-Time Clients).

**Code example 110** Configuration of Real-Time Priority Model Policy at POA creation

```
// Create Real-Time CORBA Priority Model Policy
// (Already obtained RTORB reference)
RTCORBA::PriorityModelPolicy_ptr priority_model_policy =
    rtorb->create_priority_model_policy(
    RTCORBA::CLIENT_PROPAGATED, 7 );
```

```

// Create Policy List containing this RT CORBA Policy
// (Include any required non-Real-Time policies in the same list)
CORBA::PolicyList policies;
policies.length(1);
policies[0] = priority_model_policy;

// Create POA, using the Policy List
// (Associate POA with the Root POA's POA manager, if none other)
// (Already obtained Root POA reference)
PortableServer::POAManager_var poa_manager =
    rootPOA->the_POAManager();
VISTRY
{
    poa = rootPOA->create_POA("myPOA", poa_manager, policies);
}

VISCATCH(CORBA::Exception, e)
{
    // handle exceptions here
}
VISEND_CATCH

```

See the `priority_models` example included with the VisiBroker installation for further examples of configuring the two different Priority Models.

## Client Model Backwards Compatability with VisiBroker 3.2.2

VisiBroker-RT for C++ 3.2.2 was implemented before the Real-Time CORBA Specification was finalized. As a consequence, it uses a non-standard value for the ServiceId of the Service Context used to propagate the client thread's Real-Time CORBA Priority from the client to the server.

By default, VisiBroker-RT for C++ 6.0 sends only the standard ServiceId value. Setting the property `vbroker.orb.clientModel.backCompat` to true causes two Service Contexts to be sent:

- one with the standard Service Id and
- one with the old 322 Service Id.

This allows a VisiBroker 6.0 client to propagate Real-Time CORBA Priority values to a VisiBroker 3.2.2 server.

### NOTE

VisiBroker-RT for C++ 6.0 servers always accept Real-Time CORBA Priority values from VisiBroker 3.2.2 clients, whether this property is true or false.

## Setting Priority at the Object Level

When the Server Declared Model is selected a priority value is supplied to determine the priority at which invocations will be executed on the server-side of the ORB. This priority value is used when handling invocations on behalf of any CORBA Object created by that POA.

However, this scope of control of priority is too coarse for some applications. To remedy this, Real-Time CORBA allows the priority that invocations will be executed at in the Server Declared model to be overridden on a per-Object basis.

The priority to run invocations at may be overridden for a given object by using either the operation `activate_object_with_priority` or `activate_object_with_id_and_priority` to activate the object in question. These operations work in the same way as `activate_object` and

activate `object_with_id`, but take a Real-Time CORBA Priority value as an additional parameter.

These operations are specified as part of the VisiBroker Extended POA interface, `PortableServerExt::POA`, which is accessed by narrowing a POA object reference using the static C++ method

```
PortableServerExt::POA::_narrow.
```

For an example of setting the priority on a per-Object basis, see the file `model_server.C` in the `priority_models` example included with VisiBroker.

## Real-Time CORBA Mutex API

VisiBroker-RT for C++ implements the following Real-Time CORBA Mutex interface:

```
#include "timebase.idl"
module RTCORBA {

    // locality constrained interface
    interface Mutex {
        void lock();
        void unlock();
        boolean try_lock(in TimeBase::TimeT max_wait);
        // if max_wait = 0 then return immediately
    };

    interface RTORB {
        ...

        Mutex create_mutex();
        void destroy_mutex( in Mutex the_mutex );
        ...
    };
};
```

A new `RTCORBA::Mutex` object is obtained using the `create_mutex` operation of `RTCORBA::RTORB`. A `Mutex` object has two states: locked and unlocked. `Mutex` objects are created in the unlocked state. When the `Mutex` object is in the unlocked state the first thread to call the `lock()` operation will cause the `Mutex` object to change to the locked state.

Subsequent threads that call the `lock()` operation while the `Mutex` object is still in the locked state will block until the owner thread unlocks it by calling the `unlock()` operation.

The `try_lock()` operation works like the `lock()` operation except that if it does not get the lock within `max_wait` time it returns `FALSE`. If the `try_lock()` operation does get the lock within the `max_wait` time period it returns `TRUE`.

The mutex returned by `create_mutex` must have the same priority inheritance properties as those used by the ORB to protect resources. If a Real-Time CORBA implementation offers a choice of priority inheritance protocols, or offers a protocol that requires configuration, the selection or configuration will be controlled through an implementation specific interface.

## Control of Internal ORB Thread Priorities

VisiBroker-RT for C++ allows the application to control the priority of the threads that the ORB creates for internal use.

The internal ORB threads are:

- VISLogger threads

These are the threads that VisiBroker Logger Forwarders run on. One Logger Forwarder thread is created at ORB initialization time. The thread name is 'VISLogger'. Other instances will be created if more Loggers are created by the application. The additional Logger Forwarder threads have task names of the form 'VISLogger<n>', where <n> is an index number that starts from one and corresponds to the order in which the Loggers were created.

- DSUser thread

A single DSUser thread is created the first time the ORB attempts to communicate with the OS Agent. This will usually happen the first time either `activate_object` or a `_bind` method is called. This thread manages all communication between the ORB and the OS Agent. The task name is 'VISDSUser'.

- Listener threads

Listener Threads will be created as part of the initialization of a Server Engine. (This occurs during POA initialization, whenever a POA wishes to use a Server Engine that has not been yet been used.) These threads wait for incoming CORBA invocations to be received from network connections. Listener Threads for IIOP communication have task names of the form 'VISLis<n>', where <n> is an index number that starts from zero and indicates the order in which the listeners were created.

- Garbage Collection thread

A single instance of this is created the first time a Threadpool is created. This will occur either when the application explicitly creates a Threadpool, or the first time the application creates a POA without specifying a Threadpool (in which case the General Threadpool will be created so that it can be used.) Garbage Collection Threads have task names of the form 'VISGC<n>', where <n> corresponds to the Threadpool Id of the threadpool they are associated with.

If the application does not configure the priority of these threads they all default to running at the highest `RTCORBA::Priority` in the installed priority mapping. That is the priority that is returned by the Priority Mapping's `max_priority` method. Hence, with the Default Priority Mapping installed, they will all run at `RTCORBA::Priority 31`, which maps to VxWorks Native Priority 100.

There are two ways of configuring the priority of the different types of internal ORB threads:

- Collectively, by setting a range limit on ORB internal threads. All the above types of thread will all then run at the maximum priority in the specified range.
- On a per-type basis (and in some cases a per-instance basis), through VisiBroker properties.

## Limiting the Internal ORB Thread Priority Range

A range limit is set on internal ORB threads by passing the following argument to `ORB_init`:

```
--ORBRTPriorityRange <min>,<max>
```

`-ORBRTPriorityRange` is given as one argument, and the two values are given together in another argument, separated by a comma. For example:

```

// Prepare arguments for ORB_init
int argc = 3;
char * argv[] = { "app_name", "-ORBRTPriorityRange", "10,17" };

// Initialize ORB
CORBA::ORB_ptr = ORB_init(argc, argv);

```

The two values give the minimum `RTCORBA::Priority` followed by the maximum `RTCORBA::Priority` value that internal ORB threads are permitted to run at. If this argument is given, the VisiBroker internal ORB threads will default to running at the maximum priority that is specified.

If the range is invalid for some reason the `ORB_init` call fails and throws a CORBA system exception. If the range is invalid because one or both of the values is not a valid `RTCORBA::Priority` value, or because min is greater than max, then a `CORBA::BAD_PARAM` exception is thrown. If the range is invalid because one or both of the values is outside of the range supported by the installed Priority Mapping, then a `CORBA::DATA_CONVERSION` exception is thrown.

## Configuring Individual Internal ORB Thread Priorities

The priority of different types (and in one case, different instances) of internal ORB threads may be controlled by specifying values for certain of VisiBroker properties.

In all cases, the priority value is specified as a Real-Time CORBA Priority value. The value must be a valid priority under the installed Priority Mapping:

- `vbroker.logger.default.thread.priority`  
Sets the default priority for Logger Forwarder threads. Must be set no later than the first time that `CORBA::ORB_init` is called. Note that the priority of Logger Forwarder Threads can be set on a per-instance basis using the `VISLogger::forwarder_priority()` method. See the chapter [“VisiBroker Logging”](#) for details.
- `vbroker.se.default.socket.listener.priority`  
Sets the default priority that Listener threads will run at. Can be changed at any time. The current value at the time of Server Engine creation (which occurs during POA creation) is the value used for any new Listeners that are created. Can be overridden, using the next property.
- `vbroker.se.<SE name>.scm.<SCM name>.listener.priority`  
Where `<SE name>` is the name of a Server Engine and `<SCM name>` is the name of a Server Connection Manger. Sets the priority of the Listener thread associated with a specific SCM in a specific Server Engine. Can be set at any time prior to the creation of that Server Engine (which occurs during the creation of the first POA that uses that Server Engine.)
- `vbroker.dsuser.thread.priority`  
Sets the priority at which the ORB’s DSUser thread will run. Must be set no later than the first time that the ORB attempts to communicate with an OSAgent (which is typically when a POA is created, an object is activated or a call to a `_bind` method is made.)
- `vbroker.garbageCollect.thread.priority`  
Sets the priority of all Garbage Collection threads. Can be changed at any time. The current value at the time of Threadpool creation is the value used.

# Protocol Configuration Policies

Real-Time CORBA uses two Policy types, based on a common protocol configuration framework, to enable the selection of protocols on the server and client side of the ORB.

## ServerProtocolPolicy

The ServerProtocolPolicy policy type is used to select communication protocols on the server-side of VisiBroker-RT for C++ applications.

### IDL sample 15 Server Protocol Policy IDL

```
// IDL
module RTCORBA {
  // Locality Constrained interface
  interface ProtocolProperties {};
  struct Protocol {
    IOP::ProfileId protocol_type;
    ProtocolProperties orb_protocol_properties;
    ProtocolProperties transport_protocol_properties;
  };
  typedef sequence <Protocol> ProtocolList;
  // Server Protocol Policy
  const CORBA::PolicyType SERVER_PROTOCOL_POLICY_TYPE = 1236;
  // locality constrained interface
  interface ServerProtocolPolicy : CORBA::Policy {
    readonly attribute ProtocolList protocols;
  };
  interface RTORB {
    ...
    ServerProtocolPolicy create_server_protocol_policy (
      in ProtocolList protocols
    );
  };
};
```

An instance of the ServerProtocolPolicy is created with the RTORB::create\_server\_protocol\_policy() operation. The attribute of the policy is initialized with the parameter of the same name.

A ServerProtocolPolicy allows any number of protocols to be specified. The order of the Protocols in the ProtocolList indicates the order of preference for the use of the different protocols. Information regarding the protocols is placed into IORs in that order, and the client will take that order as the default order of preference for choice of protocol to bind to the object.

The type of protocol is indicated by an IOP::ProfileId, which is an unsigned long. This means that a protocol is defined as a specific pairing of an ORB protocol (such as GIOP) and a transport protocol (such as TCP.) Hence IIOP would be selected, rather than GIOP plus TCP being selected separately. IIOP in particular is represented by the value TAG\_INTERNET\_IIOP (or the value 0, that it is defined as.)

A Protocol type contains a ProfileId plus two ProtocolProperties1, one each for the ORB protocol and the transport protocol.

(VisiBroker-RT for C++ does not use the Protocol Properties as a means of configuring protocols used by the ORB; instead Protocol Properties are configured via VisiBroker Properties. Please refer to [“Server Engines and SCM Configuration”](#) for details.)

### Code example 111 Using the ServerProtocolPolicy to create a ProtocolList

```
//poa_server_engine_policy_bankImpl.h

...
void bank_server()
{
```

```

VISTRY
{
CORBA::Object_var obj;

VISIFNOT_EXCEP
// get a reference to the root POA
obj = orb->resolve_initial_references("RootPOA");
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
    rootPOA_extended = PortableServerExt::POA::_narrow(obj);
VISEND_IFNOT_EXCEP

// Create the policies
CORBA::StringSequence engines;
CORBA::PolicyList policies;

VISIFNOT_EXCEP
    policies.length(4);
    policies[(CORBA::ULong)0] = rootPOA_extended->
        create_lifespan_policy(PortableServer::PERSISTENT);

    policies[(CORBA::ULong)1] = rootPOA_extended->
        create_id_assignment_policy(PortableServer::USER_ID);

// Define the policies for the POA, Server Engine,
// and Server Connection Manager.
engines.length(1);
    engines[0] = CORBA::string_dup("myServerEngine");

    policies[(CORBA::ULong)2] = rootPOA_extended->
        create_server_engine_policy(engines);
VISEND_IFNOT_EXCEP

// Define the RTCORBA Protocol List used in the
// ServerProtocolPolicy
RTCORBA::ProtocolList protocols;

VISIFNOT_EXCEP
    protocols.length(2);
    // MQ example transport
    protocols[0].protocol_type = 0x48454900;
    // IIOP (=TCP/IP)
    protocols[1].protocol_type = IOP::TAG_INTERNET_IOP;
VISEND_IFNOT_EXCEP

RTCORBA::RTORB_var rORB;
VISIFNOT_EXCEP
    CORBA::Object_var resolved =
        orb->resolve_initial_references("RTORB");
    rORB = RTCORBA::RTORB::_narrow(resolved);
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
    policies[(CORBA::ULong)3] =
        rORB->create_server_protocol_policy(protocols);
VISEND_IFNOT_EXCEP

PortableServer::POAManager_var manager;

VISIFNOT_EXCEP
    manager = rootPOA_extended->the_POAManager();
VISEND_IFNOT_EXCEP

PortableServer::POA_var myPOA;

VISIFNOT_EXCEP
// Create our POA with our policies
myPOA = rootPOA_extended->
    create_POA("bank_mq_transport_poa", manager, policies);

...

```



```

void bank_client(const char* ior, const char* name)
{
    VISTRY
    {
        // Define the RTCORBA Protocol List used in the
        // ClientProtocolPolicy
        RTCORBA::ProtocolList protocols;

        protocols.length(2);
        // MQ example transport
        protocols[0].protocol_type = 0x48454900;
        // IIOP (=TCP/IP)
        protocols[1].protocol_type = IOP::TAG_INTERNET_IOP;

        RTCORBA::RTORB_var rORB;
        CORBA::Object_var resolved =
            orb->resolve_initial_references("RTORB");
        rORB = RTCORBA::RTORB::_narrow(resolved);

        CORBA::PolicyList policies;
        policies.length(1);
        policies[(CORBA::ULong)0] =
            rORB->create_client_protocol_policy(protocols);

        // Find PolicyCurrent for this thread
        CORBA::PolicyCurrent_var curr;
        resolved = orb->resolve_initial_references("PolicyCurrent");
        curr = CORBA::PolicyCurrent::_narrow(resolved);

        curr->set_policy_overrides(policies, CORBA::SET_OVERRIDE);

        // Get the manager Id
        PortableServe23r::ObjectId_var managerId =
            PortableServer::string_to_ObjectId("BankManager");

        // Locate an account manager. Give the full POA name and
        // the servant ID.
        Bank::AccountManager_var manager;

    VISIFNOT_EXCEP
        // Convert given IOR string to object reference
        CORBA::Object_var obj = orb->string_to_object(ior);
        manager = Bank::AccountManager::_narrow(obj);
        VISEND_IFNOT_EXCEP

        ...
    }
}

```

# Listening and Dispatch Configuration

*This chapter describes the listening and dispatch mechanism of VisiBroker-RT for C++, how it may be configured and reasons why it may need to be configured.*

## Overview

The listening and dispatch mechanism is the part of the server-side of VisiBroker-RT for C++ that is responsible for detecting new connections and requests from clients ('listening') and, whenever a request is received, obtaining a thread for the request to be executed on ('dispatching').

The following sections describe the entities involved in the listening and dispatch mechanism, how they may be configured and reasons for configuring them.

## When to Configure Listening and Dispatching

Reasons to configure the listening and dispatch properties of VisiBroker-RT for C++ include:

- to make objects reachable at a particular ('well known') host and port
- to make objects reachable via multiple network interfaces
- to make different sets of objects reachable via different network interfaces
- to use one or more pluggable protocols
- to control the maximum number of client connections that a server will support

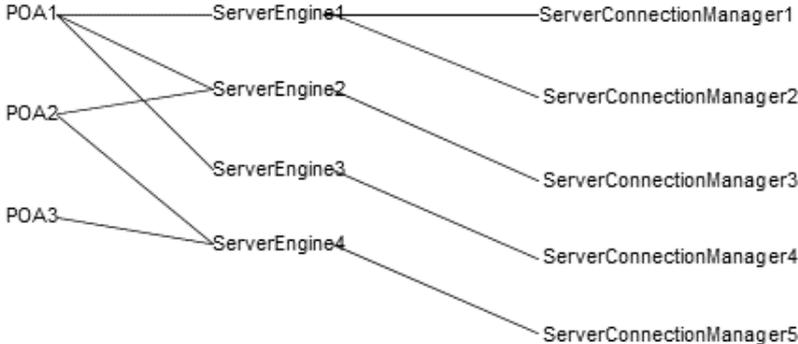
## Listening and Dispatch Architecture

The POA is the primary entity used to configure application objects on the server-side of a VisiBroker-RT for C++ application. But other entities are used to configure the following server-side properties:

- which communication protocols objects may be contacted via,
- what (and how many) protocol endpoints (address and port, for IP networking) objects may be contacted via,
- what (and how many) threads are available to execute calls to those objects and
- garbage collection characteristics for idle connections and threads.

Protocol endpoints (address and port, for IP networking) are represented in VisiBroker-RT for C++ by entities called Server Connection Managers (SCMs). SCMs are contained within entities called Server Engines (SEs). POAs are associated with Server Engines, and hence (indirectly) with SCMs. Which Server Engines a POA is associated with can be specified through a Server Engine policy at the time of POA creation.

A given POA may be associated with any number of Server Engines, and each Server Engine may be associated with any number of POAs. [Figure 29](#) shows the relationships between POAs, Server Engines and Server Connection Managers.



**Figure 29** Relationship between POAs, Server Engines and SCMs

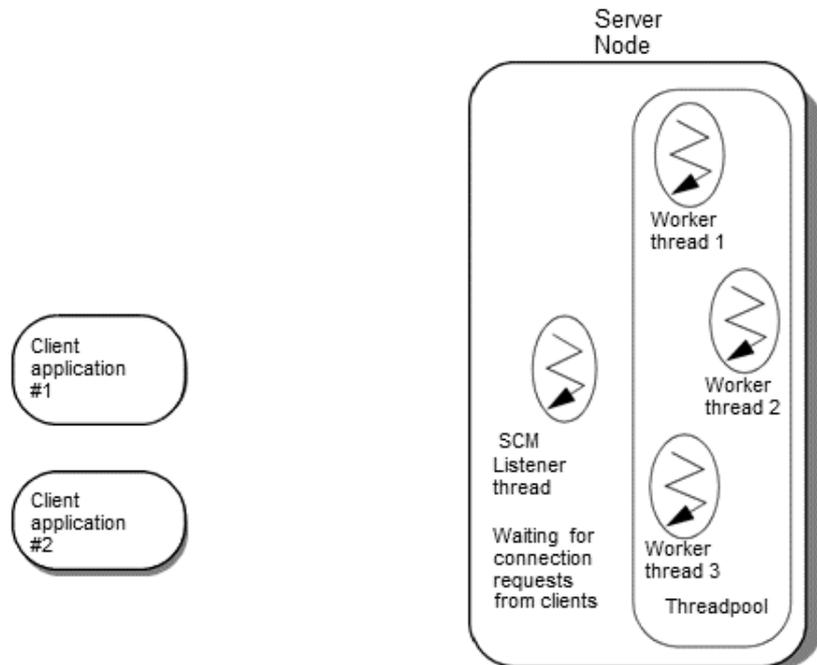
The dispatching properties of VisiBroker-RT for C++ (which and how many threads can be used to execute client requests) are governed by Threadpools. Every POA is associated with exactly one Threadpool, and one Threadpool may be associated with any number of POAs. It is actually the SCMs which interact with Threadpools, at the time of dispatching a client request. The relationship between SCMs and Threadpools is described in the section [“Interaction of an SCM and Threadpool during Dispatch”](#).

VisiBroker Threadpools conform to the Real-Time CORBA specification. For details of how to create and configure Threadpools, and associate them with POAs, see the section [“Threadpools”](#).

## Interaction of an SCM and Threadpool during Dispatch

The diagrams below illustrate the way that an SCM and a Threadpool interact, to perform the dispatch function. (The diagrams start from the point after the SCM and Threadpool are initialized. Initialization of Server

Engines and SCMs, and their association with POAs and Threadpools are discussed in later sections in this chapter.)

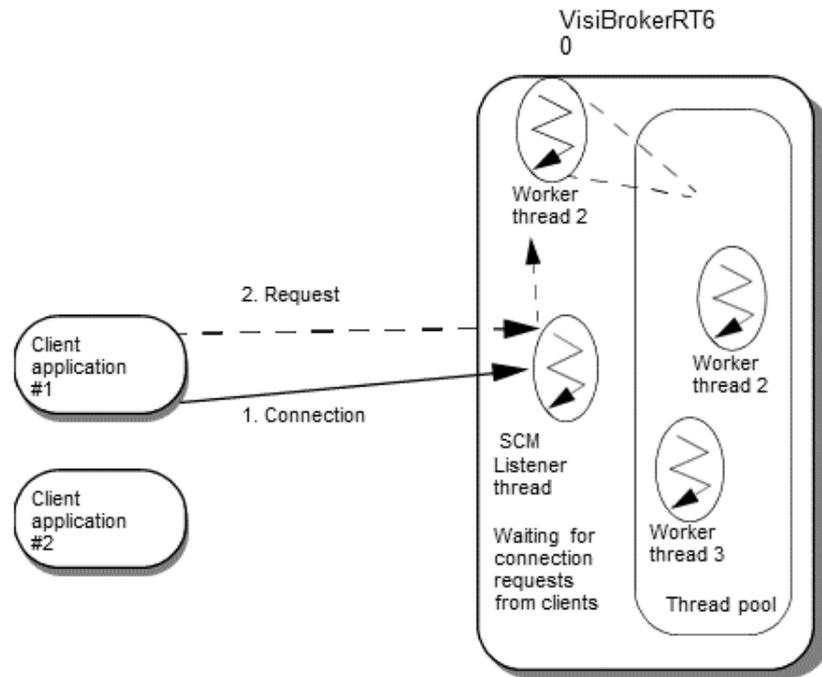


**Figure 30** SCM and Threadpool ready to handle client requests

Figure 30 shows a scenario in which a SCM (contained within a Server Engine) has been initialized, and is ready to handle requests from CORBA clients. The SCM is associated with a Threadpool that contains three 'Worker' threads.

The SCM's Listener thread is shown. This is an additional ORB thread, outside of the Threadpool, that performs the SCM's listening and dispatch functions - waiting for new connections and requests from clients.

Two client CORBA applications are also shown, running on different nodes.

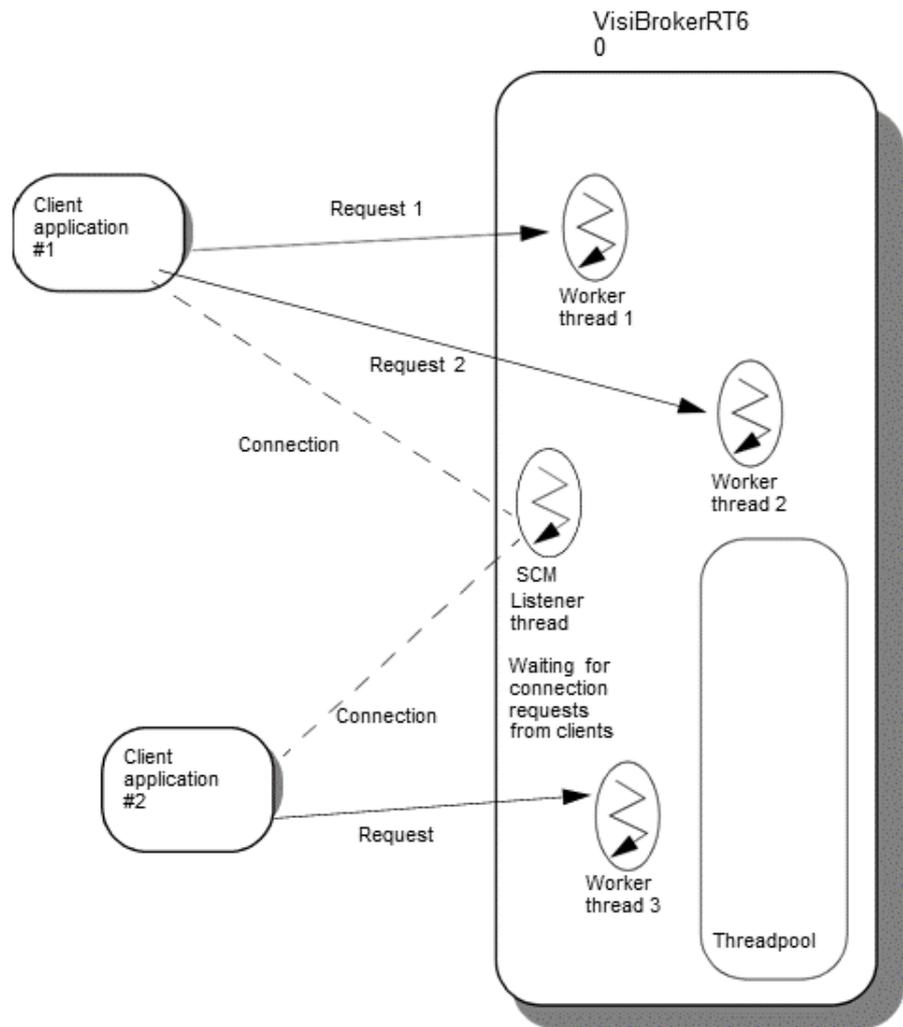


**Figure 31** Client application #1 sends a request

In [Figure 31](#), Client application #1 makes a request on a CORBA object that belongs to a POA that is associated (via a Server Engine) with this SCM.

As this is the first request from the node that the client is running on, a connection must first be established. The client application's ORB initiates the establishment of a connection to the protocol endpoint (host and port number, for IP networking) associated with this SCM. On the server-side, the connection establishment is detected and handled by the SCM's Listener thread.

Once a connection has been established, the client application's ORB sends the client's request. The incoming request is detected by the SCM's Listener thread, which assigns the request to a Worker thread. That Worker thread is removed from the Threadpool and executes the request in the context of the appropriate POA and object implementation. Upon completion of the processing of the request, the Worker thread returns to the Threadpool.



**Figure 32** Client application #1 sends a second request and client application #2 sends a request

Figure 32 shows the situation when the following additional events have occurred:

- Client application #1 has made a second request via this SCM (to either the same or another object that belongs to a POA associated with this SCM). The second request has been made before the first request from that client has finished being processed by Worker thread 1.
- Client application #2 has also made a request on an object that belongs to a POA associated with this SCM.

By default, the second request from Client application #1 is sent over the connection that was established to send the first request. This is because by default VisiBroker-RT for C++ shares connections between all clients and objects on the same pair of nodes, in order to conserve Operating System resources. However, this behavior may be overridden. For details see [“Connection Management”](#).

The request from Client application #2 is sent over a new connection, because this is the first request made from the node that Client application #2 resides on.

Because the second request from Client application #1 was made before the first request had finished being executed and Worker thread 1 had not yet been returned to the Threadpool, a second Worker thread was taken from the pool to execute this request. Similarly, because neither of these requests had finished and returned its Worker thread to the Threadpool before the request from Client application #2 was dispatched, a third Worker thread was taken from the Threadpool to execute that request.

What would happen if a fourth request is received before any of the three current requests finishes executing depends on the configuration of the Threadpool. Either the fourth request will have to wait for a Worker thread to be returned to the Threadpool (if the Threadpool is configured to not dynamically grow beyond three Worker threads), or an extra Worker thread will be created to handle the new request. For details of the Threadpool configuration options, see the section [“Threadpool Creation and Configuration”](#).

## Server Engines and SCM Configuration

Server Engines and the SCMs within them are configured by specifying a number of VisiBroker properties. The properties that can be specified are described in the following sub-sections. For information on how to set properties, see [“Setting Properties”](#).

### Required Server Engine and SCM Properties

The following Server Engine property must be specified, before a Server Engine may be associated with a POA:

- `vbroker.se.<Server_Engine_name>.host`  
This property specifies the host (hostname or dot-notation IP address) that the SCMs contained within this Server Engine will use. This property can be used to select a particular network interface on a machine with multiple network interfaces.
- `vbroker.se.<Server_Engine_name>.scms`  
This property specifies a list (comma or space separated) of the names of the SCMs that this Server Engine will contain.

The following property must be specified for each SCM named in the `scms` property:

- `vbroker.se.<Server_Engine_name>.scm.<SCM_name>.listener.type`  
This property identifies the listener type to be used for this SCM. This corresponds to the protocol to be used. Supported values for VisiBroker-RT for C++ are 'IIOP' and the name of any protocol plugged in through the Pluggable Protocol Interface.

#### Note

On some platforms, VisiBroker also supports a 'LIOP' local IPC protocol. **This is not supported by VisiBroker-RT for C++ for Tornado.**

#### Note

Note that SCM names only have to be unique within the scope of the Server Engine they are contained by. Hence the following is valid:

```
vbroker.se.SE1.scms=iiop1 vbroker.se.SE2.scms=iiop1
```

In this case, there are two Server Engines (named 'SE1' and 'SE2'), each containing an SCM named 'iiop1'. The SCM instances are not shared between Server Engines, and even though some of them have the same name, they are unique and must be configured separately.

## Optional Server Engine Properties

In addition to the above required properties, either of the following two properties may optionally be specified for a Server Engine:

- `vbroker.se.<server_engine_name>.proxyHost`

This property allows a host (hostname or dot-notation IP address) to be specified in IORs that is different to the actual host address that the SCM is listening on. The `.host` value determines the address that the SCM will actually listen on. If no `.proxyHost` value is specified, the `.host` value is also used in the IORs generated for objects belonging to POAs that are associated with this SCM. But if a `.proxyHost` value is specified, that value is used instead.

This property could be used in conjunction with a firewall, or in any other situation where a proxy is required to be contacted rather than contacting the object directly.

Note that both the `.host` and `.proxyHost` property are only for use with the IIOP protocol. If a different protocol is plugged in, via the Pluggable Protocol Interface, the implementation of the plugged in protocol must offer its own properties (at the SCM level) to support configuration of endpoint addressing information.

## Optional SCM Properties

A number of additional properties may be specified for any of the SCMs specified within a Server Engine.

- `vbroker.se.<Server_Engine_name>.scm.<SCM_name>.connectionMax`

This property defines the maximum number of concurrent, incoming connections allowed. The default value is 0, meaning an unlimited number of connections.

- `vbroker.se.<Server_Engine_name>.scm.<SCM_name>.connectionMaxIdle`

This property defines the maximum number of seconds a connection may be idle before it is shut down. The default value is 0, meaning there is no timeout.

- `vbroker.se.<Server_Engine_name>.scm.<SCM_name>.listener.port`

This property defines the listening port that this SCM will use. The default value is 0 (zero), meaning that the system will assign the port number.

- `vbroker.se.<Server_Engine_name>.scm.<SCM_name>.listener.proxyPort`

This property specifies a proxy port number to use with the `.proxyHost` property. The default value, 0 (zero), means that the true port number (assigned via the `.port` property or by the system) will be used in IORs, rather than a proxy value.

Note that, as with the `.host` and `.proxyHost` properties, `.port` and `.proxyPort` are only for use with the IIOP protocol. If a different protocol is plugged in, via the Pluggable Protocol Interface, the implementation of the plugged in protocol must offer its own properties (at the SCM level) to support configuration of endpoint addressing information.

## Server Engine and SCM Creation

A Server Engine (and all the SCMs it is specified as containing) is created automatically by VisiBroker the first time a POA is created that is associated with that Server Engine.

The following sections describe how to associate a POA with particular Server Engines, and the default behavior that occurs if Server Engines are not specified for a particular POA.

## Associating a POA with Server Engines

A POA must be associated with one or more Server Engines. Which Server Engines a POA is to be associated with can be specified at the time of POA creation, by including a `ServerEnginePolicy` in the policy list passed in to the `create_POA` call.

If a `ServerEnginePolicy` is not specified at the time of POA creation, the ORB determines which Server Engines the POA will be associated with. See the section [“Default Server Engines”](#) for details.

Each Server Engine (and the SCMs it contains) is created automatically by VisiBroker the first time a POA is created that is associated with that Server Engine.

If the creation or initialization of a Server Engine (or any of the SCMs it contains) fails for any reason, the `create_POA` call will fail with a `CORBA::INITIALIZE` system exception.

VisiBroker will also log a warning level (level 2) log message explaining the reason for the failure.

The following code sample shows an example of specifying which Server Engines a POA will be associated with. In this case, the POA is associated with two Server Engines, called `'mySE1'` and `'mySE2'`.

**Code example 113** Specifying association with particular Server Engines at time of POA creation

```
// Create sequence of Server Engine names
// (The ServerEnginePolicy requires a sequence, even if only one
// Server Engine is being specified)
CORBA::StringSequence_var engines = new CORBA::StringSequence(2);

engines->length(2);
engines[0] = CORBA::string_dup("mySE1");
engines[1] = CORBA::string_dup("mySE2");

// Place string sequence into an Any
CORBA::Any_var seAny(new CORBA::Any);
seAny <<= engines;

// Create ServerEnginePolicy
CORBA::PolicyList_var policies = new CORBA::PolicyList(1);
policies->length(1);
policies[0] = orb->create_policy(
    PortableServerExt::SERVER_ENGINE_POLICY_TYPE, seAny);

// Create POA using policy
PortableServer::POAManager_var manager =
    rootPOA->the_POAManager();
PortableServer::POA_var myPOA = rootPOA->create_POA(
    "my_poa", manager, policies);
```

## Default Server Engines

If a ServerEnginePolicy is not specified at the time of POA creation, VisiBroker determines which Server Engines the POA will be associated with.

VisiBroker-RT for C++ will associate a POA with a Server Engine named **se\_iiop\_tp<n>**, where <n> is the Id of the Threadpool that that POA is to be associated with. (For details about Real-Time CORBA Threadpools and Threadpool Ids, see “[Threadpools](#)”).

The Server Engine will have the following property values automatically set for it:

```
vbroker.se.se_iiop_tp<n>.scms=scm_iiop_tp<n>
vbroker.se.se_iiop_tp<n>.host=null
vbroker.se.se_iiop_tp<n>.scms.scm_iiop_tp<n>.listener.type=IIOP
```

That is, it will be configured to support IIOP only, and to use a default configuration, including a host and port assigned by the system.

If the POA using the Server Engine has not been explicitly associated with a particular Threadpool, it will default to using the General Threadpool, which has a Threadpool Id of 0. In this case the Server Engine name is `se_iiop_tp0`.

## Restriction on POA/Server Engine Relationship

Each POA is associated with exactly one Threadpool. Each Server Engine must be associated with exactly one Threadpool as well. A Server Engine becomes associated with the same Threadpool as the first POA that it is associated with.

There is one restriction on this relationship:

- It is not possible to associate POAs that use different Threadpools with the same Server Engine.

The first association between a POA and a particular Server Engine will always succeed (because at that time, the Server Engine is created and associated with that POAs Threadpool). But subsequent attempts to associate other POAs with the same Server Engine will fail if the other POAs do not use the same Threadpool as the first POA.

In the case that an association cannot be made between a particular POA and an existing Server Engine, the call to `create_POA` will fail with a `CORBA::INV_POLICY` system exception, and a warning level (level 2) message will be logged.

## Code Example

The code below demonstrates the steps involved in configuring a Server Engine and associating it with a POA. The example stops at the point at which the POA has been created, and is ready to have objects activated on it. A similar code example, based on the VisiBroker bank example, can be found in the VisiBroker-RT for C++ sample application located at

```
<install_location>/VisiBrokerRT60/examples/poa/
server_engine_policy.
```

A Property Table is used to specify the properties required. For more information on Property Tables, see “[Setting Properties](#)”.

**Code example 114** Creating a property table for a server engine (corba\_init.C)

```

...
void do_corba(char * ORB_options_string)
{
#ifdef BUILD_SERVER

// VISPropertyTable defining VisiBroker Properties required for
// Server Engine configuration. Note that the array of property
// strings and the VISPropertyTable object can be destructed
// any time after the ORB_init that uses them.

// Get the property manager; notice the value returned
// is not placed into a 'var' type.
const char * my_properties[] = {
    "vbroker.se.myServerEngine.scms=scml",
    "vbroker.se.myServerEngine.host=null",

    // Define two manager property values
    "vbroker.se.myServerEngine.scm.scml.manager.connectionMax=100",
    "vbroker.se.myServerEngine.scm.scml.manager.connectionMaxIdle=
        300",

    // Define three listener property values
    "vbroker.se.myServerEngine.scm.scml.listener.type=IIOP",
    "vbroker.se.myServerEngine.scm.scml.listener.port=1042",
    "vbroker.se.myServerEngine.scm.scml.listener.proxyPort=0",

    // Define dispatcher property value
    "vbroker.se.myServerEngine.scm.scml.dispatcher.coolingTime=3",

    NULL
};

VISPropertyTable property_table("my_properties",
    my_properties);

cout << "Initialize the server" << endl;
int default_argc = 4;

char *default_argv[] = {"-ORBagentport", OSAGENT_PORT,
    "-ORBpropTable", "my_properties"};

#else

    cout << "Initialize the client" << endl;
    int default_argc = 2;
    char *default_argv[] = {"-ORBagentport", OSAGENT_PORT};

#endif

    char **new_argv;
    int new_argc = VISUtil::stringToArgv(&new_argv, default_argv,
        default_argc,
    ORB_options_string);

    /*-----*/
    /* Call ORB_init */
    /*-----*/
    VISTRY
    {
        // Initialize the ORB
        orb = CORBA::ORB_init(new_argc, new_argv);
    }
    VISCATCH(CORBA::Exception, e)
    {
        cerr << e << endl;
        taskSuspend(0);
    }
    VISEND_CATCH

    return;
}

```

**Code example 115** Creating a POA with a specific server engine (server.C)

```
...
void bank_server()
{
    VISTRY
    {
        CORBA::Object_var obj;

        // get a reference to the root POA
        obj = orb->resolve_initial_references("RootPOA");

        VISIFNOT_EXCEP
        rootPOA_extended = PortableServerExt::POA::_narrow(obj);
        VISEND_IFNOT_EXCEP

        // Create the policies
        CORBA::StringSequence engines;
        CORBA::PolicyList policies;

        // Define the policies for the POA, Server Engine,
        // and Server Connection Manager.
        engines.length(1);
        engines[0] = CORBA::string_dup("myServerEngine");

        policies.length(3);
        VISIFNOT_EXCEP
        policies[(CORBA::ULong)0] = rootPOA_extended->
            create_lifespan_policy(PortableServer::PERSISTENT);
        VISEND_IFNOT_EXCEP

        VISIFNOT_EXCEP
        policies[(CORBA::ULong)1] = rootPOA_extended->
            create_id_assignment_policy(PortableServer::USER_ID);
        VISEND_IFNOT_EXCEP

        VISIFNOT_EXCEP
        policies[(CORBA::ULong)2] = rootPOA_extended->
            create_server_engine_policy(engines);
        VISEND_IFNOT_EXCEP

        PortableServer::POAManager_var manager;
        VISIFNOT_EXCEP
        manager = rootPOA_extended->the_POAManager();
        VISEND_IFNOT_EXCEP

        PortableServer::POA_var myPOA;
        VISIFNOT_EXCEP
        // Create our POA with our policies
        myPOA = rootPOA_extended->
            create_POA("bank_se_policy_poa", manager, policies);
        VISEND_IFNOT_EXCEP

        // Ready to activate objects on the new POA.
        // The objects will be contactable via myServerEngine
        //....
    }
}
```



# Connection Management

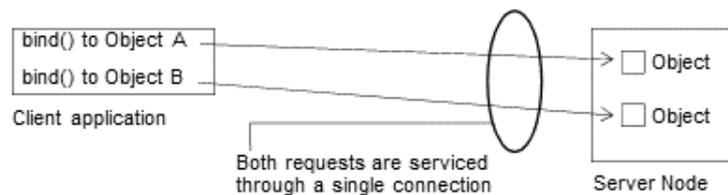
This chapter describes the connection management available in VisiBroker-RT for C++.

## VisiBroker Default Connection Behavior of VisiBrokerRT60

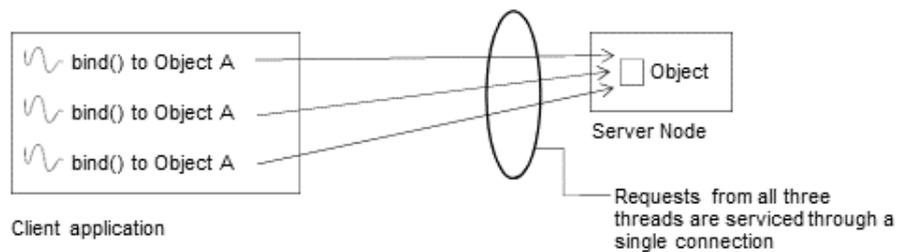
By default, VisiBroker-RT for C++'s connection management minimizes the number of client connections to the server. All client requests from one node to objects on another node are **multiplexed** over the same connection, even if they originate from different threads.

Additionally, released client connections are recycled for subsequent reconnects to the same server, eliminating the need for clients to incur the overhead of establishing a new connection to the server.

In the scenario shown in [Figure 33](#), a client application is bound to two objects on one node. Communication from the client shares a common connection to the server, even though the `targets` are two different objects.



**Figure 33** Binding to two objects on the same node



**Figure 34** Binding to an object from multiple threads on the same node

[Figure 34](#) shows a multi-threaded client that has several threads bound to an object on the same remote node. The invocations from all threads are serviced by the same connection.

# Overriding the Default Behavior with `_clone()`

VisiBroker-RT for C++ provides a `_clone()` operation, that can be called by the application to establish a new, separate connection to an object on a remote node.

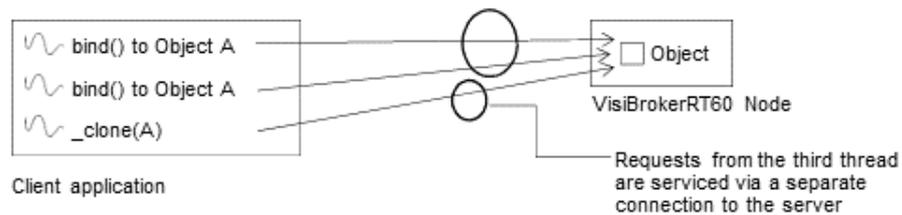
The `_clone()` operation is defined for `CORBA::Object` and for all generated IDL interface types.

**Code example 116** `_clone()` operation is available for `CORBA::Object` and all specific IDL interfaces

```
class CORBA {
    class Object {
        ...
        static CORBA::Object_ptr _clone(CORBA::Object_ptr obj);
        ...
    };
};

// Generated for IDL interface Account
class Account : public virtual CORBA::Object {
    ...
    static Account_ptr _clone(Account_ptr obj);
    ...
};
```

In [Figure 35](#) two threads have called `_bind()` (or obtained a reference to the object on the remote node by some other means) and hence experience the default behavior of sharing a connection to the remote node. The third thread has called `_clone()`, and its requests are serviced via a separate connection.



**Figure 35** `_clone()` establishes a separate connection to a remote node

Note that the different connections are not tied to particular threads. Once a connection has been created by one thread, it can be shared by any number of threads, by sharing or duplicating the same instance of the object reference.

## Limiting the Number of Connections

### Limiting Connections on the Server-Side

The maximum number of concurrent connections that VisiBroker will accept on the server-side can be configured as part of the configuration of the listening and dispatch mechanism. See [“Optional SCM Properties”](#).

The least recently used connections will be recycled when the maximum is reached, ensuring resource conservation.

## Limiting Connections on the Client-Side

The maximum number of concurrent connections that VisiBroker will establish on the client-side can be configured as part of the configuration of the client connection properties. See "Client-Side IIOP Connection properties" in the *VisiBroker-RT for C++ Reference Guide*. The least recently used connections will be recycled when the maximum is reached, ensuring resource conservation.



# Bidirectional Communication

*This chapter explains how to establish bidirectional connections in VisiBroker-RT for C++ without using the Gatekeeper. For information about bidirectional communications when using Gatekeeper, go to the Borland Enterprise Server GateKeeper Guide.*

## Note

Before enabling bidirectional IIOP, please read about "[Security considerations](#)".

## Using bidirectional IIOP

Most clients and servers that exchange information via the Internet are typically protected by corporate firewalls. In systems where requests are initiated only by the clients, the presence of firewalls is usually transparent to the clients. However, there are cases where clients need information asynchronously, that is, information must arrive that is not in response to a request. Client-side firewalls prevent servers from initiating connections back to clients.

Therefore, if a client is to receive asynchronous information, it usually requires additional configuration.

In earlier versions of GIOP and VisiBroker, the only way to make it possible for a server to send asynchronous information to a client was to use a client-side Gatekeeper to handle the callbacks from the server.

If you use bidirectional IIOP, rather than having servers open separate connections to clients when asynchronous information needs to be transmitted back to clients (these would be rejected by client-side firewalls anyway), servers use the client-initiated connections to transmit information to clients. The CORBA specification also adds a new policy to portably control this feature.

Because bidirectional IIOP allows callbacks to be set up without a Gatekeeper, it greatly facilitates deployment of clients.

## Bidirectional ORB properties

Three properties provide bidirectional support:

```
vbroker.orb.enableBiDir=client|server|both|none  
vbroker.se.<sename>.scm.<scmname>.manager.exportBiDir=true|false  
vbroker.se.<sename>.scm.<scmname>.manager.importBiDir=true|false
```

### **vbroker.orb.enableBiDir** property

The `vbroker.orb.enableBiDir` property can be used on both the server and the client to enable bidirectional communication. This property allows you to change an existing unidirectional application into a bidirectional one without

changing any code. The `vbroker.orb.enableBiDir` property may be set to the following values:

Value	Description
<code>client</code>	Enables bidirectional IIOP for all POAs and for all outgoing connections. This setting is equivalent to creating all POAs with a setting of the <code>BiDirectional</code> policy to both and setting the policy override for the <code>BiDirectional</code> policy to both on the VisiBroker ORB level. Furthermore, all created SCMs will permit bidirectional connections, as if the <code>exportBiDir</code> property had been set to true for every SCM.
<code>server</code>	Causes the server to accept and use connections that are bidirectional. This is equivalent to setting the <code>importBiDir</code> property on all SCMs to true.
<code>both</code>	Sets the property to both client and server.
<code>none</code>	Disables bidirectional GIOP altogether. This is the default value.

### **vbroker.se.<sename>.scm.<scmname>.manager.exportBiDir property**

The `vbroker.se.<sename>.scm.<scmname>.manager.exportBiDir` property is a client-side property. By default, it is not set to anything by the VisiBroker ORB. Setting it to true enables creation of a bidirectional callback POA on the specified server engine. Setting it to false disables creation of a bidirectional POA on the specified server engine.

### **vbroker.se.<sename>.scm.<scmname>.manager.importBiDir property**

The `vbroker.se.<sename>.scm.<scmname>.manager.importBiDir` property is a server-side property. By default, it is not set to anything by the VisiBroker ORB. Setting it to true allows the server-side to reuse the connection already established by the client for sending requests to the client. Setting it to false prevents reuse of connections in this fashion.

#### **Note**

These properties are evaluated only once--when the SCMs are created. In all cases, the `exportBiDir` and `importBiDir` properties on the SCMs govern the `enableBiDir` property. In other words, if both properties are set to conflicting values, the SCM-specific properties will take effect. This allows you to set the `enableBiDir` property globally and specifically turn off `BiDir` in individual SCMs.

## **About the examples**

Examples demonstrating use of this feature are located in subdirectories of `examples/bidir-iiop` in the VisiBroker installation directory.

All the examples are based on a simple stock quote callback application:

- The client creates a CORBA object that processes stock quote updates;
- The client sends the object reference of this CORBA object to the server;
- The server invokes this callback object to periodically update stock quotes.

In the sections that follow, these examples are used to explain different aspects of the bidirectional IIOP feature.

# Enabling bidirectional IIOP for existing applications

You can enable bidirectional communication in existing VisiBroker-RT for C++ applications without modifying any source code. A simple callback application that does not use Bidirectional IIOP at all is stored in the `examples/bidir-iiop/basic/` directory.

To enable bidirectional IIOP for this application, you set the `vbroker.orb.enableBiDir` property:

- 1 Make sure the `osagent` is running.
- 2 Initialize the server ORB

```
prompt-> start_corba ("-Dvbroker.orb.enableBiDir=server -
Dvbroker.se.default.local.manager.enabled=false")
```

- 3 Initialize the client ORB

```
prompt-> start_corba ("-Dvbroker.orb.enableBiDir=client -
Dvbroker.se.default.local.manager.enabled=false")
```

- 4 Start the Server

```
prompt-> start_bidir_server
```

- 5 Start the Client

```
prompt-> start_bidir_client
```

The existing callback application now uses bidirectional IIOP and works through a client- side firewall.

## Security considerations

Use of bidirectional IIOP may raise significant security issues. In the absence of other security mechanisms, a malicious client may claim that its connection is bidirectional for use with any host and port it chooses. In particular, a client may specify the host and port of security- sensitive objects not even resident on its host. In the absence of other security mechanisms, a server that has accepted an incoming connection has no way to discover the identity or verify the integrity of the client that initiated the connection. Further, the server might gain access to other objects accessible through the bidirectional connection. This is why use of a separate, bidirectional SCM for callback objects is encouraged. If there are any doubts as to the integrity of the client, it is recommended that bidirectional IIOP not be used.

For security reasons, a server running VisiBroker will not use bidirectional IIOP unless explicitly configured to do so. The property `vbroker.<se>.<sename>.scm.<scmname>.manager.importBiDir` gives you control of bidirectionality on a per-SCM basis. For example, you might choose to enable bidirectional IIOP only on a server engine that uses SSL to authenticate the client, and to not make other, regular IIOP connections available for bidirectional use. (See [“Bidirectional ORB properties”](#) for more information about how to do this.) In addition, on the client-side, you might want to enable bidirectional connections only to those servers that do callbacks outside of the client firewall. To establish a high degree of security between the client and server, you should use SSL with mutual authentication (set `vbroker.security.peerAuthenticationMode` to `REQUIRE_AND_TRUST` on both the client and server).



# VisiBroker Pluggable Transport Interface

*VisiBroker-RT for C++ provides a Pluggable Transport Interface, to support the use of transport protocols besides TCP for the transmission of CORBA invocations. The Interface supports the 'plugging in' of multiple transport protocols simultaneously, and is designed to provide a common interface that is suitable for use with a wide variety of transport types. The interface uses CORBA standard classes wherever possible, but is itself VisiBroker proprietary.*

## NOTE

The library `libpluggable.o` is required when building a VisiBrokerRT60 application to support use of the VisiBroker Pluggable Transport Interface. For a description of all the libraries provided by the VisiBroker-RT for C++ product please refer to [“Step 6: Integrating VisiBroker-RT with Tornado/VxWorks”](#).

## Pluggable Transport Interface Files

The VisiBroker Pluggable Transport Interface is delivered as a library and a supporting header file:

- The library `libpluggable.o` can be found in the `lib` directory of the VisiBroker installation (`.../<installation_dir>/VisiBrokerRT60/lib/<CPU>/`).
- The header file `vptrans.h` can be found in the `include` directory of the VisiBroker installation (`.../<installation_dir>/VisiBrokerRT60/include`).

The library `libpluggable.o` must be linked in, in addition to the ORB library (`liborb_dyn.o` or `liborb_min.o`), in order to use the Pluggable Transport Interface. `vptrans.h` contains the declarations of the types used in the Pluggable Transport Interface. It will be included in the files that the developer writes to interface a given transport protocol to the ORB.

## Transport Layer Requirements

Any transport protocol plugged in to VisiBroker via the Pluggable Transport Interface will be used by the ORB to send and receive messages encoded using the standard GIOP protocol that is defined as part of the CORBA specification.

GIOP makes certain assumptions about the transport layer used to exchange these messages. The same assumptions have been used in the design of the Pluggable Transport Interface.

Therefore, the user code that interfaces a specific transport to the ORB must 'mask' any differences between these requirements and the actual behavior of the transport.

The Pluggable Transport Interface assumes:

- A reliable, bi-directional data exchange channel (connection) is used to send data 'point-to-point' between a single server endpoint of the transport and a single client endpoint of the transport. Thus it is assumed that any reply message from a server may be reliably received by

examining a connection endpoint after a request was sent via that connection. (This does not preclude the ORB from using the same connection to multiplex client requests to the same server.)

- Data sent through the transport is (in principle) unlimited in size and can be viewed as a continuous stream of bytes. All packaging of data and issues related to flow control, package reassembly, and error handling must be hidden.
- Connections can be dynamically opened and closed at the request of the client. The request to open a connection is made on a specific endpoint, which the client obtains from the IOR generated by the server.

Note that the connection request message is not part of the GIOP protocol, but resides in the scope of the pluggable transport connection management and must be handled by the transport specific code.

- A server connection endpoint is described in a way that can be stored in an IOR as specified in the CORBA specification. Such an endpoint must be unique in the transport's addressing scheme and it must be usable at any time to contact the server. Conversion functions must be provided to create a CDR compliant representation of the endpoint address, so it can be used as part of a Profile in an IOR.

## User-Provided Code Required for a Protocol Plugin

Three main classes must be implemented by the user for each transport protocol that is to be plugged in to the ORB via the Pluggable Transport Interface:

### 1 Connection Class

Provides the means to write and read data from the transport layer, associating the data with a particular 'connection' between a client and a server. The use of the concept of a 'connection' does not mean that the physical transport layer used must support connection oriented IO, however the user code must present such a view to the Pluggable Transport Interface and provide all the related functionality described below.

### 2 Listener Class

Represents a server-side 'endpoint' of the transport. It receives client requests to create a 'connection' instance, handles the dynamic opening and closing of such connections, and initiates the 'dispatch' of incoming client requests through open connections.

### 3 Profile Class

Enables the description of the server-side endpoint information of Listener instances in a way that is 'portable', meaning it can be included in an IOR as defined in the CORBA specification, and thus can be exchanged with other ORBs using GIOP or other suitable protocols.

Additionally, the Pluggable Transport Interface uses a "Factory" pattern to manage the instantiation each of these classes. Therefore three Factory classes must be provided, each creating instances of one of the above classes.

A transport protocol is initialized by instantiating the three Factory classes and registering them with the ORB via the Pluggable Protocol Interface. The registration is performed by calling a static function of the Pluggable

Protocol Interface during the system initialization stage, before starting any CORBA server or client code.

## Unique Profile ID Tag

Each plugged in transport is required to have a unique 4 byte Profile ID tag, to distinguish it from other protocols. Profile ID tags are managed by the OMG.

**Note: changed Borland to Micro Focus below.**

Micro Focus has a range of Profile ID tags registered with the OMG, and four of these tags are available for use by protocol plugins:

- 0x48454901 ("HEI\001")
- 0x48454902 ("HEI\002")
- 0x48454903 ("HEI\003")
- 0x48454904 ("HEI\004")

One of these tags should be used rather than a randomly chosen value, to avoid conflict with any third-party CORBA-based products.

Note, however, that there will still be the possibility of conflict, if the system that uses the protocol plugin is integrated with other systems based on VisiBroker-RT for C++ that happen to contain a protocol plugin that choose the same Profile ID tag. This could occur either when different sub-systems, developed independently within the same organization, are integrated, or if the final system is required to interoperate with another CORBA-based system developed by another organization.

**Note: I've updated the link in the following para**

If either of the above scenarios is a serious possibility, a reserved number should be obtained from the OMG. See the OMG FAQ on CORBA tags, available at [http://www4.cs.fau.de/Lehre/WS00/V\\_OODS1/Tutorial/CORBA/ptc/99-02-01.pdf](http://www4.cs.fau.de/Lehre/WS00/V_OODS1/Tutorial/CORBA/ptc/99-02-01.pdf), for details. The minimum number of tags required should be reserved, bearing in mind that a set of tags may normally only be reserved once per year. It is recommended that the numbers only be reserved as the developed system nears deployment.

## Example Code

There are two example transports provided in the 'examples/pluggable' directory of the VisiBroker installation, and an example client and server program using an added transport.

- The 'PROTO' transport is non-functional but contains all the necessary classes and class methods to document the Pluggable Transport API. It compiles and can be loaded and registered with the ORB. However, it will give errors when you try to use it to send ORB requests.
- The 'MQ' transport is functional and is based on shared message queues in the target's address space. It demonstrates in more detail how to implement a transport successfully. Although it is functional, it is only an **unsupported contribution**. Please, **do not use this code for any actual application**.
- The 'mq\_bank' example programs implementing the standard Bank example using the MQ transport layer. It shows how to use the VisiBroker property system to add a new transport to a POA, and how to bind the

client to the server by using the stringified IOR created by the MQ transport library.

The directories contain HTML files that explain how to compile and run these examples.

## Implementing a New Transport

The following sections describe in detail the classes that must be implemented by the user to plug in a new transport protocol into the ORB. Each method is described, and the PROTO and MQ examples should be referred to, to see how they might be implemented and used.

### Connection Class

#### Base Class

VISPTransConnection from file vptrans.h

#### Abstract Methods to be Implemented by Subclass

```
void write(CORBA::Boolean _isFirst, CORBA::Boolean
           _isLast, const char* _data, CORBA::ULong _offset,
           CORBA::ULong _length, CORBA::ULongLong _timeout)

void read(CORBA::Boolean _isFirst, CORBA::Boolean
          _isLast, char* _data, CORBA::ULong _offset,
          CORBA::ULong _length, CORBA::ULongLong _timeout)

void flush() void close()

void connect(CORBA::ULongLong _timeout) CORBA::Long
            id()

CORBA::Boolean isConnected()

CORBA::Boolean isDataAvailable() CORBA::Boolean
               no_callback() CORBA::Boolean isBridgeSignalling()

CORBA::Boolean waitNextMessage(CORBA::ULong _timeout)
IOP::ProfileValue_ptr getPeerProfile()

void setupProfile(const char* prefix,
                 VISPTransProfileBase_ptr peer)
```

#### Other Necessary Methods

Default constructor Destructor

#### Class Description

This class represents a single connection between a server and a client. Whenever a program reads or writes to it, that data will be received or sent to one single peer endpoint on the remote side. When a client wants to send a request to a server, the ORB will look for a valid connection to that server and create one, if it does not exist, yet. The remote endpoint of the connection is setup using the given Profile of the server and communicating with the Listener (see "Listener Class" below) on the server side. Besides

general status information, this class also must either (a) provide a method to wait for data coming through the connection, *that times out* after a given number of seconds, or (b) use the 'Pluggable Transport Bridge' class to perform that function by signalling incoming data to the Bridge when it is available.

## Method Descriptions

`write()`

Sends data through the connection to the remote peer. It does *not* return any error code, but must signal transport related errors by throwing exceptions. The arguments describe a byte array with a given length that needs to be sent. This function *must* either send the complete byte array successfully, timeout, or throw an exception. By default, the timeout is not used (0 value) until the user sets its value to something different, through the VisiBroker property system. Therefore, if this transport does not support timeouts on read/write, it still can be used successfully. In this case the write call must block until all the data has been sent. The call arguments also include two boolean flags that indicate whether this is the first or the last time that data is being sent through the connection.

`read()`

Reads data from the connection sent by the remote peer. It does *not* return any error code, but must signal transport related errors by throwing exceptions. The arguments describe a byte array with a given length that needs to be filled. This function must either fill the complete byte array successfully, timeout, or throw an exception. By default, the timeout is not used (0 value) until the user sets its value to a different value, through the VisiBroker property system. Therefore, if this transport does not support timeouts on read/write, it still can be used successfully. In this case the read call must block until all data has arrived. The call arguments also include two boolean flags, that indicate whether this is the first or the last time data will be read from the connection.

`flush()`

If this transport buffers data, this call is used to flush them on the local side and send/receive all data immediately.

`close()`

Orderly close of a connection on both sides should be performed.

`connect()`

Communicate with the remote peer's 'Listener' instance to setup a new connection on the server side. The function does not return any error code, but should throw exceptions if any transport layer errors occur. By default, the timeout is not used (0 value) until the user sets it to a different value, through the VisiBroker policy system. Therefore, if this transport does not support timeouts on connect, it still can be used successfully. In this case the connect call must block until the connection is established or has failed.

`id()`

This method must return a unique number for each connection instance. The ID only needs to be unique for *this* transport. It is used to lookup/locate a connection instance during request dispatching for this transport.

`isConnected()`

Should return 1 (TRUE), if the remote peer is still connected. If the connection was closed by the peer or any error condition exists that prevents the use of this connection, it must return 0 (FALSE).

`isDataAvailable()`

Should return 1 (TRUE), if data is ready to be read from the connection. Otherwise, it must return 0 (FALSE).

`no_callback()`

Status flag signalling if a connection in this transport can be used to reverse the client/server setup and callback to a servant in the client code. Return 0 (FALSE) if it can not, which will cause the ORB to create a new connection for this kind of call, or 1 (TRUE) if it can. (See the GIOP-1.2 specification from the OMG for details.)

`isBridgeSignalling()`

Flag to tell the ORB to use the bridge to wait for new incoming data *with* a timeout. To optimize the dispatching of requests, new incoming data may be read from a connection that was previously used. This action *must* timeout to free the related thread for other purposes. If this transport can not support such a timeout by itself, 1 (TRUE) must be returned and the Pluggable Transport Bridge (see "Transport Bridge Class" below) is used to perform the timeout logic. Otherwise, 0 (FALSE) is returned and the necessary logic should be implemented in the following method.

`waitNextMessage()`

Block the calling thread until either data has arrived on this connection or the given timeout (in seconds) has expired. Return 1 (TRUE) if data is available, or 0 (FALSE) if not.

`getPeerProfile()`

Returns a copy of the Profile describing the peer endpoint used in this connection. The copy must be created on the heap and the caller is responsible for releasing the used memory. The Profile does not describe the actual connection for this instance, but the Profile of the 'Listener' endpoint used during the 'connect' call.

`setupProfile()`

This call is used to tell a newly created connection what peer it should try to connect to in later steps. (When `connect()` is called.) The given base class should be cast to the expected subclass, if needed, and member data in the connection instance should be initialized from that profile. A prefix string is also passed, for property lookup, in case additional property parameters need to be read.

## Connection Factory Class

### Base Class

`VISPTTransConnectionFactory` from file `vptrans.h`

## Abstract Methods to be Implemented by Subclass

```
VISPTransConnection_ptr create(const char* prefix)
```

## Other Necessary Methods

Constructor, Destructor

## Class Description

This class is used by the Pluggable Transport Interface to generically create a Connection instance for this transport. It is passed to the caller as a pointer to its base class and the virtual functions are used to interface to it.

## Method Description

```
create()
```

Create a new instance and return the pointer to it. The caller is responsible for the memory used by this instance. We pass a string prefix as parameter which can be used to read properties for a connection of this type.

## Listener Class

### Base Class

```
VISPTransListener from file vptrans.h
```

## Abstract Methods to be Implemented by Subclass

```
void setBridge(VISPTransBridge* up)
    IOP::ProfileValue_ptr getListenerProfile() void
    completedData(CORBA::Long id)
```

```
CORBA::Boolean isDataAvailable(CORBA::Long id)
```

```
void destroy()
```

## Other Necessary Methods

None

## Class Description

This class is used by the server-side code to wait for incoming connections and requests from clients. New connections and requests on existing connections are signalled to the ORB via the Pluggable Transport Interface's Bridge class (see "Transport Bridge Class", below).

Instances of this class are created each time a Server Engine is created that includes Server Connection Managers ('SCMs') that specify the particular transport protocol. One instance is created per SCM instance that specifies the protocol.

When a request is received on an existing connection, the connection goes through a 'Dispatch Cycle'. The Dispatch Cycle starts when the connection delivers data to the transport layer. In this initial state, the arrival of this data must be signalled to the ORB via the Bridge (see "Transport Bridge Class", below) and then the Listener ignores the connection until the

Dispatch process is completed (in the mean time, the connection is said to be in the 'dispatch state'). The connection is returned to the initial state when the ORB makes a call to the Listener's completedData() method. During the dispatch state the ORB will read directly from the connection until all requests are exhausted, avoiding any overhead incurred by the Bridge-Listener communication.

In most cases, the transport layer uses blocking calls that wait for new connections. In order to handle this situation, the Listener should be made a subclass of the class VISThread and start a separate thread of execution that can be blocked without holding up the whole ORB. (See the MQ example transport.)

## Method Description

`setBridge()`

This call establishes the 'link' to the Pluggable Transport Bridge instance to be used by this Listener instance. The pointer it passes to the Listener should be stored to allow 'upcalls' to be made into ORB when necessary.

`destroy()`

Instructs the Listener instance to tear down its endpoint and close all related active connections.

`getListenerProfile()`

This call should return the Profile describing the Listeners endpoint on this transport. The returned Profile should be a copy on the heap and the caller (the ORB) takes over memory management of it.

`isDataAvailable()`

Should return 1 (TRUE), if the connection with the given id number has data ready to be read. Returns 0 (FALSE) otherwise. Normally the call should just be forwarded to the transport layer to find out.

`completedData()`

Called when the ORB has completed reading a request for the given id and wants the Listener to once again signal (via the Bridge) any new incoming request.

`Constructor()`

A string prefix can be passed to the constructor to enable the reading of transport specific properties. To support this, the string used in the Listener Factory method needs to be passed.

## Listener Factory Class

### Base Class

`VISPTTransListenerFactory` from file `vptrans.h`

## Abstract Methods to be Implemented by Subclass

```
VISPTransListener_ptr create(const char* propPrefix)
```

## Other Necessary Methods

Constructor, Destructor

## Class Description

This class allows the Pluggable Transport library to provide Listener classes to the ORB when needed. It should create an instance of this transport's Listener and return a pointer to it (as its base class type). The ORB will use the virtual functions to perform 'down calls' into the created instance.

## Method Description

```
create()
```

Make a new instance of this class (optionally passing along the given string prefix). Return a pointer to it. The caller takes over management of this instance.

## Profile Class

### Base Class

VISPTransProfileBase from file vptrans.h

## Abstract Methods to be Implemented by Subclass

```
IOP::ProfileId tag() IOP::TaggedProfile*  
  toTaggedProfile() IOP::ProfileValue_ptr copy()
```

```
CORBA::Boolean matchesTemplate(IOP::ProfileValue_ptr  
  body)
```

## Other Necessary Methods

Default constructor, Destructor

```
static _downcast method accepting  
  'IOP::ProfileValue_ptr' as argument virtual void*  
  _safe_downcast(const VISValueInfo &info) const
```

## Recommended methods

Constructor with 'const IOP::TaggedProfile&' argument  
 Accessor and Mutator methods for any member data

## Class Description

This class provides the functionality to convert between a transport specific endpoint description and an IOP based IOR that can be exchanged with other CORBA implementations. It is also used during the process of binding a client to a server, by passing a ProfileValue to a 'parsing' function that has to return TRUE or FALSE, depending on whether an IOR usable for this transport was found inside of it.

An instance of this class is frequently passed to functions via a pointer to its base class type. In order to support safe runtime downcasting with any C++ compiler, a `'_downcast'` function must be provided that can test if the cast is legal or not. See the 'MQ' example code for an example.

## Method Description

`tag()`

Return the unique tag value for this Profile (see note above).

`toTaggedProfile()`

Return a tagged (stringified) Profile instance created with the values read from this instance's member data.

`copy()`

Make an exact copy on the free store and return a pointer to it. It is good coding practice to use the copy constructor inside of this function.

`matchesTemplate()`

Return 1 (TRUE) if there is an IOR in the given data, that can be used to connect through this transport. Otherwise return 0 (FALSE).

`static _downcast()`

Function to downcast a base class pointer to an instance of this Profile class.

`_safe_downcast()`

Virtual method called by ORB during downcast, to check type info data.

## Profile Factory Class

### Base class

`VISPTransProfileFactory` from file `vptrans.h`

### Abstract Methods to be Implemented by Subclass

```
IOP::ProfileValue_ptr create(const IOP::TaggedProfile&
    profile) CORBA::ULong hash(VISPTransProfileBase_ptr
    prof) IOP::ProfileId getTag()
```

### Other Necessary Methods

Constructor

### Recommended Methods

None

### Class description:

This class is used to generically create a new C++ Profile object, to represent an IOR Profile in memory. It will return a pointer to the new Profile instance, cast to the base type `'IOP::ProfileValue_ptr'`.

## Method description:

`create()`

Read the tagged IOR and create a Profile describing a Listener endpoint.

`hash()`

Support the optimized storage of profiles in a hashed lookup table by calculating a hash number for the given instance. Return 0 if you do not provide hash values.

`getTag()`

Return the unique Profile Id tag for the type of Profile created by this factory.

## Classes Provided by the Interface

Two additional classes are provided by the Pluggable Transport Interface, that the user-provided transport plugin code will make calls to.

### Transport Bridge Class

#### Class name

`VISPTransBridge` in file `vptrans.h`

#### Provided Methods

```
CORBA::Boolean addInput(VISPTransConnection_ptr con)
    void signalDataAvailable(CORBA::Long conId)
```

```
void closedByPeer(CORBA::Long conId)
```

#### Class Description

Generic interface between the transport classes and the ORB. It provides methods to signal various events occurring in the transport layer.

#### Method Description

`addInput()`

Send a connection request to the ORB through the bridge, by passing a pointer to the Connection instance representing the newly established connection. The returned flag signals whether the ORB has accepted the new connection (returns 1 (TRUE)) or refused it (returns 0 (FALSE)). The latter might happen due to resource constraints or due to a restriction on connections (set up through the property system).

`signalDataAvailable()`

Passes to the ORB the connection id of a connection that just got new data from the transport layer. This will start the dispatch cycle for incoming requests.

```
closedByPeer()
```

Tell the ORB that the connection with the given id was closed by the remote peer.

## Transport Registrar Class

### Class Name

VISPTransRegistrar in file vptrans.h

### Provided Methods

```
static void addTransport(const char* protocolName,  
    VISPTransConnectionFactory* connFac,  
    VISPTransListenerFactory* listFac,  
    VISPTransProfileFactory* profFac)
```

### Class Description

This class must be used to register a new transport with the ORB. The string given during registration is used as identifier of this transport and must be unique in the scope of that ORB. It will also be used in the prefix string of properties related to this transport.

### Method Description

```
addTransport()
```

Register the transport identifier string and the three Factory instances used to create specific classes for this transport. This method is static and can therefore be called at any time during the initialization of the ORB.

## Creating a Loadable Library

After compiling all classes described in the above chapter, you have to link this code to the ORB library before you start any server or client.

Create an object file linking all transport specific object code and the Pluggable Transport library code.

You must link this file with ORB into the kernel to plug in the new transport.

# Using Portable Interceptors

*This chapter provides an overview of Portable Interceptors. It discusses several Portable Interceptor examples and including the advanced features of Portable Interceptor factories.*

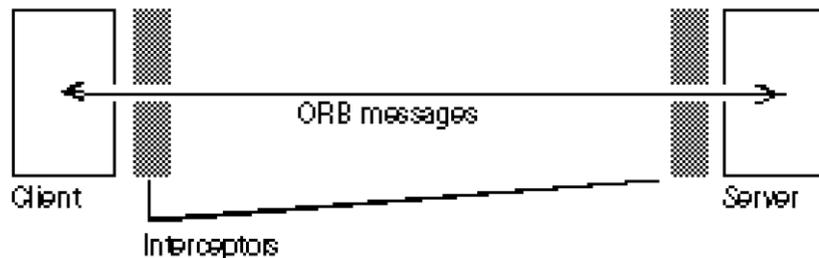
*For a complete description of Portable Interceptor, refer to OMG Final Adopted Specification, ptc/2001-04-03, Portable Interceptors.*

## Overview

The VisiBroker-RT for C++ ORB provides a set of interfaces known as interceptors which provide a framework for plugging-in additional ORB behavior such as security, transactions, or logging. These interceptor interfaces are based on a *callback* mechanism. For example, using the interceptors, you can be notified of communications between clients and servers, and modify these communications if you wish, effectively altering the behavior of the VisiBroker ORB.

At its simplest usage, the interceptor is useful for tracing through code. Because you can see the messages being sent between clients and servers, you can determine exactly how the ORB is processing requests.

If you are building a more sophisticated application such as a monitoring tool or security layer, interceptors give you the information and control you need to enable these lower-level applications. For example, you could develop an application that monitors the activity of various servers and performs load balancing.



**Figure 36** How Interceptors work

There are two types of interceptors supported by the VisiBroker ORB; they are Portable Interceptors and VisiBroker Interceptors. Portable Interceptors are an OMG standardized feature that allows writing of portable code as interceptors, which can be used with different ORB vendors. VisiBroker Interceptors are specific for VisiBroker-RT for C++. See [“Using VisiBroker Interceptors”](#) for more information on VisiBroker Interceptors.

There are two kinds of Portable Interceptors defined by the OMG specification:

- Request Interceptors can enable the VisiBroker ORB services to transfer context information between clients and servers. Request Interceptors are further divided into Client Request Interceptors and Server Request Interceptors.
- An IOR interceptor is used to enable a VisiBroker ORB service to add information in an IOR describing the server's or object's ORB-service-related capabilities. For example, a security service (like SSL) can add its tagged component into the IOR so that clients recognizing that

component can establish the connection with the server based on the information in the component.

For more details on Portable Interceptors, see the *VisiBroker-RT for C++ Programmer's Reference*.

For more details on using both Portable Interceptors and VisiBroker Interceptors, see ["Using VisiBroker Interceptors"](#).

## Portable Interceptor and Information interfaces

All Portable Interceptors implement one of the following base interceptor API classes which are defined and implemented by the VisiBroker ORB:

- Request Interceptor:
  - ClientRequestInterceptor
  - ServerRequestInterceptor
- IORInterceptor

All the interceptor classes listed above are derived from a common class: Interceptor. This Interceptor class has defined common methods that are available to its inherited classes.

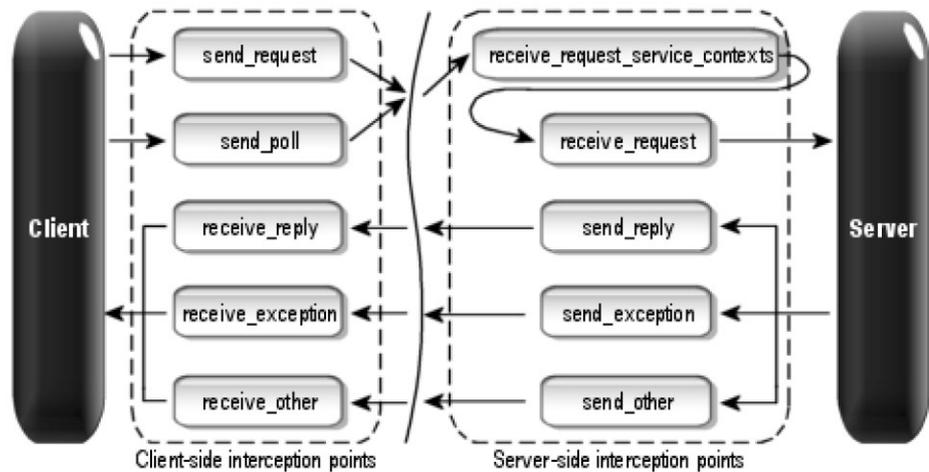
The Interceptor class:

```
class PortableInterceptor::Interceptor
{
    virtual char* name() = 0; virtual void destroy() = 0;
}
```

### Request Interceptor

A request interceptor is used to intercept flow of a request/reply sequence at specific interception points so that services can transfer context information between clients and servers. For each interception point, the VisiBroker ORB gives an object through which the Interceptor can access request information. There are two kinds of Request Interceptor and their respective request information interfaces:

- ClientRequestInterceptor and ClientRequestInfo
- ServerRequestInterceptor and ServerRequestInfo



**Figure 37** Request Interception Points

For more detail information on Request Interceptors, see the *VisiBroker Programmer's Reference*.

## ClientRequestInterceptor

ClientRequestInterceptor has its interception points implemented on the client-side. There are five interception points defined in ClientRequestInterceptor by OMG as shown in the table below, "ClientRequestInterceptor interception points."

**Table 21** ClientRequestInterceptor interception points

Interception points	Description
<code>send_request</code>	lets a client-side Interceptor query a request and modify the service context before the request is sent to the server.
<code>send_poll</code>	lets a client-side Interceptor query a request during a Time-Independent Invocation (TII) polling get reply sequence.  <b>Note</b> that TII is not implemented in the VisiBroker ORB. As a result, the <code>send_poll( )</code> interception point will <i>never</i> be invoked.
<code>receive_reply</code>	lets a client-side Interceptor query the reply information after it is returned from the server and before the client gains control.
<code>receive_exception</code>	lets a client-side Interceptor query the exception's information, when an exception occurs, before the exception is sent to the client
<code>receive_other</code>	lets a client-side Interceptor query the information which is available when a request result other than normal reply or an exception is received.

For more information on each interception point, see the *VisiBroker-RT for C++ Programmer's Reference*.

## ClientRequestInterceptor class

```
class _VISEXPORT ClientRequestInterceptor: public virtual
Interceptor
{
public:
    virtual void send_request(ClientRequestInfo_ptr _ri) = 0;
    virtual void send_poll(ClientRequestInfo_ptr _ri) = 0;
    virtual void receive_reply(ClientRequestInfo_ptr _ri) = 0;
    virtual void receive_exception(ClientRequestInfo_ptr _ri) =
0;
    virtual void receive_other(ClientRequestInfo_ptr _ri) = 0;
};
```

The client-side rules are listed as below:

- The starting interception points are: `send_request` and `send_poll`. On any given request/ reply sequence, one and only one of these interception points is called.
- The ending interception points are: `receive_reply`, `receive_exception` and `receive_other`.
- There is no intermediate interception point.
- An ending interception point is called if and only if `send_request` or `send_poll` runs successfully.
- A `receive_exception` is called with the system exception `BAD_INV_ORDER` with a minor code of 4 (ORB has shutdown) if a request is canceled because of ORB shutdown.
- A `receive_exception` is called with the system exception `TRANSIENT` with a minor code of 3 if a request is canceled for any other reason.

Successful invocations	<code>send_request</code> is followed by <code>receive_reply</code> - a start point is followed by an end point
Retries	<code>send_request</code> is followed by <code>receive_other</code> - a start point is followed by an end point

---

## ServerRequestInterceptor

`ServerRequestInterceptor` has its interception points implemented on the server-side. There are five interception points defined in `ServerRequestInterceptor`. The table below shows the `ServerRequestInterceptor` Interception points.

**Table 22** `ServerRequestInterceptor` Interception points

Interception points	Description
<code>receive_request_service_contexts</code>	lets a server-side Interceptor get its service context information from the incoming request and transfer it to <code>PortableInterceptor::Current</code> 's slot.
<code>receive_request</code>	lets a server-side Interceptor query request information after all information, including operation parameters, is available.
<code>send_reply</code>	lets a server-side Interceptor query reply information and modify the reply service context after the target operation has been invoked and before the reply is returned to the client.
<code>send_exception</code>	lets a server-side Interceptor query the exception's information and modify the reply service context, when an exception occurs, before the exception is sent to the client.
<code>send_other</code>	lets a server-side Interceptor query the information which is available when a request result other than normal reply or an exception is received.

---

For more detail on each interception point, see the *VisiBroker-RT for C++ Programmer's Reference*.

### ServerRequestInterceptor class

```
class _VISEXPORT ServerRequestInterceptor: public virtual
Interceptor
{
public:
    virtual void
    receive_request_service_contexts(ServerRequestInfo_ptr _ri) = 0;
    virtual void receive_request(ServerRequestInfo_ptr _ri) = 0;
    virtual void send_reply(ServerRequestInfo_ptr _ri) = 0;
    virtual void send_exception(ServerRequestInfo_ptr _ri) = 0;
    virtual void send_other(ServerRequestInfo_ptr _ri) = 0;
};
```

The server-side rules are listed as below:

- The starting interception point is: `receive_request_service_contexts`. This interception point is called on any given request/reply sequence.
- The ending interception points are: `send_reply`, `send_exception` and `send_other`. On any given request/reply sequence, one and only one of these interception points is called.
- The intermediate interception point is `receive_request`. It is called after `receive_request_service_contexts` and before an ending interception point.
- On an exception, `receive_request` may not be called.
- An ending interception point is called if and only if `send_request` or `send_poll` runs successfully.
- A `send_exception` is called with the system exception `BAD_INV_ORDER` with a minor code of 4 (ORB has shutdown) if a request is canceled because of ORB shutdown.
- A `send_exception` is called with the system exception `TRANSIENT` with a minor code of 3 if a request is canceled for any other reason.

### Successful invocations

The order of interception points: `receive_request_service_contexts`, `receive_request`, `send_reply` - a start point is followed by an intermediate point which is followed by an end point.

## IOR Interceptor

### IORInterceptor

IORInterceptors give applications the ability to add information describing the server's or object's ORB service related capabilities to object references to enable the VisiBroker ORB service implementation in the client to function properly. This is done by calling the interception point, `establish_components`. An instance of `IORInfo` is passed to the interception point. For more information on `IORInfo`, see the *VisiBroker-RT for C++ Programmer's Reference*.

### IORInterceptor class

```
class _VISEXPORT IORInterceptor: public virtual Interceptor
{
public:
```

```

virtual void establish_components(IORInfo_ptr _info) = 0;
virtual void components_established(IORInfo_ptr _info) = 0;
virtual void adapter_manager_state_changed(
    CORBA::Long _id, CORBA::Short _state) = 0;
virtual void adapter_state_changed(
    const ObjectReferenceTemplateSeq& _templates,
    CORBA::Short _state) = 0;
};

```

## Portable Interceptor Current

The `PortableInterceptor::Current` object (hereafter referred to as `PICurrent`) is a table of slots that can be used by Portable Interceptors to transfer thread context information to request context. Use of `PICurrent` may not be required. However, if a client's thread context information is required at interception point, `PICurrent` can be used to transfer this information.

`PICurrent` is obtained through a call to:

### PortableInterceptor::Current class

```

class _VISEXPORT Current: public virtual CORBA::Current, public
virtual CORBA_Object
{
public:
    virtual CORBA::Any* get_slot(CORBA::ULong _id);
    virtual void set_slot(CORBA::ULong _id, const CORBA::Any&
        _data);
};

```

## Codec

The Codec provides a mechanism for interceptors to transfer components between their IDL data types and their CDR encapsulation representations. A Codec is obtained from `CodecFactory` (see "[CodecFactory](#)").

### Codec class

```

class _VISEXPORT Codec
{
public:
    virtual CORBA::OctetSequence* encode(const CORBA::Any& _data)
        = 0;

    = 0;
    virtual CORBA::Any* decode(const CORBA::OctetSequence& _data)
    virtual CORBA::OctetSequence* encode_value(const CORBA::Any&
        _data) = 0;
    virtual CORBA::Any* decode_value(const CORBA::OctetSequence&
        _data,
        CORBA::TypeCode_ptr _tc) = 0;
};

```

## CodecFactory

This class is used to create a Codec object by specifying the encoding format, the major and minor versions. `CodecFactory` can be obtained a call to:

```

ORB->resolve_initial_references("CodecFactory")

```

## CodecFactory class

```
class _VISEXPORT CodecFactory
{
public:
    virtual Codec_ptr create_codec(const Encoding& _enc) = 0;
};
```

## Creating a Portable Interceptor

The generic steps to create a Portable Interceptor are:

- The Interceptor must be inherited from one of the following Interceptor interfaces:
  - ClientRequestInterceptor
  - ServerRequestInterceptor
  - IORInterceptor
- The Interceptor implements one or more interception points that are available to the Interceptor.
- The Interceptor can be named or anonymous. All names must be unique among all Interceptors of the same type. However, any number of anonymous Interceptors can be registered with the VisiBroker ORB.

### Example 17 Example of Creating a PortableInterceptor in C++

```
#include "PortableInterceptor_c.hh"

class SampleClientRequestInterceptor: public
PortableInterceptor::ClientRequestInterceptor
{
    char * name() {
        return "SampleClientRequestInterceptor";
    }

    void send_request(ClientRequestInfo_ptr _ri) {
        ..... // actual interceptor code here
    }

    void send_request(ClientRequestInfo_ptr _ri) {
        ..... // actual interceptor code here
    }

    void receive_reply(ClientRequestInfo_ptr _ri) {
        ..... // actual interceptor code here
    }

    void receive_exception(ClientRequestInfo_ptr _ri) {
        ..... // actual interceptor code here
    }

    void receive_other(ClientRequestInfo_ptr _ri) {
        ..... // actual interceptor code here
    }
};
```

## Registering Portable Interceptors

Portable Interceptors must be registered with the VisiBroker ORB before they can be used. To register a Portable Interceptor, an ORBInitializer object must be implemented and registered. Portable Interceptors are instantiated and registered during ORB initialization by registering an associated ORBInitializer object which implements its `pre_init( )` or `post_init( )` method,

or both. The VisiBroker ORB will call each registered ORBInitializer with an ORBInitInfo object during the initializing process.

### ORBInitializer class

```
class _VISEXPORT ORBInitializer
{
public:
    virtual void pre_init(ORBInitInfo_ptr _info) = 0;
    virtual void post_init(ORBInitInfo_ptr _info) = 0;
};
```

### ORBInitInfo class

```
class _VISEXPORT ORBInitInfo
{
public:
    virtual CORBA::StringSequence* arguments() = 0;
    virtual char* orb_id() = 0;
    virtual IOP::CodeCFacFactory_ptr codec_factory() = 0;
    virtual void register_initial_reference(const char* _id,
CORBA::Object_ptr_obj) = 0;
    virtual CORBA::Object_ptr resolve_initial_references(const
char* _id) = 0;
    virtual void add_client_request_interceptor(
ClientRequestInterceptor_ptr _interceptor) = 0;
    virtual void add_server_request_interceptor(
ServerRequestInterceptor_ptr _interceptor) = 0;
    virtual void add_ior_interceptor(IORInterceptor_ptr
_interceptor) = 0;
    virtual CORBA::ULong allocate_slot_id() = 0;
    virtual void register_policy_factory(CORBA::ULong _type,
PolicyFactory_ptr _policy_factory) = 0;
};
```

## Registering an ORBInitializer

To register a ORBInitializer, a global method `register_orb_initializer` is provided. Each service that implements Interceptors provides an instance of ORBInitializer. To use a service, an application:

- 1 calls `register_orb_initializer( )` with the service's ORBInitializer; and makes an instantiating `ORB_Init( )` call with a new ORB identifier to produce a new ORB. During `ORB.init( )`, these ORB properties which begin with `org.omg.PortableInterceptor.ORBInitializerClass` will be collected.
- 2 the `<Service>` portion of each property will be collected.
- 3 an object shall be instantiated with the `<Service>` string as its class name.
- 4 the `pre_init( )` and `post_init( )` methods will be called on that object.
- 5 if there is any exception, the ORB will ignore them and proceed.

### Note

To avoid name collisions, the reverse DNS name convention is recommended. For example, if company ABC has two initializers, it could define the following properties:

```
org.omg.PortableInterceptor.ORBInitializerClass.com.abc.ORBInit1
org.omg.PortableInterceptor.ORBInitializerClass.com.abc.ORBInit2
```

The `register_orb_initializer` method is defined in the `PortableInterceptor` module as:

```
class _VISEXPORT PortableInterceptor {
```

```

    static void register_orb_initializer(ORBInitializer *init);
}

```

## Example

A client-side monitoring tool written by company ABC may have the following ORBInitializer implementation:

### Code example 117 Example of Registering ORBInitializer in C++

```

#include "PortableInterceptor_c.hh"

class MonitoringService: public
    PortableInterceptor::ORBInitializer
{
    void pre_init(ORBInitInfo_ptr _info)
    {
        // instantiate the service's Interceptor.
        Interceptor* interceptor = new MonitoringInterceptor();

        // register the Monitoring's Interceptor.
        _info->add_client_request_interceptor(interceptor);
    }

    void post_init(ORBInitInfo_ptr _info)
    {
        // This init point is not needed.
    }
};

MonitoringService * monitoring_service = new
MonitoringService();
PortableInterceptor::register_orb_initializer(monitoring_servi
ce);

```

## VisiBroker Edition Extensions to Portable Interceptors

### POA scoped Server Request Interceptors

Portable Interceptors specified by OMG are scoped globally. VisiBroker Edition has defined "POA scoped Server Request Interceptor", a public extension to the Portable Interceptors, by adding a new module call PortableInterceptorExt. This new module holds a local interface, IORInfoExt, which is inherited from PortableInterceptor::IORInfo and has additional methods to install POA scoped server request interceptor.

#### IORInfoExt class

```

#include "PortableInterceptorExt_c.hh"

class IORInfoExt: public PortableInterceptor::IORInfo
{
public:
    virtual void add_server_request_interceptor(
        ServerRequestInterceptor_ptr _interceptor) = 0;
    virtual char* full_poa_name();
};

```

## Limitations of VisiBroker Edition Portable Interceptors Implementation

The following are limitations of the Portable Interceptor implementation in VisiBroker for C++:

### ClientRequestInfo:

- `arguments()`, `result()`, `exceptions()`, `contexts()`, and `operation_contexts()` are only available for DII invocations.
- `operation_context()`: not available, `CORBA::NO_RESOURCES` thrown.
- `received_exception()`: available only if typecode info is available (e.g. IDL is compiled with `-typecode_info` and linked into program), otherwise `CORBA::UNKNOWN` is always returned.

### ServerRequestInfo:

- `arguments()`, `result()`, are only available for DSI invocations.
- `exceptions()`, `contexts()`, `operation_context()`: not available, `CORBA::NO_RESOURCES` thrown.
- `sending_exception()`: available only if typecode info is available (e.g. IDL is compiled with `-typecode_info` and linked into program), otherwise `CORBA::UNKNOWN` is always returned.

## Examples

This section discusses how applications are actually written to make use of Portable Interceptors and how each request interceptor is implemented. Each example consists of a set of client and server applications and their respective interceptors written in C++. For more information on the definition of each interface, see the *VisiBroker Programmer's Reference*. It is also advisable that developers who want to make use of Portable Interceptor read the chapter on Portable Interceptors in the latest CORBA specification.

### Example Code

Below is the list of examples that can be found in the directory, `<install_dir>/VisiBrokerRT60/examples/pi`. Each example is being associated with a directory name to better illustrate the objective of that example.

The following sections provide detailed description of the example on `client_server` and an explanation on the example, the compilation procedure, and their execution or deployment.

### Example: client\_server

#### Objective of example

This example demonstrates how easy a Portable Interceptor can be added into an existing CORBA application without altering any code. The Portable Interceptor can be added to any application, both client and server-side, through executing the related application again, together with the specified options or properties which can be configured during runtime.

The client and server application used is similar to the one found in `<install_dir>/VisiBrokerRT60/examples/basic/bank_agent` for UNIX or `<install_dir>\VisiBrokerRT60\examples\basic\bank_agent` for Windows. The entire example was taken out and Portable Interceptors added during runtime configuration. The reason to do so is to provide developers, who are familiar with VisiBroker's Interceptor, a fast way of coding between VisiBroker's Interceptors and OMG specific Portable Interceptors.

## Code explanation

### Importing required packages

To use Portable Interceptor interfaces, the related packages or header files are required to be included. Note that the `ORBInitInfoPackage` is optional if you are using any Portable Interceptors' exceptions, such as `DuplicateName` or `InvalidName`.

Required header files for using Portable Interceptor in C++:

```
#include "PortableInterceptor_c.hh"
#include "IOP_c.hh"
```

To load a client-side request interceptor, a class that uses the interface `ORBInitializer` must be implemented. This is also applicable for server-side request interceptor as far as initialization is concerned. The following example shows the code for loading:

Proper inheritance of a `ORBInitializer` in order to load a server request interceptor:

```
class SampleServerLoader : public
PortableInterceptor::ORBInitializer
```

Notice that each of the object that implements the interface, `ORBInitializer`, is also required to inherit from the object `LocalObject`. This is necessary because the IDL definition of `ORBInitializer` uses the keyword `local`. For more information on the IDL keyword, `local`, see ["Using Valuetypes"](#).

During the initialization of the ORB, each request Interceptor is added through the implementation of the interface, `pre_init()`. Inside this interface, the client request Interceptor is being added through the method, `add_client_request_interceptor()`. The related client request interceptor is required to be instantiated before adding itself into the ORB.

### Client-side request interceptor initialization and registration to the ORB

```
void pre_init(PortableInterceptor::ORBInitInfo_ptr _info) {
    SampleClientInterceptor *interceptor = new
SampleClientInterceptor;
    VISTRY {
        _info->add_client_request_interceptor(interceptor);
    }
    ...
}
```

According to the OMG specification, the required application will register the respective interceptors through the method `register_orb_initializer`. Refer to ["Portable Interceptor and Information interfaces"](#) for more details. `VisiBroker-RT` for C++ provides an optional way of registering these interceptors through dynamically loaded libraries. The advantage of using this method of registering is that the applications do not require changing any code but only the way they are executed.

In order to load the interceptor dynamically, the `VISInit` interface is used. This is similar to the one used in 4.x Interceptors. For more information, refer to ["Using VisiBroker Interceptors"](#). The registration of each interceptor loader is similar within the `ORB_init` implementation.

### Registration of client-side ORBInitializer dynamical loading

```
void ORB_init(int& argc, char* const* argv, CORBA::ORB_ptr orb)
{
    if( _bind_interceptors_installed) return;

    SampleClientLoader *client = new SampleClientLoader();
    PortableInterceptor::register_orb_initializer(client);
}
```

...

## Complete implementation of the client-side interceptor loader

```
// SampleClientLoader.C

#include "PortableInterceptor_c.hh"
#include "IOP_c.hh"

#include "sampleClientInterceptor.h"

#if !defined( DLL_COMPILE ) #include "vinit.h"
#include "corba.h"
#endif

// USE_STD_NS is a define setup by VisiBroker to use the std
namespace

USE_STD_NS

class SampleClientLoader :
public PortableInterceptor::ORBInitializer
{
private:
    short int _interceptors_installed;

#if defined( DLL_COMPILE )
    static SampleClientLoader _instance; #endif

public:
    SampleClientLoader() {
        _interceptors_installed = 0;
    }

    void pre_init(PortableInterceptor::ORBInitInfo_ptr _info) {
        if(_interceptors_installed) return;

        cout << "====>SampleClientLoader: Installing..." << endl;

        SampleClientInterceptor *interceptor = new
        SampleClientInterceptor;

        VISTRY {
            _info->add_client_request_interceptor(interceptor);
            _interceptors_installed = 1;
            cout << "====>SampleClientLoader: Interceptors loaded."
                << endl;
        }
        VISATCH(PortableInterceptor::ORBInitInfo::DuplicateName, e) {
            cout << "====>SampleClientLoader: "
                << e.name << " already installed!" << endl;
        }
        VISAND_CATCHALL {
            cout << "====>SampleClientLoader: other exception
            occurred!"
                << endl;
        }
        VISEND_CATCH
    }

    void post_init(PortableInterceptor::ORBInitInfo_ptr _info) {
    }
};

#if defined( DLL_COMPILE )
class VisiClientLoader : VISInit
{
private:
    static VisiClientLoader _instance; short int _
    bind_interceptors_installed;
};
#endif
```

```

public:
    VisiClientLoader() : VISInit(1) {
        _bind_interceptors_installed = 0;
    }

    void ORB_init(int& argc, char* const* argv, CORBA::ORB_ptr orb) {
        if( _bind_interceptors_installed) return;

        VISTRY {
            SampleClientLoader *client = new SampleClientLoader();
            PortableInterceptor::register_orb_initializer(client);

            _bind_interceptors_installed = 1;
        }
        VISCATCH(const CORBA::Exception, e)
        {
            cerr << e << endl;
        }
        VISEND_CATCH
    }
};

// static instance
VisiClientLoader VisiClientLoader::_instance;

#endif

```

### Implementing the ORBInitializer for a server-side Interceptor

At this stage, the client request interceptor should already have been properly instantiated and added. Subsequent code thereafter only provides exception handling and result display.

Similarly, on the server-side, the server request interceptor is also done the same way except that it uses the, `add_server_request_interceptor()` method to add the related server request interceptor into the ORB.

### Server-side request interceptor initialization and registration to the ORB

```

void pre_init(PortableInterceptor::ORBInitInfo_ptr _info) {
    SampleServerInterceptor *interceptor = new
        SampleServerInterceptor;

    VISTRY {
        _info->add_server_request_interceptor(interceptor);
    }
}

```

This method also applies similarly to loading the server-side ORBInitializer class through a DLL implementation.

### Server-side request ORB Initializer dynamic loading

```

void ORB_init(int& argc, char* const* argv, CORBA::ORB_ptr orb)
{
    if( _poa_interceptors_installed) return;

    SampleServerLoader *server = new SampleServerLoader();
    PortableInterceptor::register_orb_initializer(server);
    ...
}

```

The complete implementation of the server-side interceptor loader follows.

### Complete implementation of the server-side interceptor loader

```

// SampleServerLoader.C
#include "PortableInterceptor_c.hh"
#include "IOP_c.hh"

```

```

#if defined( DLL_COMPILE )
#include "vinit.h"
#include "corba.h"
#endif

#include "sampleServerInterceptor.h"

// USE_STD_NS is a define setup by VisiBroker to use the std
namespace
USE_STD_NS

class SampleServerLoader :
    public PortableInterceptor::ORBInitializer
{
private:
    short int _interceptors_installed;

public:
    SampleServerLoader() {
        _interceptors_installed = 0;
    }

    void pre_init(PortableInterceptor::ORBInitInfo_ptr _info) {

        if(_interceptors_installed) return;

        cout << "=====>SampleServerLoader: Installing..." << endl;

        SampleServerInterceptor *interceptor = new
            SampleServerInterceptor();

        VISTRY {
            _info->add_server_request_interceptor(interceptor);

            _interceptors_installed = 1;
            cout << "=====>SampleServerLoader: Interceptors loaded."
                << endl;
        }

        VISCATCH(PortableInterceptor::ORBInitInfo::DuplicateName, e) {
            cout << "=====>SampleServerLoader: "
                << e.name << " already installed!" << endl;
        }

        VISAND_CATCHALL {
            cout << "=====>SampleServerLoader: other exception
            occurred!"
                << endl;
        }

        VISEND_CATCH
    }

    void post_init(PortableInterceptor::ORBInitInfo_ptr _info) {}
};

#if defined( DLL_COMPILE )

class VisiServerLoader : VISInit
{
private:
    static VisiServerLoader instance; short int
        _poa_interceptors_installed;

public:
    VisiServerLoader() : VISInit(1) {
        _poa_interceptors_installed = 0;
    }

    void ORB_init(int& argc, char* const* argv, CORBA::ORB_ptr
orb) {
        if(_poa_interceptors_installed) return;
        VISTRY {
            SampleServerLoader *server = new SampleServerLoader();
            PortableInterceptor::register_orb_initializer(server);
        }
    }
};

```

```

        _poa_interceptors_installed = 1;
    }
    VISACATCH(CORBA::Exception, e)
    {
        cerr << e << endl;
    }
    }
};

// static instance
VisiServerLoader VisiServerLoader::_instance;

#endif

```

## Implementing the RequestInterceptor for Client- or Server-side Request Interceptor

Upon implementation of either client- or server-side request interceptor, two other interfaces must be implemented. They are `name()` and `destroy()`. The `name()` is important here because it provides the name to the ORB to identify the correct interceptor that it will load and call during any request or reply. According to the CORBA specification, an interceptor may be anonymous, for example, it has an empty string as the name attribute. In this example, the name, `SampleClientInterceptor`, is assigned to the client-side interceptor and `SampleServerInterceptor` is assigned to the server-side interceptor.

Implementation of interface attribute, readonly attribute name:

```

public:
    char *name(void) {
        return _name;
    }

```

## Implementing the ClientRequestInterceptor for Client

For the client request interceptor, it is necessary to implement the interface, `ClientRequestInterceptor`, for the request interceptor to be working properly. When the class implements the interface, five request interceptor methods will be implemented regardless of any implementation. They are `send_request()`, `send_poll()`, `receive_reply()`, `receive_exception()` and `receive_other()`. In addition, the interface for the request interceptor must be implemented before hand. On the client-side interceptor, the following request interceptor point will be triggered in relation to its events.

`send_request` - provides an interception point for querying request information and modifying the service context before the request is sent to the server.

### Implementation of the public void `send_request(ClientRequestInfo ri)` interface

```

void send_request(PortableInterceptor::ClientRequestInfo_ptr ri)
{
    ...
}

```

### Implementation of the void `send_poll(ClientRequestInfo ri)` interface

`send_poll` - provides an interception point for querying information during a Time- Independent Invocation (TII) polling to get reply sequence.

```

void send_poll(PortableInterceptor::ClientRequestInfo_ptr ri) {
    ...
}

```

## Implementation of the void receive\_reply(ClientRequestInfo ri) interface

receive\_reply - provides an interception point for querying information on a reply after it is returned from the server and before control is returned to the client.

```
void receive_reply(PortableInterceptor::ClientRequestInfo_ptr ri)
{
    ...
}
```

## Implementation of the void receive\_exception(ClientRequestInfo ri) interface

receive\_exception - provides an interception point for querying the exception's information before it is raised to the client.

```
void receive_exception(PortableInterceptor::ClientRequestInfo_ptr
ri) {
    ...
}
```

receive\_other - provides an interception point for querying information when a request results in something other than a normal reply or an exception. For example, a request could result in a retry (for example, a GIOP Reply with a LOCATION\_FORWARD status was received); or on asynchronous calls, the reply does not immediately follow the request. However, the control is returned to the client and an ending interception point is called.

```
void receive_other(PortableInterceptor::ClientRequestInfo_ptr ri)
{
    ...
}
```

The complete implementation of the client-side request interceptor follows.

## Complete C++ implementation of the client-side request interceptor

```
// SampleClientInterceptor.h

#include "PortableInterceptor_c.hh"
#include "IOP_c.hh"

// USE_STD_NS is a define setup by VisiBroker to use the std
namespace
USE_STD_NS

class SampleClientInterceptor :
    public PortableInterceptor::ClientRequestInterceptor
{
private:
    char *_name;

    void init(char *name) {
        name = new char[strlen(name)+1];
        strcpy(_name, name);
    }

public:
    SampleClientInterceptor(char *name) {
        init(name);
    }

    SampleClientInterceptor() {
        init("SampleClientInterceptor");
    }

    char *name(void) {
        return _name;
    }

    void destroy(void) {
```

```

        // do nothing here
        cout << "=====>SampleServerLoader: Interceptors unloaded" <<

endl;
}

/**
 * This is similar to VisiBroker 4.x ClientRequestInterceptor,
 *
 * void preinvoke_premarshal(CORBA::Object_ptr target,
 *                          const char* operation,
 *                          IOP::ServiceContextList&servicecontexts,
 *                          VISClosure& closure) = 0;
 */
void send_request(PortableInterceptor::ClientRequestInfo_ptr ri)
{
    cout << "=====> SampleClientInterceptor id " << ri->
        request_id()
            << " send_request => " << ri->operation()
            << ": Target = " << ri->target()
            << endl;
}

/**
 * There is no equivalent interface for VisiBroker 4.x
 * ClientRequestInterceptor.
 */

void send_poll(PortableInterceptor::ClientRequestInfo_ptr ri) {
    cout << "=====> SampleClientInterceptor id "
        << ri->request_id()
            << " send_poll => " << ri->operation()
            << ": Target = " << ri->target()
            << endl;
}

/**
 * This is similar to VisiBroker 4.x ClientRequestInterceptor,
 *
 * void postinvoke(CORBA::Object_ptr target,
 *                const IOP::ServiceContextList& service_contexts,
 *                CORBA_MarshalInBuffer& payload,
 *                CORBA::Environment_ptr env,
 *                VISClosure& closure) = 0;
 *
 * with env not holding any exception value.
 */
void receive_reply(PortableInterceptor::ClientRequestInfo_ptr
ri){
    cout << "=====> SampleClientInterceptor id " << ri-
>request_id()
        << " receive_reply => " << ri->operation()
        << endl;
}

/**
 * This is similar to VisiBroker 4.x ClientRequestInterceptor,
 *
 * void postinvoke(CORBA::Object_ptr target,
 *                const IOP::ServiceContextList& service_contexts,
 *                CORBA_MarshalInBuffer& payload,
 *                CORBA::Environment_ptr env,
 *                VISClosure& closure) = 0;
 *
 * with env holding the exception value.
 */
void receive_exception(PortableInterceptor::ClientRequestInfo_ptr
ri) {
    cout << "=====> SampleClientInterceptor id "
        << ri->request_id()
            << " receive_exception => " << ri->operation()
            << ": Exception = " << ri->received_exception()
            << endl;
}

```

```

}

/**
 * This is similar to VisiBroker 4.x ClientRequestInterceptor,
 *
 * void postinvoke(CORBA::Object_ptr target,
 *                 const IOP::ServiceContextList& service_contexts,
 *                 CORBA_MarshalInBuffer& payload,
 *                 CORBA::Environment_ptr env,
 *                 VISClosure& closure) = 0;
 *
 * with env holding the exception value.
 */
void receive_other(PortableInterceptor::ClientRequestInfo_ptr ri)
{ cout << "====> SampleClientInterceptor id " << ri->
  request_id()
  << " receive_other => " << ri->operation()
  << ": Exception = " << ri->received_exception()
  << ", Reply Status = " << getReplyStatus(ri->reply_status())
  << endl;
}

protected:
  char *getReplyStatus(CORBA::Short status) {
    if(status == PortableInterceptor::SUCCESSFUL)
      return "SUCCESSFUL";
    else if(status == PortableInterceptor::SYSTEM_EXCEPTION)
      return "SYSTEM EXCEPTION";
    else if(status == PortableInterceptor::USER_EXCEPTION)
      return "USER EXCEPTION";
    else if(status == PortableInterceptor::LOCATION_FORWARD)
      return "LOCATION FORWARD";
    else if(status == PortableInterceptor::TRANSPORT_RETRY)
      return "TRANSPORT_RETRY";
    else
      return "invalid reply status id";
  }
};

```

On the server-side interceptor, the following request interceptor point will be triggered in relation to its events.

`receive_request_service_contexts` - provides an interception point for getting service context information from the incoming request and transferring it to `PortableInterceptor::Current` slot. This interception point is called before the Servant Manager.

### Implementation of the void `receive_request_service_contexts` (ServerRequestInfo ri) interface

```

void receive_request_service_contexts(PortableInterceptor::
  ServerRequestInfo_ptr ri) {
  ...
}

```

`receive_request` provides an interception point for querying all the information, including operation parameters.

### Implementation of the void `receive_request` (ServerRequestInfo ri) interface

```

void receive_request(PortableInterceptor::
  ServerRequestInfo_ptr ri)
{
  ...
}

```

`send_reply` provides an interception point for querying reply information and modifying the reply service context after the target operation has been invoked and before the reply is returned to the client.

## Implementation of the void receive\_reply (ServerRequestInfo ri) interface

```
void send_reply(PortableInterceptor::ServerRequestInfo_ptr ri) {  
    ...  
}
```

`send_exception` provides an interception point for querying the exception information and modifying the reply service context before the exception is raised to the client.

## Implementation of the void receive\_exception (ServerRequestInfo ri) interface

```
void send_exception(PortableInterceptor::ServerRequestInfo_ptr  
ri)  
{  
    ...  
}
```

`send_other` provides an interception point for querying the information available when a request results in something other than a normal reply or an exception. For example, a request could result in a retry (such as, a GIOP Reply with a LOCATION\_FORWARD status was received); or, on asynchronous calls, the reply does not immediately follow the request, but control is returned to the client and an ending interception point is called.

## Implementation of the void receive\_other (ServerRequestInfo ri) interface

```
void send_other(PortableInterceptor::ServerRequestInfo_ptr ri) {  
    ...  
}
```

All the interception points allow both the client and server to obtain different types of information at different points of an invocation. In the example, this information is displayed as a debugging tool.

The following code example shows the complete implementation of the server-side request interceptor:

### Example 18 Complete C++ implementation of the server-side request interceptor

```
// SampleServerInterceptor.h  
  
#include "PortableInterceptor_c.hh"  
#include "IOP_c.hh"  
  
// USE_STD_NS is a define setup by VisiBroker to use the std  
namespace  
USE_STD_NS  
  
class SampleServerInterceptor :  
    public PortableInterceptor::ServerRequestInterceptor  
{  
private:  
    char *_name;  
  
    void init(char *name) {  
        _name = new char[strlen(name)+1];  
        strcpy(_name, name);  
    }  
  
public:  
    SampleServerInterceptor(char *name) {  
        init(name);  
    }  
  
    SampleServerInterceptor() {  
        init("SampleServerInterceptor");  
    }  
}
```

```

char *name(void) {
return _name;
}

void destroy(void) {
// do nothing here
cout << "=====>SampleServerLoader: Interceptors unloaded"
<< endl;
}
/**
* This is similar to VisiBroker 4.x ClientRequestInterceptor,
*
* void preinvoke_premarshal(CORBA::Object_ptr target,
*                          const char* operation,
*                          IOP::ServiceContextList&
*                          servicecontexts,
*                          VISClosure& closure) = 0;
*/
void
receive_request_service_contexts(PortableInterceptor::ServerRequest
Info_ptr
ri) {
cout << "=====> SampleServerInterceptor id " << ri->
request_id()
<< " receive_request_service_contexts => " << ri->operation()
<< endl;
}

/**
* There is no equivalent interface for VisiBroker 4.x
* SeverRequestInterceptor.
*/
void receive_request(PortableInterceptor::ServerRequestInfo_ptr
ri)
{
cout << "=====> SampleServerInterceptor id " << ri->request_id()
<< " receive_request => " << ri->operation()
<< ": Object ID = " << ri->object_id()
<< ", Adapter ID = " << ri->adapter_id()
<< endl;
}

/**
* There is no equivalent interface for VisiBroker 4.x
* SeverRequestInterceptor.
*/
void send_reply(PortableInterceptor::ServerRequestInfo_ptr ri) {
cout << "=====> SampleServerInterceptor id " << ri->request_id()
<< " send_reply => " << ri->operation()
<< endl;
}

/**
* This is similar to VisiBroker 4.x ServerRequestInterceptor,
*
* virtual void postinvoke_premarshal(CORBA::Object_ptr _target,
* IOP::ServiceContextList&_service_contexts,
* CORBA::Environment_ptr _env,
* VISClosure& _closure) = 0;
*
* with env holding the exception value.
*/
void send_exception(PortableInterceptor::ServerRequestInfo_ptr
ri)
{
cout << "=====> SampleServerInterceptor id " << ri->request_id()
<< " send_exception => " << ri->operation()
<< ": Exception = " << ri->sending_exception()
<< ", Reply status = " << getReplyStatus(ri->reply_status())
<< endl;
}
}

```

```

/**
 * This is similar to VisiBroker 4.x ServerRequestInterceptor,
 * virtual void postinvoke_premarshal(CORBA::Object_ptr _target,
 *                                   IOP::ServiceContextList& _service_contexts,
 *                                   CORBA::Environment_ptr _env,
 *                                   VISClosure& _closure) = 0;
 *
 * with env holding the exception value.
 */
void send_other(PortableInterceptor::ServerRequestInfo_ptr ri) {
    cout << "====> SampleServerInterceptor id " << ri->request_id()
         << " send_other => " << ri->operation()
         << ": Exception = " << ri->sending_exception()

         << ", Reply Status = " << getReplyStatus(ri->reply_status())
         << endl;
}

protected:
char *getReplyStatus(CORBA::Short status) {
    if(status == PortableInterceptor::SUCCESSFUL)
        return "SUCCESSFUL";
    else if(status ==
PortableInterceptor::SYSTEM_EXCEPTION) return "SYSTEM_EXCEPTION";
else if(status == PortableInterceptor::USER_EXCEPTION)
    return "USER_EXCEPTION";
    else if(status == PortableInterceptor::LOCATION_FORWARD)
        return "LOCATION_FORWARD";
    else if(status == PortableInterceptor::TRANSPORT_RETRY)
        return "TRANSPORT_RETRY";
    else
        return "invalid reply status id";
}
};

```

## Developing the Client and Server Application

After the interceptor classes are written, you need to register them with their respective client and server applications.

When running the server and client application on the VxWorks host, the ORB initialization is again performed in the file "corba\_init.C," which contains the registration calls for the respective interceptor loader. For the server ORB there is the file "server\_corba\_init.C", that contains the full example code:

## Implementation of the server ORB initialization

```

#include <vxWorks.h>
#include "corba.h"
#include <taskLib.h>
#include "vutil.h"

#include "sampleServerLoader.C"
#define OSAGENT_PORT "14000"

/*-----*/
/* Forward Declarations.                               */
/*-----*/

extern "C" void start_server_corba(char * ORB_options_string);
static void do_corba(char * ORB_options_string);

/*-----*/
/* Global Variable Declarations                         */
/*-----*/
CORBA::ORB_var orb;

/*-----*/
/* function ==> start_corba                            */
/*-----*/

```

```

/* This function will spawn a vxWork task @                                     */
/* priority 100, which will perform the necessary                             */
/* initialization for the ORB (i.e. ORB_init,...)                             */
/*-----*/

void start_server_corba(char * ORB_options_string)
{
char *          taskName = "DO_CORBA";
int            Prio = 100;
int            option = VX_FP_TASK;
int            stackSize = 20000;

/*-----*/
/* Spawn do_corba task. */
/*-----*/
taskSpawn(taskName,
          Prio,
          option,
          stackSize,
          (FUNCPTR)do_corba,
          (int)ORB_options_string,0,0,0,0,0,0,0,0,0);
}

/*-----*/
/* function ==>do_corba                                                       */
/* This function will perform the necessary                                   */
/* initialization for the ORB (i.e. ORB_init,...)                             */
/*-----*/

void do_corba(char * ORB_options_string)
{
/*-----*/
/* ORB_init options can be specified in two ways.                           */
/* 1) By calling start_corba and specifying the                               */
/*    ORB initialization string                                              */
/*    (e.g. start_corba("-ORBagentport 19000")                               */
/* 2) Programatically by specifying the                                       */
/*    ORB initialization options in the                                       */
/*    default_argc and default_argv variables below.                         */
/*-----*/
/* PLEASE NOTE THAT THE OPTIONS PASSED IN VIA start_corba                  */
/* OVERRIDE THE OPTIONS THAT ARE SET PROGRAMATICALLY.                       */
/*-----*/

int default_argc = 2;
char *default_argv[] = {"-ORBagentport", OSAGENT_PORT};
char **new_argv;
int new_argc = VISUtil::stringToArgv(&new_argv, default_argv,
default_argc, ORB_options_string);

/*-----*/
/* Call ORB_init */
/*-----*/
VISTRY
{
// Instantiate an interceptor loader before initializing the orb:
SampleServerLoader* loader = new SampleServerLoader();
PortableInterceptor::register_orb_initializer(loader);

// Initialize the ORB
orb = CORBA::ORB_init(new_argc, new_argv);

VISUtil::freeArgv(new_argc, new_argv);
}
VISCATCH(CORBA::Exception, e)
{
cerr << e << endl; taskSuspend(0);
}
VISEND_CATCH

return;
}

```

For the case where both the server and client applications run on the same VxWorks node, the same ORB must be used to register both the client and server interceptor loaders. This code can be found in "colocated\_corba\_init.C".

Following the loader registration(s), the client and server application code need to be developed.

## Implementation of the client application

```
#include "corba.h"
#include "bank_c.hh"

// USE_STD_NS is a define setup by VisiBroker to use the std
namespace
USE_STD_NS

/*-----*/
/* Forward Declarations. */
/*-----*/
extern "C" void start_cs_client(void);
static void cs_client(void);

extern CORBA::ORB_var orb;

void start_cs_client(void)
{
    char *      taskName = "CS_CLNT";
    int         Prio = 100;
    int         option = VX_FP_TASK;
    int         stackSize = 20000;

    taskSpawn(taskName,
              Prio,
              option,
              stackSize,
              (FUNCPTR)cs_client,
              0,0,0,0,0,0,0,0,0,0);
}

void cs_client(void)
{
    VISTRY {
        char *name = "Jack B. Quick";
        // Get the manager Id
        PortableServer::ObjectId_var managerId =
            PortableServer::string_to_ObjectId("BankManager");

        // Locate an account manager. Give the full POA name and the
        // servant ID.
        Bank::AccountManager_var manager =
            Bank::AccountManager::_bind("/bank_agent_poa", managerId);

        // Request the account manager to open a named account.
        Bank::Account_var account = manager->open(name);

        // Get the balance of the account.
        CORBA::Float balance = account->balance();

        // Print out the balance.
        cout << "The balance in " << name << "'s account is $" <<
            balance << endl;
    }
    VISACATCH(CORBA::Exception, e) {
        cerr << e << endl;
        return;
    }
    VISEND_CATCH
}
}
```

## Implementation of the server application

```
#include "corba.h"
#include "bankImpl.h"

// USE_STD_NS is a define setup by VisiBroker to use the std
namespace
USE_STD_NS

/*-----*/
/* Forward Declarations. */
/*-----*/

// Static initialization
AccountRegistry AccountManagerImpl::_accounts;

extern "C" void start_cs_server(void);
static void cs_server(void);

extern CORBA::ORB_var orb;

void start_cs_server(void)
{
    char * taskName = "CS_SRVR";
    int Prio = 100;
    int option = VX_FP_TASK;
    int stackSize = 20000;

    taskSpawn(taskName,
              Prio,
              option,
              stackSize,
              (FUNCPTR)cs_server,
              0,0,0,0,0,0,0,0,0,0);
}

void cs_server(void)
{
    VISTRY
    {
        // get a reference to the root POA
        CORBA::Object_var obj = orb->
            resolve_initial_references("RootPOA");
        PortableServer::POA_var rootPOA =
            PortableServer::POA::_narrow(obj);

        CORBA::PolicyList policies;
        policies.length(1);
        policies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy(
            PortableServer::PERSISTENT);

        // get the POA Manager
        PortableServer::POAManager_var
            poa_manager = rootPOA->the_POAManager();

        // Create myPOA with the right policies
        PortableServer::POA_var myPOA = rootPOA->
            create_POA("bank_agent_poa",
                poa_manager,
                policies);

        // Create the servant
        PortableServer::ServantBase_var managerServant = new
            AccountManagerImpl();

        // Decide on the ID for the servant
        PortableServer::ObjectId_var managerId =
            PortableServer::string_to_ObjectId("BankManager");

        // Activate the servant with the ID on myPOA
        myPOA->activate_object_with_id(managerId, managerServant);

        // Activate the POA Manager
    }
}
```

```
poa_manager->activate();

CORBA::Object_var reference = myPOA->
    servant_to_reference(managerServant);
    cout << reference << " is ready" << endl;

}
VISACATCH(CORBA::Exception, e) {
    cerr << e << endl;
    return;
}
VISEND_CATCH
}
```



# Using VisiBroker Interceptors

*This chapter provides an overview of the 4.x VisiBroker interceptors framework, walks through a interceptor example, and describes some advanced features such as interceptor factories and chaining interceptors. Lastly, this section covers the expected behaviors when both Portable and interceptors are used in the same service.*

## Overview

Similar to Portable Interceptors, VisiBroker interceptors offer CORBA services a mechanism to intercept normal flow of execution of the ORB. There are two kinds of VisiBroker-RT for C++ interceptors:

- Client interceptors are system-level interceptors which are called when a method is invoked from a VisiBroker client.
- Server interceptors are system-level interceptors which are called when a method is invoked on a server object.

To use interceptors you declare a class which implements one of the interceptor interfaces. Once you have instantiated an interceptor object, you register it with its corresponding interceptor manager. Your interceptor object will then be notified by its manager whenever, for example, an object has had one of its methods invoked or its parameters marshalled or demarshalled.

### Note

Use object wrappers, described in the chapter [“Using Object Wrappers”](#), if you want to intercept an operation request before it is marshalled on the client-side or if you want to intercept an operation request just before it is processed on the server-side.

## Interceptor interfaces and managers

Interceptor developers derive classes from one or more of the following base interceptor API classes which are defined and implemented by the VisiBroker-RT for C++ ORB.

- Client interceptors
  - BindInterceptor
  - ClientRequestInterceptor
- Server interceptors
- POALifeCycleInterceptor
- ActiveObjectLifeCycleInterceptor
- ServerRequestInterceptor
- IORCreationInterceptor

## Client interceptors

There are currently two kinds of client interceptors and their respective managers:

- `BindInterceptor` and `BindInterceptorManager`
- `ClientRequestInterceptor` and `ClientRequestInterceptorManager`

For more details about client interceptors see the chapter “Interceptor and object wrapper interfaces and classes,” in the *VisiBroker-RT for C++ Reference Guide*.

### BindInterceptor

A `BindInterceptor` object is a global interceptor which is called on the client side before and after binds.

#### Code example 118 BindInterceptor class

```
class _VISEXPORT BindInterceptor : public virtual
VISpseudoInterface {
public:
    virtual IOP::IORValue_ptr bind(IOP::IORValue_ptr ior,
CORBA_Object_ptr obj,
CORBA::Boolean rebind,
VISclosure& closure) = 0;
    virtual IOP::IORValue_ptr bind_failed(IOP::IORValue_ptr ior,
CORBA_Object_ptr object,
VISclosure& closure) = 0;
    virtual void bind_succeeded(IOP::IORValue_ptr ior,
CORBA_Object_ptr object,
CORBA::Long profile_index,
interceptor::InterceptorManagerControl_ptr control,
VISclosure& closure) = 0;
    virtual void exception_occurred(IOP::IORValue_ptr ior,
CORBA_Object_ptr object,
CORBA_Environment_ptr env, VISclosure& closure) = 0;
};
```

### ClientRequestInterceptor

A `ClientRequestInterceptor` object may be registered during a `bind_succeeded` call of a `BindInterceptor` object, and it remains active for the duration of the connection. Two of its methods are called before the invocation on the client object, one (`preinvoke_premarshal`) before the parameters are marshalled and the other (`preinvoke_postmarshal`) after they are. The third method (`postinvoke`) is called after the request has completed.

#### Code example 119 ClientRequestInterceptor class

```
class _VISEXPORT ClientRequestInterceptor : public virtual
VISpseudoInterface {
public:
    virtual void preinvoke_premarshal(CORBA::Object_ptr target,
const char* operation,
IOP::ServiceContextList& servicecontexts,
VISclosure& closure) = 0;
    virtual void preinvoke_postmarshal(CORBA::Object_ptr target,
CORBA_MarshalInBuffer& payload,
VISclosure& closure) = 0;
    virtual void postinvoke(CORBA::Object_ptr target,
const IOP::ServiceContextList& service_contexts,
CORBA_MarshalInBuffer& payload,
CORBA::Environment_ptr env,
VISclosure& closure) = 0;
    virtual void exception_occurred(CORBA::Object_ptr target,
CORBA::Environment_ptr env,
VISclosure& closure) = 0;
};
```

```
};
```

## Server interceptors

There are currently four kinds of server interceptors:

- POALifeCycleInterceptor and POALifeCycleInterceptorManager
- ActiveObjectLifeCycleInterceptor and ActiveObjectLifeCycleInterceptorManager
- ServerRequestInterceptor and ServerRequestInterceptorManager
- IORCreationInterceptor and IORCreationInterceptorManager

For more details about server interceptors see the chapter "Interceptor and object wrapper interfaces and classes," in the VisiBroker-RT for C++ *Reference Guide*.

### POALifeCycleInterceptor

A POALifeCycleInterceptor object is a global interceptor which is called every time a POA is created (via the create method) or destroyed (via the destroy method).

**Code example 120** POALifeCycleInterceptor class

```
class _VISEXPORT POALifeCycleInterceptor : public virtual
VISPPseudoInterface {
public:
    virtual void create(PortableServer::POA_ptr _poa,
        CORBA::PolicyList& _policies,
        IOP::IORValue*& _iorTemplate,
        interceptor::InterceptorManagerControl_ptr _poaAdmin) = 0;
    virtual void destroy(PortableServer::POA_ptr _poa) = 0;
};
```

### ActiveObjectLifeCycleInterceptor

An ActiveObjectLifeCycleInterceptor object is called whenever an object is added to the Active Object Map (via the create method) or after an object has been deactivated and etherealized (via the destroy method). The interceptor may be registered by a POALifeCycleInterceptor on a per-POA basis at POA creation time. **This interceptor may only be registered if the POA has the RETAIN policy.**

**Code example 121** ActiveObjectLifeCycleInterceptor class

```
class _VISEXPORT ActiveObjectLifeCycleInterceptor : public
virtual VISPPseudoInterface {
public:
    virtual void create(const PortableServer::ObjectId& _oid,
        PortableServer_ServantBase* _servant,
        PortableServer::POA_ptr _adapter) = 0;
    virtual void destroy(const PortableServer::ObjectId& _oid,
        PortableServer_ServantBase* _servant,
        PortableServer::POA_ptr _adapter) = 0;
};
```

### ServerRequestInterceptor

A ServerRequestInterceptor object is called at various stages in the invocation of a server implementation of a remote object:

- 1 before the invocation (via the preinvoke method) and

- 2 after the invocation both before and after the marshalling of the reply (via the `postinvoke_premarshal` and `postinvoke_postmarshal` methods respectively).

This interceptor may be registered by a `POALifeCycleInterceptor` object at POA creation time on a per-POA basis.

**Code example 122** `ServerRequestInterceptor` class

```
class _VISEXPORT ServerRequestInterceptor : public virtual
VISPPseudoInterface {
public:
    virtual void preinvoke(
        CORBA::Object_ptr _target,
        const char*_operation,
        const IOP::ServiceContextList& _service_contexts,
        CORBA_MarshalInBuffer& _payload,
        VISPClosure& _closure) = 0;
    virtual void postinvoke_premarshal(
        CORBA::Object_ptr _target,
        IOP::ServiceContextList& _service_contexts,
        CORBA::Environment_ptr _env,
        VISPClosure& _closure) = 0;
    virtual void postinvoke_postmarshal(
        CORBA::Object_ptr _target, CORBA_MarshalOutBuffer&
        _payload,
        VISPClosure& _closure) = 0;
    virtual void exception_occurred(
        CORBA::Object_ptr _target,
        CORBA::Environment_ptr _env,
        VISPClosure& _closure) = 0;
};
```

## IORCreationInterceptor

An `IORCreationInterceptor` object is called whenever a POA creates an object reference (via the `create` method). This interceptor may be registered by a `POALifeCycleInterceptor` at POA creation time on a per-POA basis.

**IDL sample 19** `IORCreationInterceptor` class

```
class _VISEXPORT IORCreationInterceptor : public virtual
VISPPseudoInterface {
public:
    virtual void create(PortableServer::POA_ptr _poa,
        IOP::IORValue*& _ior) = 0;
};
```

## Registering interceptors with the VisiBroker-RT for C++ ORB

Each interceptor interface has a corresponding interceptor manager interface which is used to register your interceptor objects with the ORB. The following steps are those necessary to register an interceptor:

- 1 Get a reference to an `InterceptorManagerControl` object by calling the `resolve_initial_references` method on an ORB object with the parameter "VisiBrokerInterceptorControl".
- 2 Call the `get_manager` method on the `InterceptorManagerControl` object with one of the String values in [Table 23](#). (Be sure to cast the object reference to its corresponding interceptor manager interface.)

**Table 23** String values to pass to the `get_manager` method of the `InterceptorManagerControl` object

Value	Corresponding interceptor interface
ClientRequest	ClientRequestInterceptor
Bind	BindInterceptor
POALifeCycle	POALifeCycleInterceptor
ActiveObjectLifeCycle	ActiveObjectLifeCycleInterceptor
ServerRequest	ServerRequestInterceptor
IORCreation	IORCreationInterceptor

- 3 Create an instance of your interceptor.
- 4 Register your interceptor object with the manager object by calling the `add` method.
- 5 Load your interceptor objects when running your client and server programs.

## Creating interceptor objects

Finally, you need to implement a factory class which creates instances of your interceptor and registers them with the ORB. Your factory class must derive from the `VISInit` class.

### Code example 123 `VISInit` class

```
// in the vinit.h file
class _VISEXPORT VISInit {
public:
    VISInit();
    VISInit(CORBA::Long init_priority);
    virtual ~VISInit();

    // ORB_init is called toward the beginning of
    CORBA::ORB_init()
    virtual void ORB_init(int& /*argc*/,
        char* const* /*argv*/,
        CORBA_ORB* /*orb*/)
    {}

    // ORB_initialized is called at the end of CORBA::ORB_init()
    virtual void ORB_initialized(CORBA_ORB* /*orb*/) {}

    // shutdown is called when CORBA::ORB::shutdown() was called
    // or process shutdown is detected

    virtual void ORB_shutdown() {}
    ...
};
```

### Note

You may also create new instances of your interceptors and register them with the ORB from within other interceptors as in the example below.

## Loading interceptors

To load your interceptor, simply instantiate the factory before the call to `CORBA::ORB_init` in your application.

## Example interceptors

The example interceptor below uses all of the interceptor API methods (listed in the chapter “Interceptor and object wrapper interfaces and classes” in the VisiBroker-RT for C++ *Reference Guide*) so that you can see how these methods are used, and when they are invoked.

### Example code

In “Code listings”, each of the interceptor API methods have been given simple implementations which print out informational messages to the standard output.

There are five example applications in the `examples/interceptors` directory in your VisiBroker-RT for C++ installation:

- `active_object_lifecycle`
- `authenticate`
- `client_server`
- `ior_creation`
- `encryption`

### Client-server interceptors example

To run the example, compile the files as you normally would. Then start up the Server and the Client from the Tornado WindShell as follows:

On VxWorks embedded node 1:

```
-> ld < corba_init
-> ld < server
-> start_corba
-> start_bank_server
```

On VxWorks embedded node 2:

```
-> ld < corba_init
-> ld < client
-> start_corba
-> start_bank_client
```

You specify as ORB services the two classes which implement the `ServiceLoader` interface.

The results of executing the example interceptor are shown in [Table 24](#). The execution by the client and server is listed in sequence.

**Table 24** Results of executing the example interceptor

Client	Server
	=====>SampleServerLoader:Interceptors loaded
	=====>In POA /. Nothing to do.
	=====>In POA bank_agent_poa,
	1 ServerRequest interceptor installed
	Stub[repository_id=IDL:Bank/AccountManager:1.0,key=ServiceId[service=/bank_agent_poa,id={11 bytes: [B][a][n][k][M][a][n][a][g][e][r]]}] is ready.
Bind Interceptors loaded	
=====>	
SampleBindInterceptor bind	
=====>	
SampleBindInterceptor bind_succeeded	
=====>	
SampleClientInterceptor id MyClientInterceptor preinvoke_premarshal	
=> open	
=====>	
SampleClientInterceptor id MyClientInterceptor preinvoke_postmarshal	
	=====>
	SampleServerInterceptor id MyServerInterceptor preinvoke => open
	Created john's account: Stub[repository_id=IDL:Bank/Account:1.0,key=TransientId[poaName=/,id={4 bytes: (0)(0)(0)(0)},sec=0,usec=0]]
=====>	
SampleClientInterceptor id MyClientInterceptor postinvoke	
=====>	
SampleBindInterceptor bind	
=====>	
SampleBindInterceptor bind_succeeded	
=====>	
SampleClientInterceptor id MyClientInterceptor preinvoke_premarshal => balance	
=====>	
SampleClientInterceptor id MyClientInterceptor preinvoke_postmarshal	

Client	Server
<pre> =====&gt; SampleClientInterceptor id MyClientInterceptor postinvoke The balance in john's account is \$245.64 </pre>	<pre> =====&gt; SampleServerInterceptor id MyServerInterceptor postinvoke_premarshal =====&gt; SampleServerInterceptor id MyServerInterceptor postinvoke_postmarshal </pre>

Since the OAD is not running, the `bind()` call fails and the server proceeds. The client binds to the account object, and then calls the `balance()` method. This request is received by the server, processed, and results are returned to the client. The client prints the results.

As shown through the example code and results, the interceptors for both the client and server are installed when the respective ORB instance starts. Information about registering an interceptor is covered in [“Registering interceptors with the VisiBroker-RT for C++ ORB”](#).

## Code listings

The `SampleServerInterceptorLoader.h` file contains the class `POAInterceptorLoader`. This class implements an `ORB_init()` method which is responsible for loading the `POALifeCycleInterceptor` implementation and instantiating the interceptor object. The `POAInterceptorLoader` inherits from the `VISInit` class, and therefore must implement an `ORB_init()` method which will be called by the ORB during the ORB’s execution of `ORB_init()`. Its sole purpose is to install a `POALifeCycleInterceptor` object by creating it and registering it with the `InterceptorManager`.

### Code example 124 SampleServerInterceptorLoader.h

```

#include <iostream.h>
#include "vinit.h"
#include "SamplePOALifeCycleInterceptor.h"

class POAInterceptorLoader : VISInit {
private:
    short int _poa_interceptors_installed;
public:
    POAInterceptorLoader() {
        _poa_interceptors_installed = 0;
    }
    void ORB_init(int& argc, char* const* argv, CORBA::ORB_ptr
orb) {
        if( _poa_interceptors_installed) return;
        cout << "Installing POA interceptors" << endl;
        SamplePOALifeCycleInterceptor *interceptor = new
SamplePOALifeCycleInterceptor;
        // Get the interceptor manager control
        CORBA::Object *object = orb->
resolve_initial_references("VisiBrokerInterceptorControl");
        interceptor::InterceptorManagerControl_var control =
interceptor::InterceptorManagerControl::_narrow(object);
        // Get the POA manager interceptor::InterceptorManager_var
manager = control->get_manager("POALifeCycle");
        PortableServerExt::POALifeCycleInterceptorManager_var
poa_mgr =
PortableServerExt::POALifeCycleInterceptorManager::_narrow(man
ager)
        ;
    }
};

```

```

        // Add POA interceptor to the list
        poa_mgr->add
        ((PortableServerExt::POALifeCycleInterceptor*) interceptor);
        cout << "POA interceptors installed" << endl;
        _poa_interceptors_installed = 1;
    }
};

```

The `SamplePOALifeCycleInterceptor` object is invoked every time a POA is created or destroyed. Because we have two POAs in the `client_server` example, this interceptor is invoked twice, first during `rootPOA` creation and then at the creation of `myPOA`. We install the `SampleServerInterceptor` only at the creation of `MyPOA`.

**Code example 125** `SamplePOALifeCycleInterceptor.h`

```

#include "interceptor_c.hh"
#include "PortableServerExt_c.hh"
#include "IOP_c.hh"
#include "SampleServerInterceptor.h"

class SamplePOALifeCycleInterceptor :
PortableServerExt::POALifeCycleInterceptor {
public:
    void create( PortableServer::POA_ptr poa,
                CORBA_PolicyList& policies,
                IOP::IORValue_ptr& iorTemplate,
                interceptor::InterceptorManagerControl_ptr control) {
        if( strcmp( poa->the_name(),"bank_agent_poa") == 0 ) {
            // Add the Request-level interceptor
            SampleServerInterceptor* interceptor =
                new SampleServerInterceptor("MyServerInterceptor");
            // Get the ServerRequest interceptor manager
            interceptor::InterceptorManager_var generic_manager =
                control->get_manager("ServerRequest");
            interceptor::ServerRequestInterceptorManager_var manager =
                interceptor::ServerRequestInterceptorManager::_narrow(
                    generic_manager);
            // Add the interceptor manager->add(
            (interceptor::ServerRequestInterceptor*) interceptor);
            cout <<"=====>In POA " << poa->the_name()
                << ", 1 ServerRequest interceptor installed"<< endl;
        } else
            cout << "=====>In POA " << poa->the_name() <<
                ". Nothing to do." << endl;
        }
    void destroy( PortableServer::POA_ptr poa) {
        // To be a trace!
        cout << "=====> SamplePOALifeCycleInterceptor
        destroy" <<
            poa->the_name() << endl;
        }
};

```

The `SampleServerInterceptor` object is invoked every time a request is received at or a reply is made by the server.

**Code example 126** `SampleServerInterceptor.h`

```

#include <iostream.h>
#include "vclosure.h"
#include "interceptor_c.hh"
#include "IOP_c.hh"

class SampleServerInterceptor :
interceptor::ServerRequestInterceptor {
private:
    char * _id;
public:
    SampleServerInterceptor( const char* id) {
        _id = new char[ strlen(id)];
        strcpy( _id,id);
    }
};

```

```

}
~SampleServerInterceptor() { _id = NULL;}
void preinvoke( CORBA_Object* target,
  const char* operation,
  const IOP::ServiceContextList& service_contexts,
  CORBA_MarshalInBuffer& payload,
  VISClosure& closure) {
  closure.data = new char[ strlen( _id)];
  strcpy( (char*)(closure.data), _id);
  cout << "=====> SampleServerInterceptor id " <<
    (char*)(closure.data) <<
    " preinvoke => " << operation << endl;
}
void postinvoke_premarshal( CORBA_Object* target,
  IOP::ServiceContextList& service_contexts,
  CORBA::Environment_ptr env,
  VISClosure& closure) {
  cout << "=====> SampleServerInterceptor id " <<
    (char*)(closure.data) <<
    " postinvoke_premarshal " << endl;
}
void postinvoke_postmarshal(CORBA_Object* target,
  CORBA_MarshalOutBuffer& payload,
  VISClosure& closure) {
  cout << "=====> SampleServerInterceptor id " <<
    (char*)(closure.data) <<
    " postinvoke_postmarshal " << endl;
}
void exception_occurred( CORBA_Object* target,
  CORBA::Environment_ptr env,
  VISClosure& closure) {
  cout << "=====> SampleServerInterceptor id " <<
    (char*)(closure.data) <<
    " exception_occurred" << endl;
}
};

```

The `SampleClientInterceptor` is invoked every time a request is made by or a reply is received at the client.

#### Code example 127 `SampleClientInterceptor.h`

```

#include <iostream.h>
#include "interceptor_c.hh"
#include "IOP_c.hh"
#include "vclosure.h"

class SampleClientInterceptor : public
interceptor::ClientRequestInterceptor {
private:
  char * _id;
public:
  SampleClientInterceptor( char * id) {
    _id = new char[ strlen(id)+1];
    strcpy(_id,id);
  }
  void preinvoke_premarshal(CORBA::Object_ptr target,
    const char* operation,
    IOP::ServiceContextList& servicecontexts,
    VISClosure& closure) {

    closure.data = new char[ strlen( _id)];
    strcpy( (char*)(closure.data), _id);
    cout << "SampleClientInterceptor id " << closure.data
      << "=====> preinvoke_premarshal "
      < operation << endl;
  }
  void preinvoke_postmarshal(CORBA::Object_ptr target,
    CORBA_MarshalInBuffer& payload,
    VISClosure& closure) {
    cout << "SampleClientInterceptor id " << closure.data
      << "=====> preinvoke_postmarshal "
      << endl;
  }
};

```

```

    }
    void postinvoke(CORBA::Object_ptr target,
                   const IOP::ServiceContextList& service_contexts,
                   CORBA_MarshallInBuffer& payload,
                   CORBA::Environment_ptr env,
                   VISClosure& closure) {
        cout << "SampleClientInterceptor id " << closure.data
             << "=====> postinvoke "
             << endl;
    }
    void exception_occurred(CORBA::Object_ptr target,
                           CORBA::Environment_ptr env,
                           VISClosure& closure) {
        cout << "SampleClientInterceptor id " << closure.data
             << "=====> exception_occurred "
             << endl;
    }
};

```

The `SampleClientInterceptorLoader.h` file contains the class `BindInterceptorLoader`. This class implements an `ORB_init()` method which is responsible for loading the `SampleBindInterceptor` objects. The `BindInterceptorLoader` inherits from the `VISInit` class, and therefore must implement an `ORB_init()` method which will be called by the ORB during the ORB's execution of `ORB_init()`. Its sole purpose is to install a `BindInterceptor` object by creating it and registering it with the `InterceptorManager`.

The `SampleBindInterceptor` class contains the `bind()`, `bind_succeeded()` and `bind_failed()` methods. These methods are called by the ORB during object binding. When the bind succeeds, `bind_succeeded()` will be called by the ORB and a `BindInterceptor` object is installed by creating it and registering it with the `InterceptorManager`.

#### Code example 128 `SampleClientInterceptorLoader.h`

```

#include <iostream.h>
#include "vinit.h"
#include "SampleBindInterceptor.h"

class BindInterceptorLoader : VISInit {
private:
    short int _bind_interceptors_installed;
public:
    BindInterceptorLoader() {
        _bind_interceptors_installed = 0;
    }
    void ORB_init(int& argc, char* const* argv, CORBA::ORB_ptr
orb) {
        if( _bind_interceptors_installed) return;
        cout << "Installing Bind interceptors" << endl;
        SampleBindInterceptor *interceptor =
            new SampleBindInterceptor;
        // Get the interceptor manager control
        CORBA::Object *object = orb->
resolve_initial_references("VisiBrokerInterceptorControl");
        interceptor::InterceptorManagerControl_var control =
            interceptor::InterceptorManagerControl::_narrow(object);

        // Get the Bind manager
        interceptor::InterceptorManager_var manager =
            control->get_manager("Bind");
        interceptor::BindInterceptorManager_var bind_mgr =
            interceptor::BindInterceptorManager::_narrow(manager);
        // Add Bind interceptor to the list
        bind_mgr->add( (interceptor::BindInterceptor*)interceptor);
        cout << "Bind interceptors installed" << endl;
        _bind_interceptors_installed = 1;
    }
};

```

The `SampleBindInterceptor` is invoked when the client attempts to bind to an object. The first step on the client side after ORB initialization is to bind to an `AccountManager` object. This bind invokes the `SampleBindInterceptor` and a `SampleClientInterceptor` is installed when the bind succeeds.

**Code example 129** `SampleBindInterceptor.h`

```
#include <iostream.h>
#include "interceptor_c.hh"
#include "IOP_c.hh"
#include "vclosure.h"
#include "SampleClientInterceptor.h"

class SampleBindInterceptor : public interceptor::BindInterceptor
{
public:
    IOP::IORValue_ptr bind(IOP::IORValue_ptr ior,
        CORBA_Object_ptr obj,
        CORBA::Boolean rebind,
        VISClosure& closure) {
        cout << "SampleBindInterceptor-----> bind" <<
endl;
        return NULL;
    }

    IOP::IORValue_ptr bind_failed(IOP::IORValue_ptr ior,
        CORBA_Object_ptr object,
        VISClosure& closure) {
        cout << "SampleBindInterceptor----->
            bind_failed" << endl;
        return NULL;
    }
void bind_succeeded(IOP::IORValue_ptr ior,
    CORBA_Object_ptr object,
    CORBA::Long profile_index,
    interceptor::InterceptorManagerControl_ptr control,
    VISClosure& closure) {
    cout << "SampleBindInterceptor----->
        bind_succeeded"
        << endl;
    // Add the Request-level interceptor
    SampleClientInterceptor* interceptor =
        new SampleClientInterceptor((char*)"MyClientInterceptor");
    // Get the ClientRequest interceptor manager
    interceptor::InterceptorManager_var generic_manager =
        control->get_manager("ClientRequest");
    interceptor::ClientRequestInterceptorManager_var manager =
        interceptor::ClientRequestInterceptorManager::_narrow(
            generic_manager);
    // Add the interceptor manager->add(
        (interceptor::ClientRequestInterceptor*)interceptor);
    cout <<"=====>In bind_succeeded, 1 "
        <<"ClientRequest interceptor installed"<< endl;
    }

void exception_occurred(IOP::IORValue_ptr ior,
    CORBA_Object_ptr object,
    CORBA_Environment_ptr env,
    VISClosure& closure) {
    cout << "SampleBindInterceptor----->
        exception_occured"
        << endl;
    }
};
```

## Passing information between your interceptors

`VISClosure` objects are created by the ORB at the beginning of certain sequences of interceptor calls. The same `VISClosure` object is used for all calls in that particular sequence. The `VISClosure` object contains a single public data field, object, of type `void` which may be set by the interceptor to keep state information. The sequences for which `Closure` objects are created vary depending on the interceptor type. In the `ClientRequestInterceptor`, a new `VISClosure` is created before calling `preinvoke_premarshal` and the same `VISClosure` is used for that request until the request completes, successfully or not. Likewise in the `ServerInterceptor`, a new `VISClosure` is created before calling `preinvoke`, and that `VISClosure` is used for all interceptor calls related to processing that particular request.

For an example of how `VISClosure` is used, see the `interceptors/client_server` directory in the `examples` directory in your installation.

## Using both Portable Interceptors and Interceptors simultaneously

Both Portable Interceptors and interceptors can be installed simultaneously with the VisiBroker-RT for C++. However, as they have different implementations, there are several rules of flow and constraints that developers need to understand when using both interceptors, as described below.

### Order of invocation of interception points

The order of invocation of interception points follows the interception point ordering rules of individual versions of interceptors, regardless of whether the developer actually chooses to install one of more than one version.

### Client side interceptors

When both Portable and VisiBroker client side interceptors are installed, the order of events, (assuming no interceptor throws an exception) is:

- 1 `send_request` (Portable Interceptor), followed by `preinvoke_premarshal` (interceptors)
- 2 construct request message
- 3 `preinvoke_postmarshal` (interceptor)
- 4 send request message and wait for reply
- 5 `postinvoke` (interceptor), followed by `received_reply/receive_exception/receive_other` (Portable Interceptor) depending on the type of reply.

### Server side Interceptors

When both Portable and VisiBroker server side interceptors are installed, the order of events is received (locate requests do not fire interceptors, which is the same as VisiBroker behavior), assuming no interceptor throws an exception, is:

- 1 `received_request_service_contexts` (Portable Interceptor), followed by `preinvoke` (interceptor)

- 2 `servantLocator.preinvoke` (if using servant locator)
- 3 `receive_request` (Portable Interceptor)
- 4 invoke operation on servant
- 5 `postinvoke_premarshal` (interceptor)
- 6 `servantLocator.postinvoke` (if using servant locator)
- 7 `send_reply/send_exception/send_other`, depending on the outcome of the request
- 8 `postinvoke_postmarshal` (interceptor)

## Order of ORB events during POA creation

The order of ORB events during creation of a POA is listed as follows:

- 1 An IOR template is created based on profiles of server engines servicing the POA.
- 2 A interceptors' POA life cycle interceptors' `create()` method is invoked. This method can potentially add new policies or modify the IOR template created in the previous step.
- 3 A Portable Interceptor's IORInfo object is created and the IORInterceptors' `establish_components()` method is invoked. This interception point allows the interceptor to query the policies passed to `create_POA()` and those added in the previous step, and also add components to the IOR template based on those policies.
- 4 An object reference factory and object reference template for the POA are created, and the Portable Interceptor's IORInterceptors' `components_established()` method is invoked. This interception point allows the interceptor to change the POA's object reference factory, which will be used to manufacture object references.

## Order of ORB events during object reference creation

The following events occur during calls to POA that create object reference, e.g. `create_reference( )`, `create_reference_with_id( )`.

- 1 Call the object reference factory's `make_object( )` method to create the object reference (this does not call the VisiBroker IOR creation interceptors, and the factory may be user-supplied). If there are no VisiBroker IOR creation interceptors installed, this should be the object reference returned to the application; otherwise, proceed to step 2.
- 2 Extract the IOR from the delegate of the returned object reference, and call the VisiBroker IOR creation interceptors' `create( )` method.
- 3 IOR from step 2 is returned as the object reference to the caller of `create_reference( )`, `create_reference_with_id( )`.

# Using Object Wrappers

*This chapter describes the object wrapper feature of VisiBroker-RT for C++, which allows your applications to be notified or to trap an operation request for an object.*

## NOTE

The library libobjwrap.o is required on the VxWorks embedded node to support use of the VisiBroker Object Wrappers. For a description of all the libraries provided by the VisiBroker-RT for C++ product see [“Step 6: Integrating VisiBroker-RT with Tornado/VxWorks”](#).

## Overview

VisiBroker-RT for C++’s object wrapper feature allows you to define methods that are called when a client application invokes a method on a bound object or when a server application receives an operation request. Unlike the interceptor feature described in [“Using Portable Interceptors”](#) which is invoked at the ORB level, object wrappers are invoked before an operation request has been marshalled. In fact, you can design object wrappers to return results without the operation request having ever been marshalled, sent across the network, or actually presented to the object implementation.

Object wrappers may be installed on just the client-side, just the server-side, or they may be installed in both the client and server portions of a single application.

Here are a few examples of how you might use object wrappers in your application:

- Log information about the operation requests issued by a client or received by a server.
- Measure the time required for operation requests to complete.
- Cache the results of frequently issued operation requests so results can be immediately returned, without actually contacting the object implementation each time.

## Note

Externalizing a reference to an object for which object wrappers have been installed, using the ORB Object’s `object_to_string` method, will not propagate those wrappers to the recipient of the stringified reference if the recipient is a different process.

## Typed and un-typed object wrappers

VisiBroker-RT for C++ offers two kinds of object wrappers; typed and un-typed. You can mix the use of both of these types of wrappers within a single application. For information on typed wrappers, see [“Typed object wrappers”](#). [Table 25](#) summarizes the important distinctions between these two types of object wrappers.

**Table 25** Comparison of features for typed and un-typed object wrappers

Features	Typed	Un-typed
Receives all arguments that are to be passed to the stub	Yes	No
Can return control to the caller without actually invoking the next wrapper, the stub, or the object implementation.	Yes	No
Will be invoked for all operation requests for all objects.	No	Yes

## Special idl2cpp requirements

Whenever you plan to use typed or un-typed object wrappers, you must ensure that you use the `-obj_wrapper` option with the `idl2cpp` compiler when you generate the code for your applications. This will result in the generation of Object wrapper base class for each of your interfaces

## Example applications

The `<install_location>/VisiBrokerRT60/examples/interceptors/objectWrappers` directory contains three sample client and server applications that will be used to illustrate both the typed and untyped object wrapper concepts in this chapter.

## Un-typed object wrappers

Un-typed object wrappers allow you to define methods that are to be invoked before an operation request is processed, after an operation request is processed, or both. Un-typed wrappers can be installed for client or server applications and you can also install multiple versions.

You may also mix the use of both typed and un-typed object wrappers within the same client or server application.

By default, un-typed object wrappers have a global scope and will be invoked for any operation request. You can design un-typed wrappers so that they have no effect for operation requests on object types in which you are not interested.

### Note

Unlike typed object wrappers, un-typed wrapper methods do not receive the arguments that the stub or object implementation would receive nor can they prevent the invocation of the stub or object implementation.

[Figure 38](#) shows how an un-typed object wrapper's `pre_method` is invoked before the client stub method and how the `post_method` is invoked afterward. It also shows the calling sequence on the server-side with respect to the object implementation.

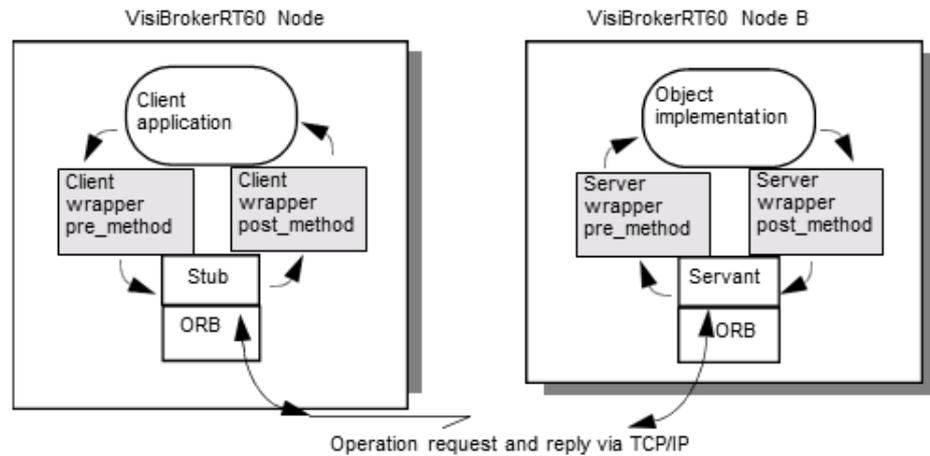


Figure 38 Single un-typed object wrapper

## Using multiple, un-typed object wrappers

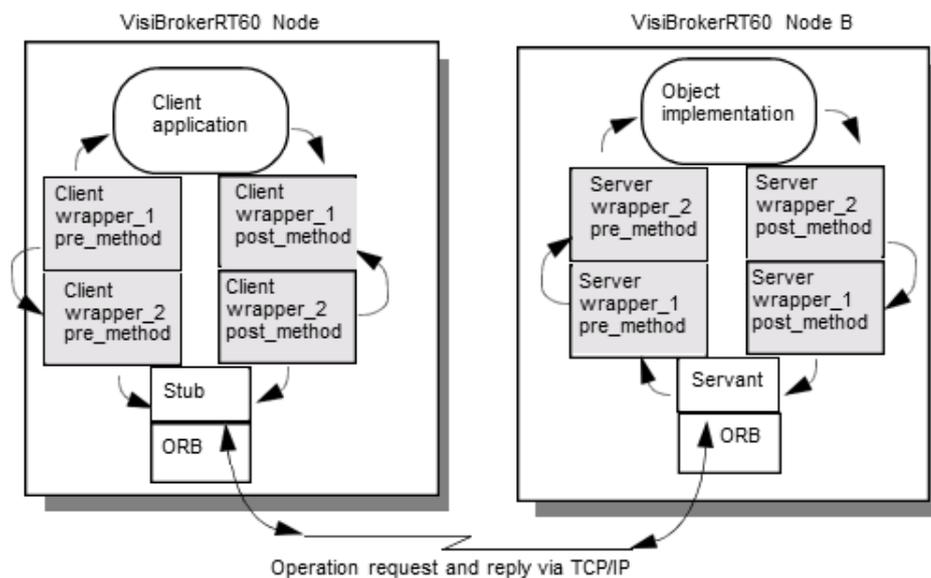


Figure 39 Multiple un-typed object wrappers

## Order of pre\_method invocation

When a client invokes a method on a bound object, each un-typed object wrapper `pre_method` will receive control before the client's stub routine is invoked. When a server receives an operation request, each un-typed object wrapper `pre_method` will be invoked before the object implementation receives control. In both cases, the first `pre_method` to receive control will be the one belonging to the object wrapper that was *registered first*.

## Order of `post_method` invocation

When a server's object implementation completes its processing, each `post_method` will be invoked before the reply is sent to the client. When a client receives a reply to an operation request, each `post_method` will be invoked before control is returned to the client. In both cases, the first `post_method` to receive control will be the one belonging to the object wrapper that was *registered last*.

### Note

If you choose to use both typed and un-typed object wrappers, see [“Command-line arguments for typed wrappers”](#) for information on the invocation order.

## Using un-typed object wrappers

You must use the following steps when using un-typed object wrappers. Each step is discussed, in turn.

- 1 Identify the interface, or interfaces, for which you want to create a un-typed object wrapper.
- 2 Generate the code from your IDL specification using the `idl2cpp` compiler using the `-obj_wrapper` option.
- 3 Create an implementation for your un-typed object wrapper factory, derived from the `VISObjectWrapper::UntypedObjectWrapperFactory` class.
- 4 Create an implementation for your un-typed object wrapper, derived from the `VISObjectWrapper::UntypedObjectWrapper` class.
- 5 Modify your application to create your un-typed object wrapper factory.

## Implementing an un-typed object wrapper factory

The `timeWrap.h` file, part of the `ObjectWrappers` sample applications, illustrates how to define an un-typed object wrapper factory that is derived from the `VISObjectWrapper::UntypedObjectWrapperFactory`. Your factory's `create` method will be invoked to create an un-typed object wrapper whenever a client binds to an object or a server invokes a method on an object implementation. The `create` method receives the target object, which allows you to design your factory to not create an un-typed object wrapper for those object types you wish to ignore. It also receives an enum specifying whether the object wrapper created is for the server-side object implementation or the client-side object.

The following code sample shows the `TimingObjectWrapperFactory`, which is used to create an un-typed object wrapper that displays timing information for method calls. Notice the addition of the `key` parameter to the `TimingObjectWrapperFactory` constructor. This parameter is also used by the service initializer to identify the wrapper.

**Code example 130** `TimingObjectWrapperFactory` implementation from the `TimeWrap.h` file

```
class TimingObjectWrapperFactory
    : public VISObjectWrapper::UntypedObjectWrapperFactory
{
public:
    TimingObjectWrapperFactory(VISObjectWrapper::Location loc,
```

```

        const char* key)
    : VISObjectWrapper::UntypedObjectWrapperFactory(loc),
      _key(key) {}

// ObjectWrapperFactory operations
VISObjectWrapper::UntypedObjectWrapper_ptr create(
    CORBA::Object_ptr target,
    VISObjectWrapper::Location loc) {
    if (_owrap == NULL) {
        _owrap = new TimingObjectWrapper(_key);
    }
}
return
VISObjectWrapper::UntypedObjectWrapper::_duplicate(_owrap);
}

private:
    CORBA::String_var _key;
    VISObjectWrapper::UntypedObjectWrapper_var _owrap;
};

```

## Implementing an un-typed object wrapper

The following code sample shows the implementation of the `TimingObjectWrapper`, also defined in the `timeWrap.h` file. Your un-typed wrapper must be derived from the

`VISObjectWrapper::UntypedObjectWrapper` class, and you may provide an implementation for both the `pre_method` or `post_method` methods in your un-typed object wrapper.

Once your factory has been installed, either automatically by the factory's constructor or manually by invoking the `VISObjectWrapper::ChainUntypedObjectWrapper::add` method, an un-typed object wrapper object will be created automatically whenever your client binds to an object or when your server invokes a method on an object implementation.

The `pre_method` shown in the following code sample, invokes the `TimerBegin` method, defined in `timeWrap.C`, which uses the `Closure` object to save the current time.

Similarly, the `post_method` invokes the `TimerDelta` method to determine how much time has elapsed since the `pre_method` was called and print the elapsed time.

### Code example 131 TimingObjectWrapper implementation

```

class TimingObjectWrapper : public
VISObjectWrapper::UntypedObjectWrapper {
public:
    TimingObjectWrapper(const char* key=NULL) : _key(key) {}

    void pre_method(const char* operation,
        CORBA::Object_ptr target,
        VISClosure& closure) {
        cout << "**Timing: [" << flush;
        if ((char *)_key)
            cout << _key << flush;
        else
            cout << "<no key>" << flush;
        cout << "] pre_method\t" << operation << "\t->" << endl;
        TimerBegin(closure, operation);
    }

    void post_method(const char* operation,
        CORBA::Object_ptr target,
        CORBA::Environment& env,
        VISClosure& closure) {
        cout << "**Timing: [" << flush;
        if ((char *)_key)

```

```

        cout << _key << flush;
    else
        cout << "<no key>" << flush; cout << "]" post_method\t" ;
        TimerDelta(closure, operation);
    }
private:
    CORBA::String_var _key;
};

```

## pre\_method and post\_method parameters

Both the `pre_method` and `post_method` receive these parameters:

**Table 26** Common arguments for the `pre_method` and `post_method` methods

Parameter	Description
<code>operation</code>	Name of the operation that was requested on the target object.
<code>target</code>	Target object.
<code>closure</code>	Area where data can be saved across method invocations for this wrapper.

The `post_method` also receives an `Environment` parameter, which can be used to inform the user of any exceptions that might have occurred during the previous steps of the method invocation.

## Creating and registering un-typed object wrapper factories

An un-typed object wrapper factory is automatically added to the chain of un-typed wrappers whenever it is created with the base class constructor that accepts a location.

On the client side, objects will be wrapped only if untyped object wrapper factories are created and registered before the objects are bound. On the server side, untyped object wrappers factories which are created and registered before an object implementation is called.

The following code sample shows a portion of the sample file `untypedClient.C`, which shows the creation, with automatic registration, of two un-typed object wrapper factories for a client.

The factories are created after the ORB has been initialized, but before the client binds to any objects.

**Code example 132** Creating and registering two client-side, un-typed object wrapper factories

```

void untyped_bank_client()
{
    VISTRY
    {
        // Install Untyped Object Wrappers for Account.
        TimingObjectWrapperFactory timingfact(VISObjectWrapper::Client,
            "timeclient");
        TraceObjectWrapperFactory tracingfact(VISObjectWrapper::Client,
            "traceclient");
        ...
    }
}

```

The following code sample shows the sample file `untypedServer.C`, which shows the creation and registration of un-typed object wrapper factories for a server. The factories are created after the ORB is initialized, but before any object implementations are created.

**Code example 133** Registering a server-side, un-typed object wrapper factory

```
// UntypedServer.C
#include <vxWorks.h>
#include "corba.h"
#include "timeWrap.h"
#include "traceWrap.h"
#include "bankImpl.h"
#include "bank_s.hh"

...

void untyped_bank_server()
{
    PortableServer::POA_var rootPOA;

    VISTRY
    {
        // get a reference to the root POA
        CORBA::Object_var obj =
            orb->resolve_initial_references("RootPOA");
        VISIFNOT_EXCEP
        rootPOA = PortableServer::POA::_narrow(obj);
        VISEND_IFNOT_EXCEP

        CORBA::PolicyList policies;
        policies.length(1);
        VISIFNOT_EXCEP
        policies[(CORBA::ULong)0] =

rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);
VISEND_IFNOT_EXCEP

// get the POA Manager
PortableServer::POAManager_var poa_manager;

VISIFNOT_EXCEP
    poa_manager = rootPOA->the_POAManager();
VISEND_IFNOT_EXCEP

// Create myPOA with the right policies
PortableServer::POA_var myPOA;

VISIFNOT_EXCEP
    myPOA = rootPOA->create_POA("bank_ow_poa", poa_manager,
policies);
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
    // Install Untyped Object Wrappers for Account Manager.
    TimingObjectWrapperFactory* timingfact =
new TimingObjectWrapperFactory(VISObjectWrapper::Server,
"timingserver");
    TraceObjectWrapperFactory* tracingfact =
new TraceObjectWrapperFactory(VISObjectWrapper::Server,
"traceserver");

// Create the servant
AccountManagerImpl *managerServant = new AccountManagerImpl;

// Create the object ID PortableServer::ObjectId_var
managerId;

VISIFNOT_EXCEP
    managerId =
PortableServer::string_to_ObjectId("BankManager");
VISEND_IFNOT_EXCEP

// Activate the servant with the ID on myPOA
VISIFNOT_EXCEP
    myPOA->activate_object_with_id(managerId, managerServant);
VISEND_IFNOT_EXCEP
}
```

```

// Activate the POA Manager
VISIFNOT_EXCEP
    poa_manager->activate();
VISEND_IFNOT_EXCEP

CORBA::Object var reference;
VISIFNOT_EXCEP
    reference = myPOA->servant_to_reference(managerServant);
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
    cout << reference << " is ready" << endl;
VISEND_IFNOT_EXCEP

}
VISCATCH(CORBA::Exception, e)
{
    cerr << e << endl;
    taskSuspend(0);
}
VISEND_CATCH

return;
}

```

## Removing un-typed object wrappers

The `VISObjectWrapper::ChainUntypedObjectWrapperFactory` class' `remove` method can be used to remove an un-typed object wrapper factory from a client or server application. You must specify a location when removing a factory. This means that if you have added a factory with a location of `VISObjectWrapper::Both`, you can selectively remove it from the `Client` location, the `Server` location, or `Both`.

### Note

Removing one or more object wrapper factories from a client will not affect objects of that class that are already bound by the client. Only subsequently bound objects will be affected. Removing object wrapper factories from a server will not affect object implementations that have already been created. Only subsequently created object implementations will be affected.

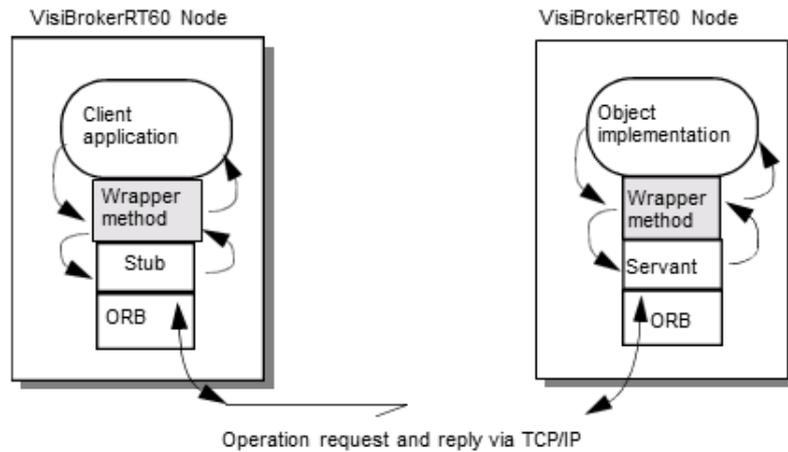
## Typed object wrappers

When you implement a typed object wrapper for a particular class, you define the processing that is to take place when a method is invoked on a bound object. [Figure 40](#) shows how an object wrapper method on the client is invoked before the client stub class method and how an object wrapper on the server-side is invoked before the server's implementation method.

### Note

Your typed object wrapper implementation is not required to implement all methods offered by the object it is wrapping.

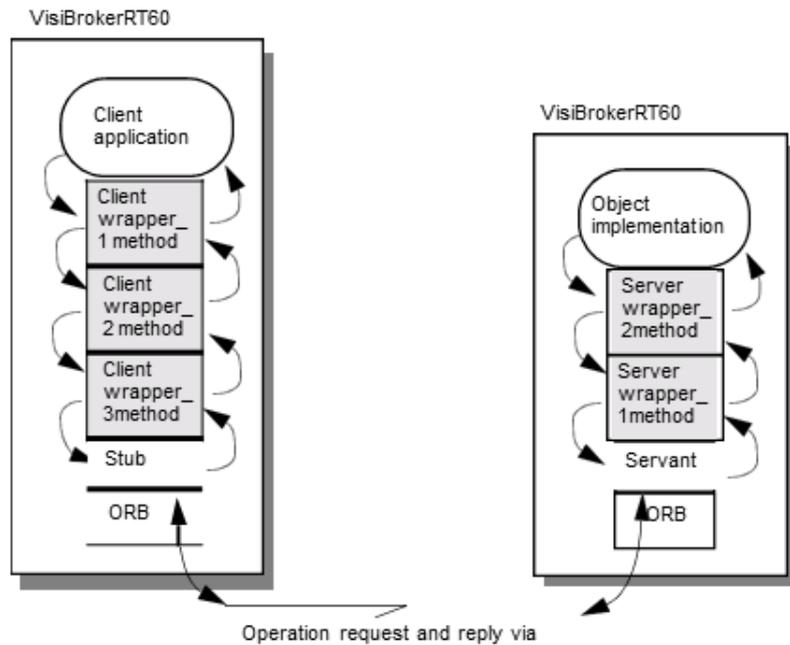
You may also mix the use of both typed and un-typed object wrappers within the same client or server application. For more information, see ["Combined use of un-typed and typed object wrappers"](#).



**Figure 40** Single typed object wrapper registered

## Using multiple, typed object wrappers

You may implement and register more than one typed object wrapper for a particular class of object, as shown in [Figure 41](#). On the client side, the first object wrapper registered is `client_wrapper_1`, so its methods will be the first to receive control. After performing its processing, the `client_wrapper_1` method may pass control to the next object's method in the chain or it may return control to the client. On the server side, the first object wrapper registered is `server_wrapper_1`, so its methods will be the first to receive control. After performing its processing, the `server_wrapper_1` method may pass control to the next object's method in the chain or it may return control to the servant.



**Figure 41** Multiple, typed object wrappers registered

## Order of invocation

The methods for a typed object wrapper that are register for a particular class will receive all of the arguments that are normally passed to the stub method on the client side or to skeleton on the server side. Each object wrapper method can pass control to the next wrapper method in the chain by invoking the parent class' method,

`<interface_name>ObjectWrapper::<method_name>`. If an object wrapper wishes to return control without calling the next wrapper method in the chain, it can `return` with the appropriate return value.

A typed object wrapper method's ability to return control to the previous method in the chain allows you to create a wrapper method that never invokes a client stub or object implementation. For example, you can create an object wrapper method that caches the results of a frequently requested operation. In this scenario, the first invocation of a method on the bound object results in an operation request being sent to the object implementation. As control flows back through the object wrapper method, the result is stored. On subsequent invocations of the same method, the object wrapper method can simply return the cached result without actually issuing the operation request to the object implementation.

If you choose to use both typed and un-typed object wrappers, see ["Combined use of un-typed and typed object wrappers"](#) for information on the invocation order.

## Typed object wrappers with co-located client and servers

When the client and server are both packaged in the same address space, the first object wrapper method to receive control will belong to the first client-side object wrapper that was installed. [Figure 42](#) illustrates the invocation order.

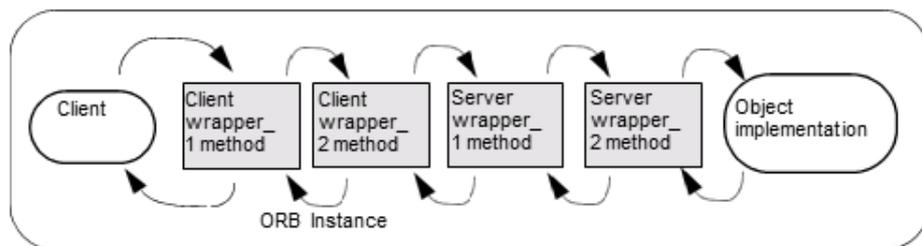


Figure 42 Typed object wrapper invocation order

## Using typed object wrappers

You must use the following steps when using typed object wrappers. Each step is discussed in turn.

- 1 Identify the interface, or interfaces, for which you want to create a typed object wrapper.
- 2 Generate the code from your IDL specification using the `idl2cpp` compiler using the `-obj_wrapper` option.
- 3 Derive your typed object wrapper class from the `<interface_name>ObjectWrapper` class generated by the `idl2cpp`

compiler and provide an implementation of those methods you wish to wrap.

- 4 Modify your application to register the typed object wrapper.

## Implementing typed object wrappers

You derive typed object wrappers from the

`<interface_name>ObjectWrapper` class that is generated by the `idl2cpp` compiler. The following code shows the implementation of a typed object wrapper for the `Account` interface from the file `bankWrap.h`. Notice that this class is derived from the `AccountObjectWrapper` interface and provides a simple caching implementation of the `balance` method, which provides these processing steps:

- 1 Check the `_inited` flag to see if this method has been invoked before.
- 2 If this is the first invocation, the `balance` method on the next object in the chain is invoked and the result is saved to `_balance`, `_inited` is set to `true`, and the value is returned.
- 3 If this method has been invoked before, simply return the cached value.

**Code example 134** Portion of the `CachingAccountObjectWrapper` implementation

```
class CachingAccountObjectWrapper : public
Bank::AccountObjectWrapper {
public:
    CachingAccountObjectWrapper(): _inited((CORBA::Boolean)0) {}
    CORBA::Float balance() {
        cout << "+ CachingAccountObjectWrapper: Before Calling
Balance" << endl;
        if (! _inited) {
            _balance = Bank::AccountObjectWrapper::balance();
            _inited = 1;
        } else {
            cout << "+ CachingAccountObjectWrapper: Returning Cached
Value" << endl;
        }
        cout << "+ CachingAccountObjectWrapper: After Calling
Balance" << endl;
        return _balance;
    }
    ...
};
```

## Registering typed object wrappers for a client

A typed object wrapper is registered on the client-side by invoking the `<interface_name>::add` method that is generated for the class by the `idl2cpp` compiler. Client-side object wrappers must be registered after the `ORB_init` method has been called, but before any objects are bound. The following code shows a portion of the `typedClient.C` file that creates and registers a typed object wrapper.

**Code example 135** Creating and registering a client-side, typed object wrapper

```
...
void typed_client(void)
{
    VISTRY{
        // Install Typed Object Wrappers for Account.

        Bank::AccountObjectWrapper::add(orb,
            CachingAccountObjectWrapper::factory,
            VISObjectWrapper::Client);

        // Get the Manager ID.
```

```

PortableServer::ObjectId_var managerId =
PortableServer::string_to_ObjectId("BankManager");

// Locate an Account Manager.
Bank::AccountManager_var manager =
    Bank::AccountManager::_bind("/bank_ow_poa", managerId);
...

```

The ORB keeps track of any object wrappers that have been registered for it on the client-side. When a client invokes the `_bind` method to bind to an object of that type, the necessary object wrappers will be created. If a client binds to more than one instance of a particular class of object, each instance will have its own set of wrappers.

## Registering typed object wrappers for a server

As with a client application, a typed object wrapper is registered on the server-side by invoking the `<interface_name>::add` method. Server-side, typed object wrappers must be registered after the `ORB_init` method has been called, but before an object implementation services a request. The following code shows a portion of the `typedServer.C` file that installs a typed object wrapper.

### Code example 136 Registering a server-side, typed object wrapper

```

// TypedServer.C
#include <vxWorks.h>
#include "corba.h"
#include "bank_s.hh"
#include "bankImpl.h"
#include "bankWrap.h"

...

void typed_bank_server()
{
    PortableServer::POA_var rootPOA;

    VISTRY
    {
        // get a reference to the root POA
        CORBA::Object_var obj =
            orb->resolve_initial_references("RootPOA");

        VISIFNOT_EXCEP
            rootPOA = PortableServer::POA::_narrow(obj);
        VISEND_IFNOT_EXCEP

        CORBA::PolicyList policies;
        policies.length(1);
        VISIFNOT_EXCEP
            policies[(CORBA::ULong)0] =
                rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);
        VISEND_IFNOT_EXCEP

        // get the POA Manager
        PortableServer::POAManager_var poa_manager;

        VISIFNOT_EXCEP
            poa_manager = rootPOA->the_POAManager();
        VISEND_IFNOT_EXCEP

        // Create myPOA with the right policies
        PortableServer::POA_var myPOA;

        VISIFNOT_EXCEP
            myPOA = rootPOA->create_POA("bank_ow_poa", poa_manager,
                policies);
        VISEND_IFNOT_EXCEP
    }
}

```

```

VISIFNOT_EXCEP
// Install Typed Object Wrappers for Account Manager.
Bank::AccountManagerObjectWrapper::add(orb,
    SecureAccountManagerObjectWrapper::factory,
    VISObjectWrapper::Server);
Bank::AccountManagerObjectWrapper::add(orb,
    CachingAccountManagerObjectWrapper::factory,
    VISObjectWrapper::Server);

// Create the servant
AccountManagerImpl *managerServant = new AccountManagerImpl;
// Create the object ID
PortableServer::ObjectId_var managerId;

VISIFNOT_EXCEP
managerId = PortableServer::string_to_ObjectId("BankManager");
VISEND_IFNOT_EXCEP

// Activate the servant with the ID on myPOA
VISIFNOT_EXCEP
myPOA->activate_object_with_id(managerId, managerServant);
VISEND_IFNOT_EXCEP

// Activate the POA Manager VISIFNOT_EXCEP
poa manager->activate();
VISEND_IFNOT_EXCEP

CORBA::Object_var reference;
VISIFNOT_EXCEP
reference = myPOA->servant_to_reference(managerServant);
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
cout << reference << " is ready" << endl;
VISEND_IFNOT_EXCEP

}
VISCATCH(CORBA::Exception, e)
{
    cerr << e << endl;
    taskSuspend(0);
}
VISEND_CATCH
return;
}

```

If a server creates more than one instance of a particular class of object, a set of wrappers will be created for each instance.

## Removing typed object wrappers

The `<interface_name>ObjectWrapper::remove` method that is generated for a class by the `idl2cpp` compiler allows you to remove a typed object wrapper from a client or server application. You must specify a location when removing a factory. This means that if you have added a factory with a location of `VISObjectWrapper::Both`, you can selectively remove it from the Client location, the Server location, or Both.

### Note

Removing one or more object wrappers from a client will not affect objects of that class that are already bound by the client. Only subsequently bound objects will be affected. Removing object wrappers from a server will not affect object implementations that have already serviced requests. Only subsequently created object implementations will be affected.

## Combined use of un-typed and typed object wrappers

If you choose to use both typed and un-typed object wrappers in your application, all `pre_method` methods defined for the un-typed wrappers will be invoked prior to any typed object wrapper methods defined for an object. Upon return, all typed object wrapper methods defined for the object will be invoked prior to any `post_method` methods defined for the un-typed wrappers.

The sample applications `Client.C` and `Server.C` make use of a sophisticated design that allows you to use command-line properties to specify which, if any, typed and un-typed object wrappers are to be used.

## Command-line arguments for typed wrappers

The table below shows the command-line arguments you can use to enable the use of typed object wrappers for the sample bank applications implemented in `typedClient.C` and `typedServer.C`.

**Table 27** Command-line arguments for the controlling typed object wrappers

Bank wrappers properties	Description
<code>-BANKaccountCacheCnt</code> <0 1>	Enables or disables a typed object wrapper that caches the results of the <code>balance</code> method for a client application.
<code>-BANKaccountCacheSrvr</code> <0 1>	Enables or disables a typed object wrapper that caches the results of the <code>balance</code> method for a server application.
<code>-BANKmanagerCacheCnt</code> <0 1>	Enables or disables a typed object wrapper that caches the results of the <code>open</code> method for a client application.
<code>-BANKmanagerCacheSrvr</code> <0 1>	Enables or disables a typed object wrapper that caches the results of the <code>open</code> method for a server application.
<code>-BANKmanagerSecurityCnt</code> <0 1>	Enables or disables a typed object wrapper that detects unauthorized users passed on the <code>open</code> method for a client application.
<code>-BANKmanagerSecuritySrvr</code> <0 1>	Enables or disables a typed object wrapper that detects unauthorized users passed on the <code>open</code> method for a server application.

## Initializer for typed wrappers

The typed wrappers are created in the `BankInit::update` initializer, defined in `<install_location>/VisiBrokerRT60/examples/interceptors/objectWrappers/bankWrap.C`. This initializer will be invoked when the `ORB_init()` method is invoked and will handle the installation of various typed object wrappers, based on the command-line properties you specify.

The code sample below shows how the initializer uses a set of `PropStruct` objects to track the command-line options that have been specified and then add or remove `AccountObjectWrapper` objects for the appropriate locations.

**Code example 137** Initializer for a typed object wrapper

```
...
static const CORBA::ULong kNumTypedAccountProps = 2;
static PropStruct TypedAccountProps[kNumTypedAccountProps] =
```

```

{ { "BANKaccountCacheClnt", CachingAccountObjectWrapper::factory,
  VISObjectWrapper::Client },
{ "BANKaccountCacheSrvr", CachingAccountObjectWrapper::factory,
  VISObjectWrapper::Server }
};
static const CORBA::ULong kNumTypedAccountManagerProps = 4;
static PropStruct
TypedAccountManagerProps[kNumTypedAccountManagerProps] =
{ { "BANKmanagerCacheClnt",
  CachingAccountManagerObjectWrapper::factory,
  VISObjectWrapper::Client },
{ "BANKmanagerSecurityClnt",
  SecureAccountManagerObjectWrapper::factory,
  VISObjectWrapper::Client },
{ "BANKmanagerCacheSrvr",
  CachingAccountManagerObjectWrapper::factory,
  VISObjectWrapper::Server },
{ "BANKmanagerSecuritySrvr",
  SecureAccountManagerObjectWrapper::factory,
  VISObjectWrapper::Server }
};

void BankInit::update(int& argc, char* const* argv) {
  if (argc > 0) {
    init(argc, argv, "-BANK");
    CORBA::ULong i;
    for (i=0; i < kNumTypedAccountProps; i++) {
      CORBA::String_var
      arg(getArgValue(TypedAccountProps[i].propname));
      if (arg && strlen(arg) > 0) {
        if (atoi((char*) arg)) {
          Bank::AccountObjectWrapper::add(_orb,
            TypedAccountProps[i].fact,
            TypedAccountProps[i].loc);
        } else {
          Bank::AccountObjectWrapper::remove(_orb,
            TypedAccountProps[i].fact,
            TypedAccountProps[i].loc);
        }
      }
    }
    for (i=0; i < kNumTypedAccountManagerProps; i++) {
      CORBA::String_var arg(
        getArgValue(TypedAccountManagerProps[i].propname));
      if (arg && strlen(arg) > 0) {
        if (atoi((char*) arg)) {
          Bank::AccountManagerObjectWrapper::add(_orb,
            TypedAccountManagerProps[i].fact,
            TypedAccountManagerProps[i].loc);
        } else {
          Bank::AccountManagerObjectWrapper::remove(_orb,
            TypedAccountManagerProps[i].fact,
            TypedAccountManagerProps[i].loc);
        }
      }
    }
  }
}

```

## Command-line arguments for un-typed wrappers

[Table 28](#) shows the command-line arguments you can use to enable the use of un-typed object wrappers for the sample bank applications implemented in `untypedClient.C` and `untypedServer.C`.

**Table 28** Command-line arguments for controlling un-typed object wrappers

Bank wrappers properties	Description
-TRACEWRAPclient <numwraps>	Instantiates the specified number of un-typed object wrapper factories for tracing wrappers for a client application.
-TRACEWRAPserver <numwraps>	Instantiate the specified number of un-typed object wrapper factories for tracing on a server application.
-TRACEWRAPboth <numwraps>	Instantiate the specified number of un-typed object wrapper factories for tracing for both a client and server application.
-TIMINGWRAPclient <numwraps>	Instantiate the specified number of un-typed object wrapper factories for timing on a client application.
-TIMINGWRAPserver <numwraps>	Instantiate the specified number of un-typed object wrapper factories for timing on a server application.
-TIMINGWRAPboth <numwraps>	Instantiate the specified number of un-typed object wrapper factories for timing on both a client and a server application.

## Initializers for un-typed wrappers

The un-typed wrappers are created and registered in the `TraceWrapInit::update` and `TimingWrapInit::update` methods, defined in `traceWrap.C` and `timeWrap.C`. These initializers will be invoked when the `ORB_init` method is invoked and will handle the installation of various un-typed object wrappers.

The code sample below shows a portion of the `traceWrap.C` file, which will install the appropriate un-typed object wrapper factories, based on the command-line properties you specify.

### Code example 138

```

TraceWrapInit::update(int& argc, char* const* argv) {
    if (argc > 0) {
        init(argc, argv, "-TRACEWRAP");

        VISObjectWrapper::Location loc;
        const char* propname;
        LIST(VISObjectWrapper::UntypedObjectWrapperFactory_ptr)
        *list;

        for (CORBA::ULong i=0; i < 3; i++) {
            switch (i) {
                case 0:
                    loc = VISObjectWrapper::Client;
                    propname = "TRACEWRAPclient";
                    list = &_clientfacts;
                    break;
                case 1:
                    loc = VISObjectWrapper::Server;
                    propname = "TRACEWRAPserver";
                    list = &_serverfacts;
                    break;
                case 2:
                    loc = VISObjectWrapper::Both;
                    propname = "TRACEWRAPboth";
                    list = &_bothfacts;
                    break;
            }
        }
        CORBA::String_var getArgValue(property_value(propname));
        if (arg && strlen(arg) > 0) {
            int numNew = atoi((char*) arg);
            char key_buf[256];
            for (CORBA::ULong j=0; j < numNew; j++) {
                sprintf(key_buf, "%s-%d", propname, list->size());
            }
        }
    }
}

```

```

        list->add(new TraceObjectWrapperFactory(loc,
        (const char*) key_buf));
    }
}
}
}
}

```

## Executing the sample applications

Before executing the sample applications, make sure that an osagent is running on your network. You can then execute the server application without any tracing or timing object wrappers from the Tornado WindShell by:

### Example

```

--> ld < corba_init
-->start_corba
--> ld < server
--> ld < client
-->start_objwrap_server

```

The Client can then be started from the same Tornado WindShell

### Example

```

-->start_objwrap_client("John")

```

You can also execute this command if you want a default name to be used.

### Example

```

-->start_objwrap_client()

```

## Turning on timing and tracing object wrappers

To execute the client with un-typed timing and tracing object wrappers enabled, use this command:

### Example

```

-->start_objwrap_client("-TRACEWRAPclient 1 -TIMINGWRAPclient 1")

```

To execute the server with un-typed wrappers for timing and tracing enabled, use this command:

### Example

```

-->start_objwrap_server("-TRACEWRAPserver 1 -TIMINGWRAPserver 1")

```

## Turning on caching and security object wrappers

To execute the client with the typed wrappers for caching and security enabled, use this command:

### Example

```

-->start_objwrap_client(" -BANKaccountCacheClnt 1
_BANKmanagerCacheClnt 1 \
-BANKmanagerSecurityClnt 1")

```

To execute the server with typed wrappers for caching and security enabled, use this command:

### Example

```
-->start_objwrap_server("-BANKaccountCacheSrvr 1
-BANKmanagerCacheSrvr 1 \
-BANKmanagerSecuritySrvr 1")
```

## Turning on typed and un-typed wrappers

To execute the client with all typed and un-typed wrappers enabled, use this command:

### Example

```
-->start_objwrap_client("-BANKaccountCacheClnt 1
-BANKmanagerCacheClnt 1 \
-BANKmanagerSecurityClnt 1-TRACEWRAPclient 1
-TIMINGWRAPclient 1")
```

To execute the server with all typed and un-typed wrappers enabled, use this command:

### Example

```
-->start_objwrap_server(" BANKaccountCacheSrvr 1
BANKmanagerCacheSrvr 1 \
-BANKmanagerSecuritySrvr 1 \ -TRACEWRAPserver 1
-TIMINGWRAPserver 1")
```

## Executing a co-located client and server

The following command will execute a co-located server and client with all typed wrappers enabled, the un-typed wrapper enables for just the client, and the un-typed tracing wrapper for just the server, use this command:

### Example

```
-->start_objwrap_server("-BANKaccountCacheClnt 1
-BANKaccountCacheSrvr 1 \
-BANKmanagerCacheClnt 1
-BANKmanagerCacheSrvr 1 \
-BANKmanagerSecurityClnt 1 \
-BANKmanagerSecuritySrvr 1 \
-TRACEWRAPboth 1 \
-TIMINGWRAPboth 1")
```

# Using Valuetypes

*This chapter explains how to use the `valuetype` IDL type in VisiBroker-RT for C++.*

## Understanding valuetypes

The IDL type `valuetype` is used to pass state data over the wire. A `valuetype` is best thought of as a `struct` with inheritance and methods. Valuetypes differ from normal interfaces in that they contain properties to describe the valuetype's state, and contain implementation details beyond that of an interface. The following IDL code declares a simple valuetype:

**IDL sample 20** Simple valuetype IDL

```
module Map {
    valuetype Point {
        public long x;
        public long y;
        private string label;
        factory create (in long x, in long y, in string z);
        void print();
    };
};
```

Valuetypes are always local. They are not registered with the ORB, and require no identity, as their value is their identity. They can not be called remotely.

## Concrete valuetypes

Concrete valuetypes contain state data. They extend the expressive power of IDL structs by allowing:

- Single concrete valuetype derivation and multiple abstract valuetype derivation
- Multiple interface support (one concrete and multiple abstract)
- Arbitrary recursive valuetype definitions
- Null value semantics
- Sharing semantics

## Valuetype derivation

You can derive a concrete valuetype from a single concrete valuetype. However, valuetypes can be derived from multiple other abstract valuetypes.

## Sharing semantics

Valuetype instances can be shared by other valuetypes across or within other instances. Other IDL data types such as structs, unions, or sequences can not be shared. Valuetypes that are shared are isomorphic between the sending context and the receiving context.

In addition, when the same valuetype is passed into an operation for two or more arguments, the receiving context receives the same valuetype reference for both arguments.

## Null semantics

Null valuetypes can be passed over the wire, unlike IDL data types such as structs, unions, and sequences. For instance by boxing a struct as a boxed valuetype, you can pass a null value struct. For more information, see [“Boxed valuetypes”](#).

## Factories

Factories are methods that can be declared in valuetypes to create valuetypes in a portable way. For more information on Factories, see [“Implementing factories”](#).

## Abstract valuetypes

Abstract valuetypes contain only methods and do not have state. They may not be instantiated. Abstract valuetypes are a bundle of operation signatures with a purely local implementation.

For instance, the following IDL defines an abstract valuetype `Account` that contains no state, but one method, `get_name()`:

```
abstract valuetype Account{
    string get_name();
}
```

Now, two valuetypes are defined that inherit the `get_name()` method from the abstract valuetype:

```
valuetype savingsAccount:Account{
    private long balance;
}
valuetype checkingAccount:Account{
    private long balance;
}
```

These two valuetypes contain a variable `balance`, and they inherit the `get_name()` method from the abstract valuetype `Account`.

# Implementing valuetypes

To implement valuetypes in an application, do the following:

- 1 Define the valuetypes in an IDL file.
- 2 Compile the IDL file using `idl2cpp`.
- 3 Implement your valuetypes by inheriting the valuetype base class.
- 4 Implement the Factory class to implement any factory methods defined in IDL
- 5 Implement the `create_for_unmarshal` method.
- 6 Register your Factory with the ORB.
- 7 Either implement the `_add_ref`, `_remove_ref`, and `_ref_countvalue` methods or derive from `CORBA::DefaultValueRefCountBase`.

## Defining your valuetypes

In [IDL sample 20](#), you define a valuetype named `Point` that defines a point on a graph. It contains two public variables, the `x` and `y` coordinates, one private variable that is the label of the point, the valuetype’s factory, and a print method to print the point.

## Compiling your IDL file

Now that you've defined your IDL, compile it using `idl2cpp`. This will create the C++ source files that you will use to implement your valuetypes.

If you compile the above IDL, your output will consist of the following files:

- `map_c.cc`
- `map_c.hh`
- `map_s.cc`
- `map_s.hh`

## Inheriting the valuetype base class

After compiling your IDL, create your implementation of the valuetype. The implementation class will inherit the base class. This class contains the constructor that is called in your `ValueFactory`, and contains all the variables and methods declared in your IDL.

For example, in `<install_location>/VisiBrokerRT60/examples/obv/point/ pntImpl.h`, the `PointImpl` class extends the `Point` class which was generated from the IDL:

```
class PointImpl : public Map::OBV_Point, public
CORBA::DefaultValueRefCountBase {
public:
    PointImpl() {}
    virtual ~PointImpl() {}

    CORBA_ValueBase* _copy_value() {
        return new PointImpl(x(), y(), new
            Map::Label(CORBA::string_dup(label())));
    }
    PointImpl( CORBA::Long x, CORBA::Long y, Map::Label_ptr label
    )
        : OBV_Point( x,y,label->_boxed_in() )
    {}
    virtual void print() {
        cout << "Point is [" << label() << ": ("
            << x() << ", " << y() << ")]" << endl << endl;
    }
};
```

## Implementing the Factory class

Now that you have created an implementation class, implement the Factory for your valuetype.

In our example, the generated `Point_init` class contains the `create` method declared in your IDL. This class extends `CORBA::ValueFactoryBase`. The `PointDefaultFactory` class implements `PointValueFactory`:

```
class PointFactory: public CORBA::ValueFactoryBase {
public:
    PointFactory() {}
    virtual ~PointFactory() {}
    CORBA::ValueBase* create_for_unmarshal() {
        return new PointImpl();
    }
};
```

`Point_init` contains a public method, `create_for_unmarshal`, that is output as a pure virtual method in `Map_c.hh`. You must derive a class from `Point_init` and implement the `create_for_unmarshal` method to produce

the Factory class. When you compile your IDL file, it won't create a skeleton class for this.

## Registering your Factory with the ORB

Call `ORB::register_value_factory` to register your Factory with the ORB. See [“Registering valuetypes”](#) for more information on registering Factories.

## Implementing factories

When the ORB receives a valuetype, it must first be unmarshalled, and then the appropriate factory for that type must be found in order to create a new instance of that type. Once the instance has been created, the value data is unmarshalled into the instance. The type is identified by the RepositoryID that is passed as part of the invocation. The mapping between the type and the factory is language specific.

The following code contains a sample implementation of the factory of the Point valuetype:

**Code example 139** Factory for Point valuetype

```
class PointFactory: public CORBA::ValueFactoryBase
{
public:
    PointFactory() {}
    virtual ~PointFactory() {}
    CORBA::ValueBase*create_for_unmarshal() {
        return new PointImpl();
    }
};
```

## Factories and valuetypes

When the ORB receives a valuetype, it will look for that type's factory. It will look for a factory named `valuetypeDefaultFactory`. For instance, the Point valuetype's factory is called `PointDefaultFactory`. If the correct factory doesn't conform to this naming schema (`valuetypeDefaultFactory`), you must register the correct factory so the ORB can create an instance of the valuetype.

If the ORB cannot find the correct factory for a given valuetype, a MARSHAL exception is raised.

## Registering valuetypes

Each language mapping specifies how and when registration occurs. If you created a factory with the `valuetypeDefaultFactory` naming convention, this is considered implicitly registering that factory, and you do not need to explicitly register your factory with the ORB.

To register a factory that doesn't conform to the `valuetypeDefaultFactory` naming convention, call `register_value_factory`. To unregister a factory, call `unregister_value_factory` on the ORB. You can also lookup a registered valuetype factory by calling `lookup_value_factory` on the ORB.

## Boxed valuetypes

Boxed valuetypes allow you to wrap non-value IDL data types as valuetypes. For example, the following IDL boxed valuetype declaration,

```
valuetype Label string;
```

is equivalent to this IDL valuetype declaration:

```
valuetype Label{
    public string name;
}
```

By boxing other data types as valuetypes, it allows you to use valuetype's null semantics and sharing semantics.

Valueboxes are implemented purely with generated code. No user code is required.

## Abstract interfaces

Abstract interfaces allow you to choose at runtime whether the object will be passed by value or by reference.

They differ from IDL interfaces in the following ways:

- The actual parameter type determines whether the object is passed by reference or a valuetype is passed. The parameter type is determined based on two rules:
  - It is treated as an object reference if it is a regular interface type or sub-type, the interface type is a sub-type of the signature abstract interface type, and the object is already registered with the ORB.
  - It is treated as a value if it can not be passed as an object reference, but can be passed as a value. If it fails to pass as a value, a `BAD_PARAM` exception is raised.
- Abstract interfaces do not implicitly derive from `CORBA::Object` because they can represent either object references or valuetypes. Valuetypes do not necessarily support common object reference operations. If the abstract interface can be successfully narrowed to an object reference type, you can invoke the operations of `CORBA::Object`.
- Abstract interfaces may only inherit from other abstract interfaces.
- valuetypes can support one or more abstract interfaces. For example, examine the following abstract interface.

### IDL sample 21 Abstract interface IDL

```
abstract interface ai{
};
interface itp : ai{
};
valuetype vtp supports ai{
};
interface x {
    void m(ai aitp);
};
valuetype y {
    void op(ai aitp);
};
```

For the argument to method `m`:

- `itp` is always passed as an object reference.
- `vtp` is passed as a value.

## Custom valuetypes

By declaring a custom valuetype in IDL, you bypass the default marshalling and unmarshalling model and are responsible for encoding and decoding.

### IDL sample 22 Custom valuetype IDL

```
custom valuetype customPoint{
    public long x;
    public long y;
    private string label;
    factory create(in long x, in long y, in string z);
};
```

You must implement the marshal and unmarshal methods from the `CustomMarshal` interface.

When you declare a custom valuetype, the valuetype extends `CORBA::CustomValue`, as opposed to `CORBA::StreamableValue`, as in a regular valuetype. The compiler doesn't generate read or write methods for your valuetype.

You must implement your own read and write methods by using `CORBA::DataInputStream` and `CORBA::DataOutputStream` to read and write the values, respectively. For more information on these classes, see the *VisiBroker-RT for C++ Reference Guide*.

## Truncatable valuetypes

Truncatable valuetypes allow you to treat an inherited valuetype as its parent.

The following IDL defines a valuetype `checkingAccount` that is inherited from the base type `Account` and can be truncated on the receiving object.

```
valuetype checkingAccount: truncatable Account{
    private long balance;
}
```

This is useful if the receiving context doesn't need the new data members or methods in the derived valuetype, and if the receiving context isn't aware of the derived valuetype. However, any state data from the derived valuetype that isn't in the parent data type will be lost when the valuetype is passed to the receiving context.

### Note

You cannot make a custom valuetype truncatable.

# VisiBroker Logging

*VisiBroker-RT for C++ provides a logging mechanism which allows applications to log messages and have them directed, via configurable logging forwarders, to an appropriate destination or destinations. The ORB itself uses this mechanism for the output of any error, warning or informational messages.*

*The application can choose to log its and the ORB's messages to the same destination, producing a single message log for the entire system, or to log messages from different sources to independent destinations.*

## Logging Overview

VisiBroker Logging employs one or more Logger objects, that applications (including the ORB) may log messages to. When a message is logged to a Logger, it is queued rather than being output by the calling thread.

Each Logger has one or more Forwarders associated with it: application-definable pieces of code that read the queued messages and forward them to desired destinations such as standard error, a file or over a network. All the Forwarders associated with a given Logger run on a single Forwarder Thread. The priority of the Forwarder Thread is configurable.

However, forwarding is not enabled when a Logger is created. Messages logged before forwarding is enabled are queued until it is enabled. This allows messages to be logged before the Logger or all of the output destinations have been fully configured (for example during static initialization of C++ constructors.)

The ORB uses a special Logger instance (the 'Default Logger'), which is created automatically the first time the ORB logs a message to it. Applications can log messages to the Default Logger as well, to integrate their logging output with that of the ORB, or they can create one or more other Loggers, to log messages independently. The 'standard error' iostream is the default destination for messages logged to the Default Logger.

## The Logger Manager

The Logger Manager is used to manage the lifecycle of Loggers and to configure them. The Logger Manager is a singleton object belonging to the ORB. A reference to it is obtained by calling its static instance method. No reference counting is performed upon the Logger Manager.

**Code example 140** Using the static instance method to access the singleton Logger Manager object

```
// Use its static instance method to obtain a reference to the
// Logger Manager
VISLoggerManager_ptr logger_manager =
    VISLoggerManager::instance();

...

// Alternatively, the Logger Manager reference may be obtained
// each time it is used
// Here, for example, when calling its get_logger method
VISLogger_ptr logger =
    VISLoggerManager::instance()->get_logger("LoggerName");
```

The methods of the Logger Manager are introduced along with the description of their use, in the sections that follow.

## Configuring ORB Logging

Even if an application does not log messages of its own, it may wish to configure the logging of messages by the ORB. The following aspects of ORB logging can be controlled:

- The level of ORB logging ('verbosity')
- The destination of logged messages - by installing different Forwarders
- The priority of the Forwarder Thread that runs the installed Forwarders

The following sections describe the ORB's logging output and how to control it.

### ORB Log Levels

Messages logged by the ORB have one of four Log Levels:

- Level 1: ERROR

Messages at this level indicate a fatal error during the operation of the ORB, that has caused one of the threads running ORB code to abort. Note that the cause of the error may be external to the ORB - for example a network interface configuration that cannot be supported.

- Level 2: WARNING

Messages at this level indicate a non-fatal error during the operation of the ORB. A problem was encountered that may cause subsequent failures or unexpected behavior, but the ORB will carry on trying to work as normal for now. Again, the cause of the problem may be external to the ORB.

- Level 3: INFORMATION

Messages at this level provide 'verbose' information about the normal operation of the ORB. For example, information about the successful configuration of a Server Engine at the time of its creation.

- Level 4: DEBUG

Messages at this level provide detailed information about certain aspects of the ORB's operation. They do not normally need to be viewed, but may be useful in certain debugging scenarios.

By default only messages at Log Levels 1 and 2 (ERROR and WARNING) will be logged by the ORB. The number of levels that are logged can be increased or decreased on a per-ORB component basis. ORB components are described in the next section.

### ORB Logging Components

For the purpose of logging, VisiBroker is divided into a number of components:

- ORB

The majority of VisiBroker, including the Object Request Broker itself.

- POA  
Portable Object Adapters. Note that individual POAs are not distinguished as separate logging components, so the same level of logging output will be used for all POAs. However, POA component messages usually identify the POA concerned as part of the logged message.
  - OSAgent  
The code of the OSAgent itself, and also the agent client code ('DSUser' code) that is used by an ORB when it interacts with the OSAgent.
  - LocSvc  
The 'Location Service' programmatic API to the OSAgent.
  - CosName  
The code of the COS Naming Service, that is provided with VisiBroker.
  - CosEvent  
The code of the COS Event Service, that is provided with VisiBroker.
- The level of logging may be configured on a per-component basis. The way to configure the level of output is described in the next section.
- By default, only messages at Log Levels 1 and 2 (ERROR and WARNING) are output, for all ORB components.

## Controlling the Level of ORB Logging

The level of ORB logging is controlled on a per-component basis, by specifying the highest message Log Level that should be logged for each ORB component.

The `Logger Manager` provides methods that allow the setting and reading of the maximum Log Level for each ORB component:

**Code example 141** The methods of `VISLoggerManager` that allow the setting and reading of the maximum Log Level on a per ORB component basis

```
class VISLoggerManager {
public:
...
void ORB_log_level( VISLogLevel level );
VISLogLevel ORB_log_level();

void POA_log_level( VISLogLevel level );
VISLogLevel POA_log_level();

void OSAgent_log_level( VISLogLevel level );
VISLogLevel OSAgent_log_level();

void LocSvc_log_level( VISLogLevel level );
VISLogLevel LocSvc_log_level();

void CosName_log_level( VISLogLevel level );
VISLogLevel CosName_log_level();

void CosEvent_log_level( VISLogLevel level );
VISLogLevel CosEvent_log_level();
...
};
```

For example, the following code sets the maximum Log Level to 3 (INFORMATION) for the ORB and POA components. This has the effect of producing 'verbose' output about the operation of the majority of the ORB.

**Code example 142** Setting the maximum Log Level for the ORB and POA components to Log Level 3 (INFORMATION) - 'verbose' output about most ORB operations

```
VISLoggerManager::instance()->ORB_log_level(3);  
VISLoggerManager::instance()->POA_log_level(3);
```

## Library `liblog_message_catalog.o` and Formatted ORB Log Messages

The VisiBrokerRT60 log message structure contains a message key and arguments. By default the message key and arguments are output by the Default Forwarder. Building a VisiBrokerRT60 application with the `liblog_message_catalog.o` VisiBroker library will cause formatted text messages corresponding to the message\_key and containing the arguments to be output. For more information on selecting VisiBrokerRT60 libraries for your application please refer to [“Step 6: Integrating VisiBroker-RT with Tornado/VxWorks”](#).

## Controlling the Priority of ORB Logging

The ORB logs its messages to a special Logger called the Default Logger. As with all Loggers, when a message is logged to the Default Logger, the message is written to a queue without being output to its final destination.

Each Logger has a Forwarder Thread associated with it, that reads the queued messages and executes the Forwarder code that outputs them to their final destination. The priority that the Forwarder Thread executes at can be controlled for each Logger, via a method on the VISLogger class.

The following code example demonstrates how to set the priority of the Default Logger's Forwarder Thread.

**Code example 143** Setting the priority of the Default Logger's Forwarder Thread

```
// Obtain handle to the Default Logger  
VISLogger_ptr logger =  
    VISLoggerManager::instance()->get_logger("DefaultLogger");  
  
// Set the Real-Time CORBA Priority of the Default Logger's  
// Forwarder  
// Note that this will only be effective before forwarding has  
// been enabled (no later than ORB_init)  
logger->forwarder_priority(27);
```

The priority is specified as a Real-Time CORBA priority value, and hence must be a valid value in the currently installed Real-Time CORBA Priority Mapping.

Note that the priority of the Forwarder Thread may only be changed before forwarding is enabled. For the Default Forwarder, forwarding is enabled automatically when `CORBA::ORB_init` is called, if it hasn't already been explicitly enabled before that time.

If a priority is not specified by the application, the Default Logger's Forwarder Thread priority defaults to the maximum priority in the Real-Time CORBA Priority Mapping installed at the time forwarding is enabled. This is the same behavior as for any other Loggers created by the application.

The enabling of forwarding is discussed further in the next section.

## Enabling Forwarding of ORB Logging

Like all Loggers, the Default Logger, used by the ORB, does not have the forwarding of logged messages enabled when it is created. Messages logged before forwarding is enabled are queued until it is enabled. This allows messages to be logged before the Logger or all of the output destinations have been fully configured - for example, before the priority of the Forwarder Thread has been configured, or during static initialization of C++ constructors, when the initialization of the C++ iostreams package may not yet have occurred.

However, the Default Logger differs from other Loggers in that for the Default Logger, forwarding is automatically enabled when either `CORBA::ORB_init` or `startOsagent` is called. Hence, at the time of calling `CORBA::ORB_init` or `startOsagent`, any messages previously logged by the ORB will be forwarded to the specified destinations.

Forwarding can still be explicitly enabled for the Default Logger, prior to calling `CORBA::ORB_init` or `startOsagent`. This might be done, for example, to investigate any messages logged by the ORB if a problem is encountered prior to calling `CORBA::ORB_init` or `startOsagent`. This should not normally be necessary. See the section [“Enabling Message Forwarding”](#) below for details of how to enable forwarding explicitly. The string identifier for the Default Logger is `'DefaultLogger'`.

## Controlling the Destination of ORB Logging

Messages logged to a Logger may be output to any number of destinations simultaneously, and the destinations that messages are logged to may be configured on a per-Logger basis, and at any time in the lifetime of the Logger.

Because the Default Logger, used by the ORB, is just a special Logger instance, the procedure for adding, removing and replacing logging destinations is the same as for Loggers created by applications. See the section [“Adding and Removing Logger Forwarders”](#), below. The string identifier for the Default Logger is `'DefaultLogger'`.

## Application Logging

Applications that wish to log messages via the VisiBroker logging mechanism may log messages to the same Default Logger that the ORB logs messages to, and may also create additional Loggers to log messages independently of the ORB's logging.

The following sections described how an application can create and configure additional Loggers, and log messages to them or the Default Logger.

## Creating or Obtaining a Reference to a Logger

A Logger object can be created using the `get_logger` method of the `VISLoggerManager` object.

**Code example 144** The `get_logger` method is used both to create new Loggers and to obtain a reference to existing Loggers.

```
typedef VISLogger * VISLogger_ptr;
class VISLoggerManager {
    ...
    VISLogger_ptr get_logger(const char * logger_name,
```

```

CORBA::Boolean create_flag = 1);
...
};

```

`get_logger` takes two parameters: a name for the Logger and a flag indicating whether a Logger should be created if one of that name does not already exist. With the second parameter set to true (non-zero), a new Logger will be created if one of that name does not already exist. If a Logger of that name already exists, a reference to it will be returned. This is the default behavior.

However, if the second parameter of `get_logger` is set to false (zero), the `get_logger` method will fail if a Logger of the specified name does not already exist. In that case a `CORBA::OBJECT_NOT_EXISTS` system exception is thrown.

Thus `get_logger` can be used both to create a new Logger and to obtain a reference to an existing Logger without attempting to create it. The following code example illustrates both these uses:

**Code example 145** Using `get_logger` to create a new Logger and to obtain a reference to an already existing Logger without attempting to create it.

```

// Obtain reference to a Logger called 'myAppLogger' - create it
// if it doesn't already exist
VISLogger_ptr my_app_logger;
VISTRY
{
    // Using one parameter version of get_logger indicates we want
    // to create this Logger
    // if it doesn't already exist
    my_app_logger =
        VISLoggerManager::instance()->get_logger("myAppLogger");
}
VISCATCH( CORBA::Exception, e )
{
    // Handle exceptions here
}

...
// Obtain reference to a Logger - throw an exception if it
// doesn't already exist
VISLogger_ptr logger;
VISTRY
{
    // second argument to get_logger indicates we do not want to
    // create this Logger
    // if it doesn't already exist
    logger =
        VISLoggerManager::instance()->get_logger("myAppLogger",0);
}
VISCATCH( CORBA::Exception, e )
{
    // Handle exceptions here
}

```

No reference counting is performed on Logger references (`VISLogger *` or `VISLogger_ptr`.)

## Setting the Forwarder Thread Priority of a Logger

When a message is logged to a Logger, the message is written to a queue without being output to its final destination. Each Logger has a Forwarder Thread associated with it, that executes any installed Forwarders - code that reads the queued messages and outputs them to their final destination. The

priority that the Forwarder Thread executes at can be controlled for each Logger, via a method on the VISLogger class.

The following code example demonstrates how to set the priority of a Logger's Forwarder Thread:

**Code example 146** Setting the priority of a Logger's Forwarder Thread

```
// Obtain handle to my Logger
VISLogger_ptr logger =
VISLoggerManager::instance()->get_logger("myAppLogger");

// Set the Real-Time CORBA Priority of the Logger's Forwarder
Thread
// Note that this will only be effective before forwarding has
been enabled
logger->forwarder_priority(27);
```

The priority is specified as a Real-Time CORBA priority value, and hence must be a valid value in the currently installed Real-Time CORBA Priority Mapping.

If a priority is not specified by the application, by default a Logger's Forwarder Thread will run at the maximum priority in the Real-Time CORBA Priority Mapping installed at the time forwarding is enabled. However, this default can be changed to any other priority in the installed Real-Time CORBA priority mapping, by calling the `default_forwarder_thread_priority` of the Logger Manager:

**Code example 147** The `default_forwarder_thread_priority` method may be used to change the default Real-Time CORBA Priority that Forwarder threads will run at

```
// Set the default Real-Time CORBA Priority value that Forwarder
Threads will run at, if
// a priority is not specified before forwarding is enabled
VISLoggerManager::instance()->
    default_forwarder_thread_priority(17)
;
```

Note that the priority of the Forwarder Thread is determined at the time when forwarding is enabled. Hence it may only be changed before forwarding is enabled. Enabling of forwarding is discussed in the next section.

## Enabling Message Forwarding

A Logger does not have the forwarding of logged messages enabled when it is created. Messages logged before forwarding is enabled are queued until it is enabled. This allows messages to be logged before the Logger or all of the output destinations have been fully configured - for example, before the priority of the Forwarder thread has been configured, or during static initialization of C++ constructors, when the initialization of the C++ iostreams package may not yet have occurred.

For all Loggers apart from the Default Logger (which is used by the ORB), logging must be explicitly enabled by calling the `enable_forwarding` method on that Logger.

**Code example 148** Forwarding of logged messages must be explicitly enabled for all Loggers apart from the Default Logger

```
// Enable forwarding once Logger and logging destinations are
ready
VISLogger_ptr my_app_logger =
    VISLoggerManager::instance()->get_logger("myAppLogger");

my_app_logger->enable_forwarding();
```

The Default Logger differs from other Loggers in that for the Default Logger, forwarding is automatically enabled when either CORBA::ORB\_init or startOsagent is called. See “[Enabling Forwarding of ORB Logging](#)” for details

## Logging a Message to a Logger

Applications log a message to a Logger by calling its log method:

```
class VISLogger {
...
void log( const char *      source_name,
VISLogLevel               level,
const char *              message_key,
VISLogArgs *              message_args,
const char *              source_thread_identifier,
const char *              location_code,
VISLogApplicationFields *application_fields );
...
};
```

The purpose of each of the parameters is as follows:

- **source\_name**  
Identifies the application or application component (in a complex system, with multiple logging sources) that is logging the message. The source name may be used by Forwarders, to determine how to handle the message. Certain source names are reserved by VisiBroker, and are used to determine which message catalog is used to produce the message text. Specifically, the names `vbroker\_en`, `nm\_vbroker\_en`, and `ev\_vbroker\_en` are reserved.
- **level**  
Indicates the Log Level of the message. Messages logged by the ORB use this field to indicate one of four levels, as described in the section “[ORB Log Levels](#)”. VISLogLevel is actually of type short, so the application is not restricted to just four levels. This parameter can be used just for informational purposes, or a Forwarder could make use of it to decide how to handle messages. (Note that the filtering of ORB messages based upon Log Level takes place in ORB code, before calling the log method. To filter messages based upon Log Level in a convenient fashion, an application could write a logging wrapper class, that the application calls instead of VISLogger::log, which only logs messages with currently selected Log Levels.)
- **message\_key**  
A string identifier that indicates what kind of message this is. The ORB uses a fixed set of message keys, so that there is a well known set of message types. These are then used as the keys in a `message catalog`, with the values being message `format strings`, that are used in combination with the message\_args parameter to produce the full text of the message. This separation of message text and arguments simplifies the support of internationalization in log message output. Applications may do the same. However, for a simpler form of logging, the application may just give the full text of its message in the message\_key parameter and leave the message\_args field null.
- **message\_args**  
These are copied by reference rather than by value. See the description of message\_key, above.

- `source_thread_identifier`  
Identifies the thread that logged this message. If this field is left null, the ORB will provide a default value.
- `location_code`  
Identified the location in application code that is logging this message on this occasion. For ORB log messages, this is the source code file name and line number of the calling line of ORB code (produced using the ANSI `C_FILE__` and `__LINE__` macros.) Applications may do the same, identify the location in some other way, or even leave this field blank.
- `application_fields`  
Any additional data that the application wishes to associate with this logged message. Copied by reference rather than by value. It is the application's responsibility to make sure that a Forwarder is installed that can interpret this data.

Note that the memory ownership semantics for the `message_args` and `application_fields` parameters are different to those of the other parameters. `message_args` and `application_fields` are passed by reference rather than by value. The Logger takes ownership of them and they are automatically deallocated after the last installed Forwarder has made use of them.

All other parameters are passed by value. That is, the Logger takes a copy of them when the log method is called. Thus it is the application's responsibility to deallocate any memory that it allocated for the `source_name`, `message_key`, `source_thread_identifier` or `location_code` parameters. The memory may be deallocated as soon as the call to the log method returns.

## Adding and Removing Logger Forwarders

Any number of Forwarders may be associated with a given Logger at the same time. Forwarders are added and removed through a set of methods on the `VISLogger` class:

**Code example 149** Methods used to add and remove logger forwarders

```
class VISLogger {
    ...
    // Add/Remove a Forwarder
    void add_forwarder( VISLoggerForwarder_ptr forwarder );
    void remove_forwarder( VISLoggerForwarder_ptr forwarder );

    // Remove the Default Forwarder
    void remove_default_forwarder();
    ...
};
```

`add_forwarder` and `remove_forwarder` allow a particular Forwarder to be added to or removed from the list of Forwarders associated with a Logger. They both take a handle to a Logger Forwarder object as a parameter.

`remove_default_forwarder` is provided to allow the removal of the Default Forwarder - the Forwarder that is associated with each Logger by default. This separate method is used as the application does not have a handle to the Default Forwarder, to provide as the parameter to `remove_forwarder`.

The next section describes how an application can create a Forwarder of its own.

## Implementing a Logger Forwarder

A new Logger Forwarder is implemented by defining a C++ class that inherits from the class `VISLoggerForwarder`:

**Code example 150** The `VISLoggerForwarder` class, from which Logger Forwarder implementations inherit

```
class VISLoggerForwarder {
public:
    VISLoggerForwarder();
    virtual ~VISLoggerForwarder();

    virtual void forward_message( VISLogMessage * message );
    virtual void handle_memory_failure(
        CORBA::ULongLongmessage_identifier,
        CORBA::ULongLongmessage_creation_time, VISLogLevellevel,
        const char *source_host,
        const char *source_name,
        const char *location_code,
        CORBA::ULongsource_process_identifier,
        const char *source_thread_identifier,
        VISLogApplicationFields *application_fields,
        const char *message_key,
        VISLogArgs *message_args );
};
```

A derived Forwarder class implements the forwarding behavior it desires by implementing the `forward_message` and `handle_memory_failure` methods. However, `VISLoggerForwarder` provides a default implementation for each of these methods, so that the application is not obliged to implement both of them. For details of the default implementation of these two methods see “[The Default Logger Forwarder](#)”, below.

`forward_message` is the method that is called under normal circumstances. It is called once for each message that is logged to any Logger that the Forwarder is associated with. The `VISLogMessage` data type that it is passed as a parameter has the following structure:

**Code example 151** The `VISLogMessage` data structure, that is passed as the parameter to `VISLoggerForwarder::forward_message`

```
struct VISLogMessage {
    CORBA::ULongLong          message_identifier;
    CORBA::ULongLong          message_creation_time;
    VISLogLevel                level;
    const char *              source_host;
    const char *              source_name;
    const char *              location_code;
    CORBA::ULong              source_process_identifier;
    const char *              source_thread_identifier;
    VISLogApplicationFields * application_fields;
    const char *              message_key;
    VISLogArgs *              message_args;

    VISLogMessage() {}
    ~VISLogMessage();
};
```

The fields in the `VISLogMessage` structure correspond to the parameters to the `VISLogger::log` method (described in the section “[Logging a Message to a Logger](#)” above), plus the following additional fields:

- `message_identifier`

A message sequence number, starting at one and incrementing for each message logged to that Logger.

- `message_creation_time`

A timestamp, taken from the system clock at the time the message was logged (rather than forwarded.) Held in the `TimeBase::TimeT` format: one unit is 100 nanoseconds or one tenth of a microsecond.

The Logger retains ownership of the `VISLogMessage` parameter. If the Forwarder wishes to keep a copy of any of the data it must copy it before `forward_message` returns. The memory associated with the `VISLogMessage` structure is deallocated by the Logger after the last installed Forwarder returns.

`handle_memory_failure` is called instead of `forward_message` in the event that the Logger experiences a memory allocation failure at any point during the logging of a message, up to and including the creation of the `VISLogMessage` parameter. As with `forward_message`, the Logger retains ownership of the parameters. Note that one or more of the parameters may be null, depending on when the memory allocation failure occurred.

The below code illustrates the installation of an application-defined Forwarder. The Forwarder is shown being installed on the Default Logger (which is used by the ORB), but any other Logger could be specified.

**Code example 152** Installation of an application-defined Forwarder, and removal of the Default Forwarder.

```
#include "vlogger.h"

class ExampleForwarder : public VISLoggerForwarder {
public:
void forward_message( struct VISLogMessage * message );
// Implementation not shown here - see below

void handle_memory_failure(
CORBA::ULongLong          message_identifier,
CORBA::ULongLong          message_creation_time,
VISLogLevel               level,
const char *               source_host,
const char *               source_name,
const char *               location_code,
CORBA::ULong               source_process_identifier,
const char *               source_thread_identifier,
VISLogApplicationFields * application_fields,
const char *               message_key,
VISLogArgs *               message_args );
// Implementation not shown here - see below
};

ExampleForwarder * example_forwarder; void install_forwarder()
{
// Obtain handle to Logger want to install Forwarder on
VISLogger_ptr logger =
VISLoggerManager::instance()->get_logger("DefaultLogger");

// Create instance of Forwarder and add to the list of
Forwarders installed for that Logger
example_forwarder = new ExampleForwarder;
logger->add_forwarder(example_forwarder);

// (Optionally) remove the Default Forwarder from this
Logger
logger->remove_default_forwarder();
}
}
```

The above example does not show the implementation of the `forward_message` and `handle_memory_failure` methods. For a sample implementation for these methods see the next section, "The Default Logger Forwarder".

## The Default Logger Forwarder

The Default Forwarder implements both the `forward_message` and `handle_memory_failure` methods of the `VISLoggerForwarder` base class.

The implementation of `forward_message` uses the `VISLogMessageCatalog` class to retrieve a message format string from an appropriate Message Catalog, if one is installed that is associated with the source name indicated in the message. If there is a message format that corresponds to the message key, it then uses the `VISLogMessageFormat` helper class to produce the full text of the message, by combining the message format string with any message arguments that were specified as part of the message. The message text is output to standard error (iostream `'cerr'`), along with the rest of the message fields.

`handle_memory_failure` just outputs the fields of the message that can be output without allocating memory to format them.

This is the code for `forward_message` and `handle_memory_failure`:

### Code example 153 Code for the Default Forwarder

```
#include "vlogger.h"
#include "vlogmfmt.h"
#include "vport.h"

void VISLoggerForwarder::forward_message( struct VISLogMessage *
message )
{
    // Obtain the message catalog that corresponds to the message
    // source
    // (If there is one - else null)
    VISLogMessageCatalog_ptr catalog =
        VISLogMessageCatalog::instance(message->source_name);

    // If there is a message key (and a catalog), look the key up
    // in the message catalog, to get corresponding format
    const char * message_format = 0;
    if (message->message_key && catalog)
    {
        message_format = catalog->search(message->message_key);
    }

    // If there was a format string for that key, use it to format
    // the text
    const char * message_text = 0;
    CORBA::Boolean format_error = 0;
    if (message_format)
    {
        VISTRY
        {
            message_text = VISLogMessageFormat::format(
                message_format, message->message_args );
        }
        VISCATCH(CORBA::Exception, e) {
            format_error = 1;
        }
        VISEND_CATCH
    }

    // Convert message identifier (ulonglong) to string
    char * msg_id_str =
        VISPortable::ulonglong_to_str(message->message_identifier);

    // Convert message creation time (TimeBase::TimeT) to seconds
    CORBA::ULongLong secs = message->message_creation_time /
        10000000;
    CORBA::ULong nsec = (message->message_creation_time %
        10000000) * 100;
    char * secs_str = VISPortable::ulonglong_to_str(secs);
    char time_str[32];
```

```

sprintf(time_str, "%s.%09lu", secs_str, nsec);

// Output the message to standard error
cerr << endl << "Logging: message " << msg_id_str
<< " time " << time_str << " level " << message->level <<
endl;

delete [] msg_id_str;
delete [] secs_str;

if (message_text)
{
    cerr << "Message: " << message_text << endl;
} else
{
    // If didn't end up with formatted message text, explain why
    if (!catalog)
    {
        cerr << "Msg key   : " << message->message_key
            << " (Message catalog '" << message->source_name
            << "' not installed)" << endl;
    } else if (!message->message_key)
    {
        cerr << "Msg key   : (null)" << endl;
    }
    else if (format_error)
    {
        cerr << "Msg Key   : " << message->message_key
            << " (Error formatting message text)" << endl;
    } else
    {
        cerr << "Msg key: " << message->message_key
            << " (No entry in message catalog)" << endl;
    }
}

// Output message arguments if there are any
if (!message->message_args ||
(message->message_args ->num_args() == 0))
{
    cerr << "Arguments: (none)" << endl;
} else
{
    cerr << "Arguments:";
    for (int i=0; i < message->message_args->num_args(); i++)
    {
        VISLogArgsType* arg = (*(message->message_args))[i];
        switch(arg->data_type())
        {
            case VISLogArgsType::INTEGER:
            {
                cerr << " Integer("
                    << ((VISLogInteger*)arg)->integer_value() << ")";
                break;
            }
            case VISLogArgsType::STRING:
            {
                cerr << " String("
                    << ((VISLogString*)arg)->string_value() << ")";
                break;
            }
            case VISLogArgsType::BOOLEAN:
            {
                cerr << " Boolean(";
                if (((VISLogBoolean*)arg)->boolean_value())
                {
                    cerr << "true)";
                } else
                {
                    cerr << "false)";
                }
                break;
            }
        }
    }
}

```

```

    }
}

cerr << endl;
}
}

cerr << "Source: " << (message->source_host ?
    message->source_host : "(null)")
    << " " << (message->source_name ? message->source_name :
"(null)")
    << endl;
cerr << "Location : "
    << (message->location_code ? message->location_code :
"(null)")
    << endl;
cerr << "PID : " << message->source_process_identifier << "
TID : "
    << (message->source_thread_identifier ?
    message->source_thread_identifier : "(null)")
    << endl;

// application fields are not being output

// Delete the formatted text
delete [] message_text;
}

void VISLoggerForwarder::handle_memory_failure(
    CORBA::ULongLong message_identifier,
    CORBA::ULongLong message_creation_time,
    VISLogLevel level,
    const char * source_host,
    const char * source_name,
    const char * location_code,
    CORBA::ULong source_process_identifier,
    const char * source_thread_identifier,
    VISLogApplicationFields * application_fields,
    const char * message_key,
    VISLogArgs * message_args )
{
    // Output subset of data that is output by forward_message
    // Don't convert long long values, as this requires memory
    allocation

    cerr << endl << "*** Logging Memory Failure ** for message
with id"
        << (unsigned long)message_identifier << endl
        << " (id truncated to 32 bits)" << " level " << level
        << endl;
    // Don't format message text, as this requires memory
    allocation

    cerr << "Msg key: " << (message_key ? message_key :
"(null)")
        << endl;
    cerr << "Source: " << (source_host ? source_host : "(null)")
        << " " << (source_name ? source_name : "(null)") << endl;
    cerr << "Location : " << (location_code ? location_code :
"(null)") << endl;
    cerr << "PID: " << source_process_identifier << "TID : "
        << (source_thread_identifier ? source_thread_identifier
: "(null)")
        << endl;
}
}

```

# Part 7

## Dynamic CORBA Concepts

### In this part

This part contains the following chapters:

<a href="#">Using Interface Repositories</a>	page 335
<a href="#">Using the Dynamic Invocation Interface</a>	page 347
<a href="#">Using the Dynamic Skeleton Interface</a>	page 365
<a href="#">Using the Dynamically Managed Types</a>	page 375

### NOTE

The library `liborb_dyn.o` is required when building a VisiBrokerRT60 application to support use of the Dynamic CORBA concepts. For a description of all the libraries provided by the VisiBroker-RT for C++ product, see "[Step 6: Integrating VisiBroker-RT with Tornado/VxWorks](#)".

***In the original, this and several other paragraphs refer to "Step 5: Selecting VisiBroker libraries"; but no such step exists. I think this is the right one!***



# Using Interface Repositories

*An interface repository (IR) contains descriptions of CORBA object interfaces. The data in an IR is the same as in IDL files—descriptions of modules, interfaces, operations, and parameters—but it is organized for runtime access by clients. A client can browse an interface repository (perhaps serving as an online reference tool for developers) or can look up the interface of any object for which it has a reference (perhaps in preparation for invoking the object with the Dynamic Invocation Interface).*

*Reading this chapter will enable you to create an interface repository and access it with VisiBroker-RT for C++ utilities or with your own code.*

## NOTE

The library `liborb_dyn.o` is required when building a VisiBrokerRT60 application to support use of the Dynamic CORBA concepts. For a description of all the libraries provided by the VisiBroker-RT for C++ product, see [<~XRef>“Step 6: Integrating VisiBroker-RT with Tornado/VxWorks”](#).

The Interface Repository (IR) is available ONLY on the development host. VisiBroker-RT for C++ does NOT provide an Interface Repository as a runtime library.

Additionally the IR provides functionality which address the more Dynamic aspects of CORBA, and therefore the IR is excluded as per the "minimum CORBA" OMG specification. The "minimum CORBA" OMG specification identifies dynamic functionality which should be excluded from an ORB, in an effort to reduce the ORB footprint.

For details, see the minimum CORBA specification document, OMG document number orbos/ 98-08-04. This document is available for download using the URL <ftp://ftp.omg.org/pub/docs/orbos/98-08-04.pdf>.

## What is an interface repository?

An interface repository (IR) is like a database of CORBA object interface information that enables clients to learn about or update interface descriptions at runtime. In contrast to the VisiBroker-RT for C++ Location Service, described in the chapter [“Using the Location Service”](#) which holds data describing object *instances*, an IR’s data describes *interfaces* (types). There may or may not be available instances that satisfy the interfaces stored in an IR. The information in an IR is equivalent to the information in an IDL file (or files), but it is represented in a way that is easier for clients to use at runtime.

Clients that use interface repositories may also use the Dynamic Invocation Interface (DII) described in [“Using the Dynamic Invocation Interface”](#). Such clients use an interface repository to learn about an unknown object’s interface, and they use the DII to invoke methods on the object. However, there is no necessary connection between an IR and the DII. For example, someone could use the IR to write an “IDL browser” tool for developers—in such a tool, dragging a method description from the browser to an editor would insert a template method invocation into the developer’s source code. In this example, the IR is used without the DII.

You create an interface repository with the VisiBroker-RT for C++ `irep` program, which is the IR server (implementation). *The `irep` program is a development host ONLY program.* You can update or populate an interface

repository with the VisiBroker-RT for C++ `idl2ir` program (also a development host ONLY program), or you can write your own IR client that inspects an interface repository, updates it, or does both.

## What does an interface repository contain?

An interface repository contains hierarchies of objects whose methods divulge information about interfaces. Although interfaces are usually thought of as describing objects, using a collection of objects to describe interfaces makes sense in a CORBA environment because it requires no new mechanism such as a database.

As an example of the kinds of objects an IR can contain, consider that IDL files can contain IDL module definitions, and modules can contain interface definitions, and interfaces can contain operation (method) definitions. Correspondingly, an interface repository can contain `ModuleDef` objects which can contain `InterfaceDef` objects, which can contain `OperationDef` objects. Thus, from an IR `ModuleDef`, you can learn what `InterfaceDef`s it contains. The reverse is also true—given an `InterfaceDef` you can learn what `ModuleDef` it is contained in. All other IDL constructs—including exceptions, attributes, and valuetypes—can be represented in an interface repository.

An interface repository also contains typecodes. Typecodes are not explicitly listed in IDL files, but are automatically derived from the types (`long`, `string`, `struct`, and so on) that are defined or mentioned in IDL files. Typecodes are used to encode and decode instances of the CORBA `any` type—a generic type that stands for any type and is used with the dynamic invocation interface.

## How many interface repositories can you have?

Interface repositories are like other objects—you can create as many as you like. There is no VisiBroker-RT for C++-mandated policy governing the creation or use of IRs. You determine how interface repositories are deployed and named at your site. You may, for example, adopt the convention that a central interface repository contains the interfaces of all “production” objects, and developers create their own IRs for testing.

### Note

Interface repositories are writable and are not protected by access controls. An erroneous or malicious client can corrupt an IR or obtain sensitive information from it.

If you want to use the `_get_interface_def()` method defined for all objects, you must have at least one interface repository server running so the ORB can look up the interface in the IR. If no interface repository is available, or if the IR that the ORB binds to has not been loaded with an interface definition for the object, `_get_interface_def()` raises a `NO_IMPLEMENT` exception.

## Creating and viewing an interface repository with `irep`

The VisiBroker-RT for C++ interface repository server is called `irep`, and is located in the `<install_location>/VisiBrokerRT60/bin` directory. The `irep` program runs as a daemon.

## Creating an interface repository with irep

Use the `irep` program to create an interface repository and view its contents. The usage syntax for the `irep` program is as follows:

```
irep <driverOptions> <otherOptions> IRepName [file.idl]
```

The syntax for creating an interface repository in the `irep` is described in the following table:

Syntax	Description
<code>IRepName</code>	Specifies the instance name of the interface repository. Clients can bind to this interface repository instance by specifying this name.
<code>file.idl</code>	Specifies the IDL file whose contents <code>irep</code> will load into the interface repository it creates and will store the IR contents into when it exits. If no file is specified, <code>irep</code> creates an empty interface repository.

The `irep` arguments are defined in the following table.

Argument	Description
<b>Driver options</b>	
<code>-J&lt;java option&gt;</code>	Pass the option to JVM directly.
<code>-VBJversion</code>	Print VBJ version
<code>-VBJdebug</code>	Print VBJ debug information.
<code>-VBJclasspath</code>	Specify classpath, precedes CLASSPATH env variable.
<code>- VBJprop &lt;name&gt;[=&lt;value&gt;</code>	Pass name/value pair to JVM.
<code>-VBJjavavm &lt;jvmpath&gt;</code>	Specify JVM path.
<code>-VBJaddJar &lt;jarfile&gt;</code>	Append jarfile to the CLASSPATH before execing the JVM.
<b>Other options</b>	
<code>-D, -define foo[=bar]</code>	Define a preprocessor macro, optionally with value.
<code>-I, -include &lt;dir&gt;</code>	Specify additional directory for <code>#include</code> searching.
<code>-P, -no_line_directives</code>	Do not emit <code>#line</code> directives from preprocessor. The default is off.
<code>-H, -list_includes</code>	Display <code>#included</code> file names as they are encountered. The default is off.
<code>-C, -retain_comments</code>	Retain comments in preprocessed output. The default is off.
<code>-U, -undefine foo</code>	Undefine a preprocessor macro.
<code>-[no_]idl_strict</code>	Strict OMG-standard interpretation of IDL source. The default is off.
<code>-[no_] warn_ unrecognized_pragmas</code>	Warn if a <code>#pragma</code> is not recognized. The default is on.
<code>-[no_] back_compat_mapp ing</code>	Use mapping that is compatible with VisiBroker 3.x.
<code>-h, -help, -usage, -?</code>	Print this usage information.
<code>-version</code>	Display software version numbers.
<code>-install &lt;service name&gt;</code>	Install as a NT service.
<code>-remove &lt;service name&gt;</code>	Deinstall this NT service.

The following example shows how an interface repository named `TestIR` can be created from a file called **bank.idl**:

```
irep myIrep bank.idl
```

## Viewing the contents of the interface repository

You can view the contents of the interface repository with either the VisiBroker-RT for C++ `ir2idl` utility, or the VisiBroker-RT for C++ Console application. The syntax for the `ir2idl` utility is:

```
ir2idl [-irep IRname]
```

The syntax for viewing the contents of an interface repository in the `irep` is described in the following table:

Syntax	Description
<code>-irep IRname</code>	Directs the program to bind to the interface repository instance named <code>IRname</code> . If the option is not specified, it binds to any interface repository returned by the Smart Agent.

For more details on the `ir2idl` utility arguments see the section on “idl2ir” in the chapter “Programmer tools” of the VisiBroker-RT for C++ *Reference Guide*.

## Updating an interface repository with idl2ir

You can update an interface repository with the VisiBroker-RT for C++ `idl2ir` utility, which is an IR client. The syntax for the `idl2ir` utility is:

```
idl2ir [arguments] idl_file_list
```

For more details on the `idl2ir` utility arguments see the section on “idl2ir” in the VisiBroker-RT for C++ *Reference Guide*.

The following example shows how the `TestIR` interface repository would be updated with definitions from the **bank.idl** file.

```
idl2ir -irep myIrep -replace bank.idl
```

Entries in an interface repository cannot be removed using the `idl2ir` or `irep` utilities. To remove an item,

- 1 Exit or quit the `irep` program.
- 2 Edit the IDL file named in the `irep` command line.
- 3 Start `irep` again with the updated file.

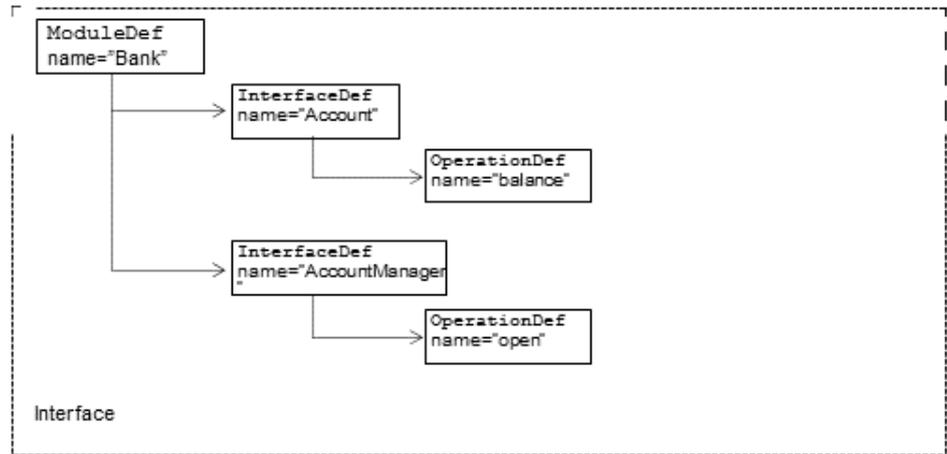
Interface repositories have a simple transaction service. If the specified IDL file fails to load, the interface repository rolls back its content to its previous state. After loading the IDL, the interface repository commits its state to be used in subsequent transactions. For any repository, there is a file `IRname.rollback` in the home directory that contains the state of the last uncommitted transaction.

## Understanding the structure of the interface repository

An interface repository organizes the objects it contains into a hierarchy that corresponds to the way interfaces are defined in an IDL specification. Some objects in the interface repository contain other objects, just as an IDL module definition might contain several interface definitions. Consider how the following example IDL file would translate to a hierarchy of objects in an interface repository.

### IDL sample 23 bank.idl file

```
// bank.idl
module Bank {
    interface Account {
        float balance();
    };
    interface AccountManager {
        Account open(in string name);
    };
};
```



**Figure 43** Interface repository object hierarchy for Bank.idl

`OperationDef` object contains references to additional data structures (not interfaces) that hold the parameters and return type.

## Identifying objects in the interface repository

The following table shows the objects that are provided to identify and classify interface repository objects.

**Table 29** Objects used to identify and classify interface repository objects

Item	Description
name	A character string that corresponds to the identifier assigned in an IDL specification to a module, interface, operation, and so forth. An identifier is not necessarily unique.
id	A character string that uniquely identifies an <code>IObject</code> . A <code>RepositoryID</code> contains three components, separated by colon ( <code>:</code> ) delimiters. The first component is <code>"IDL:"</code> and the last is a version number such as <code>"1.0"</code> . The second component is a sequence of identifiers separated by slash ( <code>/</code> ) characters. The first identifier is typically a unique prefix.
def_kind	An enumeration that defines values which represent all the possible types of interface repository objects.

## Types of objects that can be stored in the interface repository

[Table 30](#) summarizes the objects that can be contained in an interface repository. Most of these objects correspond to IDL syntax elements. A

`StructDef`, for example, contains the same information as an IDL struct declaration, an `InterfaceDef` contains the same information as an IDL interface declaration, all the way down to a `PrimitiveDef` which contains the same information as an IDL primitive (`boolean`, `long`, and so forth) declaration.

**Table 30** Objects that can be stored in the interface repository

Object type	Description
<code>Repository</code>	Represents the top-level module that contains all other objects.
<code>ModuleDef</code>	Represents an IDL module declaration that can contain <code>ModuleDefs</code> , <code>InterfaceDefs</code> , <code>ConstantDefs</code> , <code>AliasDefs</code> , <code>ExceptionDefs</code> , and the IR equivalents of other IDL constructs that can be defined in IDL modules.
<code>InterfaceDef</code>	Represents an IDL interface declaration and contain <code>OperationDefs</code> , <code>ExceptionDefs</code> , <code>AliasDefs</code> , <code>ConstantDefs</code> , and <code>AttributeDefs</code> .
<code>AttributeDef</code>	Represents an IDL attribute declaration.
<code>OperationDef</code>	Represents an IDL operation (method) declaration. Defines an operation on an interface.  It includes a list of parameters required for this operation, the return value, a list of exceptions that may be raised by this operation, and a list of contexts.
<code>ConstantDef</code>	Represents an IDL constant declaration.
<code>ExceptionDef</code>	Represents an IDL exception declaration.
<code>ValueDef</code>	Represents a valuetype definition containing lists of constants, types, valuemembers, exceptions, operations, and attributes.
<code>ValueBoxDef</code>	Represents a simple boxed valuetype of another IDL type.
<code>ValueMemberDef</code>	Represents a member of the valuetype.
<code>NativeDef</code>	Represents a native definition. Users can not define their own natives.
<code>StructDef</code>	Represents an IDL structure declaration.
<code>UnionDef</code>	Represents an IDL union declaration.
<code>EnumDef</code>	Represents an IDL enumeration declaration.
<code>AliasDef</code>	Represents an IDL typedef declaration. Note that the IR <code>TypedefDef</code> interface is a base interface that defines common operations for <code>StructDefs</code> , <code>UnionDefs</code> , and others.
<code>StringDef</code>	Represents an IDL bounded string declaration.
<code>SequenceDef</code>	Represents an IDL sequence declaration.
<code>ArrayDef</code>	Represents an IDL array declaration.
<code>PrimitiveDef</code>	Represents an IDL primitive declaration: <code>null</code> , <code>void</code> , <code>long</code> , <code>ushort</code> , <code>ulong</code> , <code>float</code> , <code>double</code> , <code>boolean</code> , <code>char</code> , <code>octet</code> , <code>any</code> , <code>TypeCode</code> , <code>Principal</code> , <code>string</code> , <code>objref</code> , <code>longlong</code> , <code>ulonglong</code> , <code>longdouble</code> , <code>wchar</code> , <code>wstring</code> .

## Inherited interfaces

Three non-instantiatable (that is, abstract) IDL interfaces define common methods that are inherited by many of the objects contained in an IR (see [Table 30](#)). [Table 31](#) summarizes these widely inherited interfaces.

**Table 31** Interfaces inherited by many IR objects

Interface	Inherited by	Principal query methods
IObject	All IR objects including <code>Repository</code>	<code>def_kind()</code> —Returns an IR object's definition kind, for example, module or interface
Container	IR objects that can contain other IR objects, for example, module or interface	<code>lookup()</code> —Looks up a contained object by name <code>contents()</code> —Lists the objects in a Container <code>describe_contents()</code> —Describes the objects in a Container
Contained	IR objects that can be contained in other objects, that is, Containers	<code>name()</code> —Name of this object <code>defined_in()</code> —Container that contains an object <code>describe()</code> —Describe an object <code>move()</code> —Moves an object into another container.

## Accessing an interface repository

Your client program can use an interface repository's IDL interface to obtain information about the objects it contains. Your client program can bind to the `Repository` and then invoke the methods shown in Code sample 27.1 . A complete description of this interface can be found in the *VisiBroker-RT for C++ Reference Guide*.

### Code example 154 Repository class

```
class CORBA {
    class Repository : public Container {
    ...
    CORBA::Contained_ptr lookup_id(const char * search_id);
    CORBA::PrimitiveDef_ptr get_primitive(CORBA::PrimitiveKind
kind);
    CORBA::StringDef_ptr create_string(CORBA::ULong bound);
    CORBA::SequenceDef_ptr create_sequence(CORBA::ULong bound,
CORBA::IDLType_ptr element_type);
    CORBA::ArrayDef_ptr create_array(CORBA::ULong length,
CORBA::IDLType_ptr element_type);
    ...
    };
    ...
};
```

### Note

A program that uses an interface repository must be compiled with the `-D_VIS_INCLUDE_IR` flag.

## Example programs

The Interface Repository example (`<install_location>/VisiBrokerRT60/examples/ir`) has a simple `AccountManager` interface to create an account and open/reopen an account. At the initialization time the `AccountManager` implementation bootstraps the Interface Repository definition for the managed `Account` interface with purpose to expose to the clients the additional operation that has been already implemented by this particular `Account` implementation. The clients now can access all known (described in IDL) operations as they do this usually and, additionally, the can verify with the Interface Repository the support for other operations and invoke them. This example illustrates how we can manage the Interface Repository

definition objects and how we can do the remote object's introspection using the Interface Repository.

Before this program can be tested, the following conditions should exist:

- OSAgent should be up and running.
- Interface repository should be started on the development host using irep.
- Interface Repository should be loaded with an IDL file either by the command line when you start the Interface Repository, or by using idl2ir.

**Code example 155** Looking up an interface's operations and attributes in an IR

```
#ifndef VIS_INCLUDE_IR
#define _VIS_INCLUDE_IR
#endif

#include <vxWorks.h>
#include "corba.h"
#include <math.h>
#include "bank_c.hh"

extern CORBA::ORB_var orb;

char* getDescription(CORBA::ORB_ptr orb, Bank::Account_ptr
account)
{
    CORBA::Any_ptr resultAny;
    CORBA::NamedValue_var result;
    CORBA::NVList_var operation_list;
    CORBA::Request_var request;
    CORBA::OperationDef_var odef;

    // Obtain operation description for the "describe" method of
    the account
    VISTRY {

        // Obtain a reference to the Interface Repository
        CORBA_Repository_var ir = CORBA_Repository::_narrow(
            orb->resolve_initial_references("InterfaceRepository"));

        // Obtain a reference to the Bank::Account interfaceDef
        CORBA::InterfaceDef_var intf;
        VISIFNOT_EXCEP

        {
            intf =
            CORBA_InterfaceDef::_narrow(ir->lookup("::Bank::Account"));
            if (intf == CORBA::InterfaceDef::_nil()) {
                cout << "Account returned a nil interface definition. " <<
                endl;
                cout << "Be sure an Interface Repository is running and"
                << endl;
                cout << "properly loaded." << endl;
                return (char *)NULL;
            }
        }
        VISEND_IFNOT_EXCEP
    }

    CORBA::Contained_var container;
    VISIFNOT_EXCEP
        container = intf->lookup("describe");
    VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
    {
        odef = CORBA::OperationDef::_narrow(container);
        if (odef == CORBA::OperationDef::_nil()) {
            cout << "Can not find \"describe\" method in irep." <<

```

```

endl;
    cout << "Please check if Server application is started" <<
endl;
    return (char *)NULL;
}
}
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
    orb->create_operation_list(odef, operation_list.out());
VISEND_IFNOT_EXCEP
}

VISCATCH (CORBA::Exception, e) {
    cout << "Error while obtaining operation list: " << e <<
endl;
    return (char *)NULL;
}
VISEND_CATCH

// Create request that will be sent to the account object
VISTRY {

    // Create placeholder for result
    orb->create_named_value(result.out());
    VISIFNOT_EXCEP
        resultAny = result->value();
    VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
        resultAny->replace(odef->result(), NULL);

    VISEND_IFNOT_EXCEP

    // Create the request
    VISIFNOT_EXCEP
        account->create_request(
            CORBA::Context::_nil(), "describe", operation_list,
result,
            request.out(), 0);
    VISEND_IFNOT_EXCEP
}

VISCATCH (CORBA::Exception, e) {
    cout << "Error while creating request: " << e << endl;
    return (char *)NULL;
}
VISEND_CATCH

// Execute the request
VISTRY {
    request->invoke();
    CORBA::Environment_ptr env = request->env();
    if (env->exception()) {
        cout << "Exception occured: " << *(env->exception()) <<
endl;
        return (char *)NULL;
    }
    else {
        char *desc;
        *resultAny >>= desc;
        return CORBA::string_dup(desc);
    }
}

VISCATCH (CORBA::Exception, e) {
    cout << "Error while invoking request: " << e << endl;
    return (char *)NULL;
}
VISEND_CATCH

return (char *)NULL;
}

```

```

static void bank_client(char * in_name, char * new_balance);

void start_bank_client(char * in_name, char * new_balance)
{
    char *      taskName = "BANK_CLNT";
    int         Prio = 100;
    int         option = VX_FP_TASK;
    int         stackSize = 20000;

    taskSpawn(taskName,
              Prio,
              option,
              stackSize,
              (FUNCPTR)bank_client,
              (int)in_name, (int)new_balance, 0, 0, 0, 0, 0, 0, 0);
}

void bank_client(char * in_name, char *new_balance)
{
    VISTRY {

        // Get the manager Id PortableServer::ObjectId_var managerId
        = PortableServer::string_to_ObjectId("BankManager");

        // Locate an account manager Bank::AccountManager_var
manager;
        VISIFNOT_EXCEP
            manager = Bank::AccountManager::_bind("/bank_ir_poa",
managerId);
        VISEND_IFNOT_EXCEP

        // Request the account manager to open a named account
        if (!in_name)
        {
            in_name="Jack B. Quick";
        }
        CORBA::String_var name = CORBA::string_dup(in_name);

        Bank::Account_var account;
        VISIFNOT_EXCEP
            account = manager->open(name);
        VISEND_IFNOT_EXCEP

        // Get the balance of the account
        CORBA::Float balance;
        VISIFNOT_EXCEP
            balance = account->balance();
        VISEND_IFNOT_EXCEP

        // Print out the balance
        VISIFNOT_EXCEP
            cout << "The old balance in " << name <<
                "'s account is $" << balance << endl;
        VISEND_IFNOT_EXCEP

        // Calculate and set a new balance
        VISIFNOT_EXCEP
        {
            balance = new_balance ? atof(new_balance) :
                abs(rand()) % 111111 / 50.0;
            account->balance(balance);
        }
        VISEND_IFNOT_EXCEP

        // Get the balance description if it is possible and print
        VISIFNOT_EXCEP
        {
            CORBA::String_var desc = getDescription(orb, account);
            cout << "New account description:" << endl << desc <<
                endl;
        }
        VISEND_IFNOT_EXCEP
    }
}

```

```
    }  
    VISCATCH(CORBA::Exception, e) {  
        cout << "Exception occurred: " << e << endl;  
    }  
    VISEND_CATCH  
  
    return;  
}  
...
```



# Using the Dynamic Invocation Interface

*The developers of most client programs know the types of the CORBA objects their code will invoke, and they include the compiler-generated stubs for these types in their code. By contrast, developers of generic clients cannot know what kinds of objects their users will want to invoke. Such developers use the Dynamic Invocation Interface (DII) to write clients that can invoke any method on any CORBA object from knowledge obtained at runtime.*

## NOTE

The library `liborb_dyn.o` is required when building a `VisiBrokerRT52` application to support the use of Dynamic Invocation Interface.. For a description of all the libraries provided by the `VisiBroker-RT` for C++ product, see [“Step 6: Integrating VisiBroker-RT with Tornado/VxWorks”](#).

The Dynamic Invocation Interface (DII) is not supported as part of the “minimum CORBA” version of `VisiBroker-RT` for C++ (i.e. `liborb_min.o`).

The “minimum CORBA” OMG specification identifies dynamic functionality which should be excluded from an ORB, in an effort to reduce the ORB footprint.

For details, see the minimum CORBA specification document, OMG document number `orbos/98-08-04`. This document is available for download using the URL <ftp://ftp.omg.org/pub/docs/orbos/98-08-04.pdf>.

## What is the Dynamic Invocation Interface?

The Dynamic Invocation Interface (DII) enables a client program to invoke a method on a CORBA object whose type was unknown at the time the client was written. The DII contrasts with the default static invocation, which requires that the client source code include a compiler-generated stub for each type of CORBA object that the client intends to invoke. In other words, a client that uses static invocation declares in advance the types of objects it will invoke. A client that uses the DII makes no such declaration because its programmer doesn’t know what kinds of objects will be invoked.

The advantage of the DII is flexibility—it can be used to write generic clients that can invoke any object, including objects whose interfaces did not exist when the client was compiled.

The DII has two disadvantages:

- It is more difficult to program (in essence, your code must do the work of a stub).
- Invocations take longer because more work is done at runtime.

The DII is purely a client interface—static and dynamic invocations are identical from an object implementation’s point of view.

You can use the DII to build clients like these:

- Bridges or adapters between script environments and CORBA objects. For example, a script calls your bridge, passing object and method identifiers and parameter values. Your bridge constructs and issues a dynamic

request, receives the result, and returns it to the scripting environment. Such a bridge could not use static invocation because its developer could not know in advance what kinds of objects the script environment would want to invoke.

- Generic object testers. For example, a client takes an arbitrary object identifier, looks up its interface in the interface repository (see “[Using Interface Repositories](#)”), and then invokes each of its methods with artificial argument values. Again, this style of generic tester could not be built with static invocation.

### Note

Clients *must* pass valid arguments in DII requests. Failure to do so can produce unpredictable results, including server crashes. Although it is possible to dynamically type-check parameter values with the interface repository, it is expensive. For best performance, ensure that the code (for example, script) that invokes a DII-using client can be trusted to pass valid arguments.

## Introducing the main DII concepts

The dynamic invocation interface is actually distributed among a handful of CORBA interfaces. Furthermore, the DII frequently offers more than one way to accomplish a task — the trade-off being programming simplicity versus performance in special situations. As a result, DII is one of the more difficult CORBA facilities to grasp. This section is a starting point, a high-level description of the main ideas. Details, including code examples, are provided later in the chapter.

To use the DII you need to understand these concepts, starting from the most general:

- Request objects
- Any and Typecode objects
- Request sending options
- Reply receiving options

### Using request objects

A `Request` object represents one invocation of one method on one CORBA object. If you want to invoke two methods on the same CORBA object, or the same method on two different objects, you need two `Request` objects. To invoke a method you first need an object reference representing the CORBA object—the target reference. Using the target reference, you create a `Request`, populate it with arguments, send the `Request`, wait for the reply, and obtain the result from the `Request`.

There are two ways to create a `Request`.

- 1 The simpler way is to invoke the target object’s `_request()` method, which all CORBA objects inherit. This does not, in fact, invoke the target object. You pass `_request()` the IDL name of the method you intend to invoke in the `Request`, for example, “`get_balance`”. To add argument values to a `Request` created with `_request()`, you invoke the `Request`’s `add_value()` method for each argument required by the method you intend to invoke. To pass one or more Context objects to the target, you must add them to the `Request` with its `ctx()` method.

Although not intuitively obvious, you must also specify the type of the `Request`’s result with its `result()` method. For performance reasons, the messages exchanged between ORBs do not contain type information.

By specifying a place holder result type in the `Request`, you give the ORB the information it needs to properly extract the result from the reply message sent by the target object. Similarly, if the method you are invoking can raise user exceptions, you must add place holder exceptions to the `Request` before sending it.

- 2 The more complicated way to create a `Request` object is to invoke the target object's `_create_request()` method, which, again, all CORBA objects inherit. This method takes several arguments which populate the new `Request` with arguments and specify the types of the result and user exceptions, if any, that it may return. To use the `_create_request()` method you must have already built the components that it takes as arguments. The potential advantage of the `_create_request()` method is performance. You can reuse the argument components in multiple `_create_request()` calls if you invoke the same method on multiple target objects.

### Note

There are two overloaded forms of the `_create_request()` method—one that includes `ContextList` and `ExceptionList` parameters, and one that does not. If you want to pass one or more `Context` objects in your invocation, and/or the method you intend to invoke can raise one or more user exceptions, you must use the `_create_request()` method that has the extra parameters.

## Encapsulating arguments with the Any type

The target method's arguments, result, and exceptions are each specified in special objects called `Any`s. An `Any` is a generic object that encapsulates an argument of any type. An `Any` can hold any type that can be described in IDL. Specifying an argument to a `Request` as an `Any` allows a `Request` to hold arbitrary argument types and values without making the compiler complain of type mismatches. (The same is true of results and exceptions.)

An `Any` consists of a `TypeCode` and a value. A value is just a value, and a `TypeCode` is an object that describes how to interpret the bits in the value (that is, the value's type). Simple `TypeCode` constants for simple IDL types, such as `long` and `Object`, are built into the header files produced by the `idl2cpp` compiler. `TypeCodes` for IDL constructed types, such as `structs`, `unions`, and `typedefs`, have to be constructed. Such `TypeCodes` can be recursive because the types they describe can be recursive. Consider a `struct` consisting of a `long` and a `string`. The `TypeCode` for the `struct` contains a `TypeCode` for the `long` and a `TypeCode` for the `string`. The `idl2cpp` compiler will generate `TypeCodes` for the constructed types in an IDL file if the compiler is invoked with the `-type_code_info` option. However, if you are using the DII, you need to obtain `TypeCodes` at runtime. You can get a `TypeCode` at runtime from an interface repository (see ["Using Interface Repositories"](#)) or by asking the ORB to create one by invoking `ORB::create_struct_tc()` or `ORB::create_exception_tc()`.

If you use the `_create_request()` method, you need to put the `Any`-encapsulated target method arguments in another special object called an `NVList`. No matter how you create a `Request`, its result is encoded as an `NVList`. Everything said about arguments in this paragraph applies to results as well. `NV` stands for named value, and an `NVList` consists of a count and number of items, each of which has a name, a value, and a flag. The name is the argument name, the value is the `Any` encapsulating the argument, and the flag denotes the argument's IDL mode (for example, `in` or `out`). The result of the `Request` is represented as a single named value.

## Options for sending requests

Once you've created and populated a `Request` with arguments, a result type, and exception types, you send it to the target object. There are several ways to send a `Request`:

- The simplest is to call the `Request`'s `invoke()` method, which blocks until the reply message is received.
- More complex, but not blocking, is the `Request`'s `send_deferred()` method. This is an alternative to using threads for parallelism. For many operating systems the `send_deferred()` method is more economical than spawning a thread.
- If your motivation for using the `send_deferred()` method is to invoke multiple target objects in parallel, you can use the ORB object's `send_multiple_requests_deferred()` method instead. It takes a sequence of `Request` objects.
- Use the `Request`'s `send_oneway()` method if, and only if, the target method has been defined in IDL as `oneway`.
- You can invoke multiple `oneway` methods in parallel with the ORB's `send_multiple_requests_oneway()` method.

## Options for receiving replies

If you send a `Request` by calling its `invoke()` method, there is only one way to get the result—use the `Request` object's `env()` method to test for an exception, and if none, extract the `NamedValue` from the `Request` with its `result()` method. If you used the `send_oneway()` method then there is no result. If you used the `send_deferred()` method, you can periodically check for completion by calling the `Request`'s `poll_response()` method which returns a code indicating whether the reply has been received. If, after polling for a while, you want to block waiting for completion of a deferred send, use the `Request`'s `get_response()` method.

If you have sent `Requests` with the `send_multiple_requests_deferred()` method, you can find out if a particular `Request` is complete by invoking that `Request`'s `get_response()` method. To learn when any outstanding `Request` is complete, use the ORB's `get_next_response()` method. To do the same thing without risking blocking, use the ORB's `poll_next_response()` method.

## Steps for invoking object operations dynamically

To summarize, here are the steps that a client follows when using the DII,

- 1 Make sure the `-type_code_info` option is passed to the `idl` compiler so that type codes are generated for IDL interfaces and types. See the *VisiBroker-RT for C++ Reference Guide* for a complete description of the `idl2cpp` tool.
- 2 Obtain a generic reference to the target object you wish to use.
- 3 Create a `Request` object for the target object.
- 4 Initialize the request parameters and the result to be returned.
- 5 Invoke the request and wait for the results.
- 6 Retrieve the results.

## Location of example programs for using the DII

An example programs that illustrate the use of the DII is included in the `<install_location>/VisiBrokerRT52/examples/bank_dynamic` directory of the VisiBroker-RT for C++ distribution. This example program will be used to illustrate DII concepts in this chapter.

## Obtaining a generic object reference

When using the DII, a client program does not have to use the traditional bind mechanism to obtain a reference to the target object, because the class definition for the target object may not have been known to the client at compile time. [Code example 156](#) shows how your client program can use the `bind()` method offered by the ORB object to bind to any object by specifying its name. This method returns a generic `CORBA::Object`.

**Code example 156** Obtaining a generic object reference

```
...
void bank_client(void)
{
    VISTRY
    {
        // Get the manager Id
        PortableServer::ObjectId_var managerId =
            PortableServer::string_to_ObjectId("BankManager");

        // Locate an account manager
        CORBA::Object_var manager;

        VISIFNOT_EXCEP
            manager = orb->bind("IDL:Bank/AccountManager:1.0",
                               "/bank_agent_poa", managerId);
        VISEND_IFNOT_EXCEP
    }
    ...
}
```

## Creating and initializing a request

When your client program invokes a method on an object, a `Request` object is created to represent the method invocation. The `Request` object is written, or *marshalled*, to a buffer and sent to the object implementation. When your client program uses client stubs, this processing occurs transparently. Client programs that wish to use the DII must create and send the `Request` object themselves.

### Note

There is no constructor for this class. The `Object`'s `_request()` method or `Object`'s `_create_request()` method are used to create a `Request` object.

## Request class

The following code sample shows the `Request` class. The `target` of the request is set implicitly from the object reference used to create the `Request`. The name of the operation must be specified when the `Request` is created.

**Code example 157** Request class

```
class Request {
public:
    CORBA::Object_ptr target() const;
    const char* operation() const;
};
```

```

CORBA::NVList_ptr arguments();
CORBA::NamedValue_ptr result();
CORBA::Environment_ptr env();
void ctx(CORBA::Context_ptr ctx);
CORBA::Context_ptr ctx() const;
CORBA::Status invoke();
CORBA::Status send_oneway();
CORBA::Status send_deferred();
CORBA::Status get_response();
CORBA::Status poll_response();
...
};
};

```

## Ways to create and initialize a DII request

Once you have issued a bind to an object and obtained an object reference, you can use one of two methods for creating a `Request` object. The following code sample shows the methods offered by the `CORBA::Object` class.

**The paragraph above says two methods; the code example caption says three. Which is correct?**

**Code example 158** Three methods for creating a `Request` object

```

class Object {
...
CORBA::Request_ptr request(Identifier operation);
CORBA::Status create_request(
CORBA::Context_ptr ctx, const char *operation,
CORBA::NVList_ptr arg_list,
CORBA::NamedValue_ptr result,
CORBA::Request_ptr request,
CORBA::Flags req_flags);
CORBA::Status create_request(
CORBA::Context_ptr ctx, const char *operation,
CORBA::NVList_ptr arg_list,
CORBA::NamedValue_ptr result,
CORBA::ExceptionList_ptr eList,
CORBA::ContextList_ptr ctxList,
CORBA::Request_out request,
CORBA::Flags req_flags);
...
};

```

## Using the `create_request` method

You can use the `_create_request()` method to create a `Request` object, initialize the `Context`, the operation name, the argument list to be passed, and the result. Optionally, you can set the `ContextList` for the request, which corresponds to the attributes defined in the request's IDL. The request parameter points to the `Request` object that was created for this operation.

## Using the `_request` method

Code sample 28.4 shows the use of the `_request()` method to create a `Request` object, specifying only the operation name. After creating a float request, calls to its `add_in_arg` method add an input parameter `Account` name and its result type is initialized to be of `Object` reference type via a call to `self_return_type` method. After a call has been made, the return value is extracted with the result's call to the method `result()`. The same

steps are repeated to invoke another method on an Account Manager instance with the only difference being in-parameters and return types.

The `req`, an `Any` object is initialized with the desired account `name` and added to the request's argument list as an input argument. The last step in initializing the request is to set the `result` value to receive a `float`.

## Example of creating a Request object

A `Request` object maintains ownership of all memory associated with the operation, the arguments, and the result so you should never attempt to free these items.

### Code example 159 Creating a request object

```
...
CORBA::NamedValue_ptr result;
CORBA::Any_ptr resultAny;
CORBA::Request_var req;
CORBA::Any customer;
...
VISTRY {
    // Create request that will be sent to the manager object
    CORBA::Request_var request;

    VISIFNOT_EXCEP
        request = manager->_request("open");
    VISEND_IFNOT_EXCEP

    // Create argument to request
    CORBA::Any customer;
    customer <<= (const char *) name;
    CORBA::NVList_ptr arguments = request->arguments();
    arguments->add_value( "name", customer, CORBA::ARG_IN );

    // Set result type
    VISIFNOT_EXCEP
        request->set_return_type(CORBA::_tc_Object);
    VISEND_IFNOT_EXCEP
}
VISCATCH (CORBA::Exception, excep) {
...

```

## Setting the context for the request

Though it is not used in the example program, the `Context` object can be used to contain a list of properties, stored as `NamedValue` objects, that will be passed to the object implementation as part of the `Request`. These properties represent information that is automatically communicated to the object implementation.

### Code example 160 Context class

```
class Context {
public:
    const char *context_name() const;
    CORBA::Context_ptr parent();
    CORBA::Status create_child(const char *name,
CORBA::Context_ptr&);
    CORBA::Status set_one_value(const char *name, const
CORBA::Any&);
    CORBA::Status set_values(CORBA::NVList_ptr);
    CORBA::Status delete_values(const char *name);
    CORBA::Status get_values(
        const char *start_scope,
        CORBA::Flags,
        const char *name,
        CORBA::NVList_ptr&) const;

```

```
};
```

## Setting arguments for the request

The arguments for a Request are represented with a `NVList` object, which stores name-value pairs as `NamedValue` objects. You can use the `arguments()` method to obtain a pointer to this list. This pointer can then be used to set the names and values of each of the arguments.

### Note

Always initialize the arguments before sending a Request. Failure to do so will result in marshalling errors and may even cause the server to abort.

## Implementing a list of arguments with the NVList

This class implements a list of `NamedValue` objects that represent the arguments for a method invocation. Methods are provided for adding, removing, and querying the objects in the list.

### Code example 161 NVList class

```
class NVList {
public:
    ...
    CORBA::Long count() const;
    CORBA::NamedValue_ptr add(Flags);
    CORBA::NamedValue_ptr add_item(const char *name,
CORBA::Flags flags);
    CORBA::NamedValue_ptr add_value(
        const char *name,
        const CORBA::Any *any,
        CORBA::Flags flags);

    CORBA::NamedValue_ptr add_item_consume(char *name,
CORBA::Flags flags);
    CORBA::NamedValue_ptr add_value_consume(
        char *name,
        CORBA::Any *any,
        CORBA::Flags flags);
    CORBA::NamedValue_ptr item(CORBA::Long index);
    CORBA::Status remove(CORBA::Long index);
    ...
};
```

## Setting input and output arguments with the NamedValue Class

This class implements a name-value pair that represents both input and output arguments for a method invocation request. The `NamedValue` class is also used to represent the result of a request that is returned to the client program. The `name` property is simply a character string and the `value` property is represented by an `Any` class.

### Code example 162 NamedValue class

```
class NamedValue {
public:
    const char *name() const;
    CORBA::Any *value() const;
    CORBA::Flags flags() const;
};
```

The following table describes the methods in the `NamedValue` class.

**Table 32** NamedValue methods

Method	Description
<code>name()</code>	Returns a pointer to the name of the item that you can then use to initialize the name.
<code>value()</code>	Returns a pointer to an <code>Any</code> object representing the item's value that you can then use to initialize the value. For more information, see <a href="#">"Passing type safely with the Any class"</a> .
<code>flags()</code>	Indicates if this item is an input argument, an output argument, or both an input and output argument. If the item is both an input and output argument, you can specify a flag indicating that the ORB should make a copy of the argument and leave the caller's memory intact. Flags are:  ARG_IN ARG_OUT ARG_INOUT

## Passing type safely with the Any class

This class is used to hold an IDL-specified type so that it may be passed in a type-safe manner. Objects of this class have a pointer to a `TypeCode` that defines the contained object's type and a pointer to the contained object. Methods are provided to construct, copy, and release an object as well as initialize and query the object's value and type. In addition, streaming operators are provided to read and write the object from and to a stream.

**Code example 163** Any class

```
class Any {
public:
    ...
    CORBA_TypeCode_ptr type();
    void type(CORBA_TypeCode_ptr tc);
    const void *value() const;
    static CORBA::Any_ptr _nil();
    static CORBA::Any_ptr _duplicate(CORBA::Any *ptr);
    static void _release(CORBA::Any *ptr);
    ...
}
```

## Representing argument or attribute types with the TypeCode class

This class is used by the Interface Repository and the IDL compiler to represent the type of arguments or attributes. `TypeCode` objects are also used in a `Request` object to specify an argument's type, in conjunction with the `Any` class. `TypeCode` objects have a `kind` and `parameter list` property.

The following table shows the kinds and parameters for the `TypeCode` objects.

**Table 33** TypeCode kinds and parameters

Kind	Parameter list
<code>tk_abstract_interface</code>	<code>interface_id</code> , <code>interface_name</code>
<code>tk_alias</code>	<code>interface_id</code> , <code>alias_name</code> , <code>TypeCode</code>
<code>tk_any</code>	None
<code>tk_array</code>	<code>length</code> , <code>TypeCode</code>
<code>tk_boolean</code>	None
<code>tk_char</code>	None

Kind	Parameter list
tk_double	None
tk_enum	enum-name, enum-id1, enum-id2,... enum-idn
tk_except	interface_id, exception_name, StructMembers
tk_fixed	digits, scale
tk_float	None
tk_long	None
tk_longdouble	None
tk_longlong	None
tk_native	id, name
tk_null	None
tk_objref	interface_id
tk_octet	None
tk_Principal	None
tk_sequence	TypeCode, maxlen
tk_short	None
tk_string	maxlen-integer
tk_struct	struct-name, {member1, TypeCode1},... {membern, TypeCodeN}
tk_TypeCode	None
tk_ulong	None
tk_ulonglong	None
tk_union	union-name, switch TypeCode, {label-value1, member-name1, TypeCode1},... {label1-value1, member-name1, TypeCode1}
tk_ushort	None
tk_value	id, name, boxType
tk_value_box	id, name, typeModifier, concreteBase, members
tk_void	None
tk_wchar	None
tk_wstring	None

#### Code example 164 TypeCode class

```

class VISEXPORT CORBA_TypeCode {
public:
    ...
    // For all CORBA_TypeCode kinds
    CORBA::Boolean equal(CORBA_TypeCode_ptr tc) const;
    CORBA::Boolean equivalent(CORBA_TypeCode_ptr tc) const;
    CORBA_TypeCode_ptr get_compact_typecode() const;
    CORBA::TCKind kind() const //...
    // For tk_objref, tk_struct, tk_union, tk_enum, tk_alias and
tk_except
    virtual const char* id() const; // raises(BadKind);
    virtual const char *name() const; // raises(BadKind);
    // For tk_struct, tk union, tk_enum and tk_except
    virtual CORBA::ULong member_count() const;
    //raises((BadKind));
    virtual const char *member_name(CORBA::ULong index) const;
    // raises((BadKind, Bounds));
    // For tk_struct, tk union and tk_except
    virtual CORBA_TypeCode_ptr member_type(CORBA::ULong index)
const;
    // raises((BadKind, Bounds));
    // For tk union
    virtual CORBA::Any_ptr member_label(CORBA::ULong index)
const;
    // raises((BadKind, Bounds));
    virtual CORBA_TypeCode_ptr discriminator_type() const;

```

```

        // raises((BadKind));
virtual CORBA::Long default_index() const;
        // raises((BadKind));
// For tk_string, tk_sequence and tk_array
virtual CORBA::ULong_length() const; // raises((BadKind));
// For tk_sequence, tk_array and tk_alias
virtual CORBA_TypeCode_ptr content_type() const;
        // raises((BadKind));
// For tk_fixed
virtual CORBA::UShort fixed_digits() const;
        // raises (BadKind)
virtual CORBA::Short fixed_scale() const; // raises (BadKind)
// for tk_value
virtual CORBA::Visibility member_visibility(CORBA::ULong
index) const;
        // raises(BadKind, Bounds);
virtual CORBA::ValueModifier type_modifier() const;
        // raises(BadKind);
virtual CORBA::TypeCode_ptr concrete_base_type() const;
        // raises(BadKind);
};

```

## Sending DII requests and receiving results

The `Request` class, shown in [Code example 157](#), provides several methods for sending a request, once it has been properly initialized.

### Invoking a request

The simplest way to send a request is to call its `invoke()` method, which sends the request and waits for a response before returning to your client program. The `return_value()` method returns a pointer to an `Any` object that represents the return value.

**Code example 165** Sending a request with `invoke()`

```

...
VISTRY {
    ...
    // Create request that will be sent to the account object
    request = account->_request("balance");

    VISIFNOT_EXCEP
        // Set the result type
        request->set_return_type(CORBA::_tc_float);
    VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
        // Execute the request to the account object
        request->invoke();
    VISEND_IFNOT_EXCEP

    // Get the return balance
    CORBA::Float balance;
    VISIFNOT_EXCEP
        CORBA::Any& balance_result = request->return_value();
    VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
        balance_result >>= balance;
    VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
        // Print out the balance
        cout << "The balance in " << name << "'s account is $"
        << balance << endl;
    VISEND_IFNOT_EXCEP
}

```

```

}VISACATCH (const CORBA::Exception, e)
{
    cerr << e << endl;
    return 1;
}
    VISEND_CATCH
}
return 0;
...

```

## Sending a deferred DII request with the `send_deferred()` method

A non-blocking method, `send_deferred()`, is also provided for sending operation requests. It allows your client to send the request and then use the `poll_response()` method to determine when the response is available. The `get_response()` method blocks until a response is received. The following code shows how these methods are used.

**Code example 166** Using the `send_deferred()` and `poll_response()` methods to send a deferred DII request

```

...
VISTRY {
    // Create request that will be sent to the manager object
    CORBA::Request_var request = manager->_request("open");

    // Create argument to request
    CORBA::Any customer;
    VISIFNOT_EXCEP
        customer <<= (const char *) name;
    VISEND_IFNOT_EXCEP

    CORBA::NVLlist_ptr arguments;
    VISIFNOT_EXCEP
        arguments = request->arguments();
    VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
        arguments->add_value( "name", customer, CORBA::ARG_IN );
    VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
        // Set result type
        request->set_return_type(CORBA::_tc_Object);
    VISEND_IFNOT_EXCEP

    // Creation of a new account can take some time
    // Execute the deffered request to the manager object
    VISIFNOT_EXCEP
        request->send_deferred();
    VISEND_IFNOT_EXCEP VISIFNOT_EXCEP

    VISPortable::vsleep(1);
    while (!request->poll_response()) {
        cout << "Waiting for response..." << endl;
        VISPortable::vsleep(1);    // Wait one second between polls
    }
    VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
        request->get_response();
    VISEND_IFNOT_EXCEP

    // Get the return value
    CORBA::Object_var account;
    CORBA::Any& open_result;
    VISIFNOT_EXCEP
        open_result = request->return_value();
}

```

```

        VISEND_IFNOT_EXCEP

        VISIFNOT_EXCEP
        open_result >>= CORBA::Any::to_object(account.out());
        VISEND_IFNOT_EXCEP
    ...
}

```

## Sending an asynchronous DII request with the send\_oneway method

The `send_oneway()` method can be used to send an asynchronous request. `Oneway` requests do not involve a response being returned to the client from the object implementation.

## Sending multiple requests

A sequence of DII Request objects can be created using array of Request objects. A sequence of requests can be sent using the ORB methods `send_multiple_requests_oneway()` or `send_multiple_requests_deferred()`. If the sequence of requests is sent as oneway requests, no response is expected from the server to any of the requests.

[Code example 167](#) shows how two requests are created and then used to create a sequence of requests. The sequence is then sent using the `send_multiple_requests_deferred()` method.

**Code example 167** Sending multiple deferred requests with the `send_multiple_requests_deferred()` method

```

...
// Create request to balance
VISTRY {
    req1 = account->_request("balance");

    CORBA::NVList_ptr arguments;
    VISIFNOT_EXCEP

    // Create argument to request
    customer1 <<= (const char *) "Happy";
    arguments= req1->arguments();
    VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
    arguments->add_value("customer", customer1, CORBA::ARG_IN);
    VISEND_IFNOT_EXCEP

    // Set result
    ...
} VISCATCH(const CORBA::Exception,excep)
{
    cout << "Error while creating request" << endl;
    cout << excep << endl;
}
VISEND_CATCH
// Create request2 to slowBalance
VISTRY{
    req2 = account->_request("slowBalance");

    CORBA::NVList_ptr arguments;
    VISIFNOT_EXCEP
    // Create argument to request
    customer2 <<= (const char *) "Sleepy";
    CORBA::NVList_ptr arguments = req2->arguments();
    VISEND_IFNOT_EXCEP
}

```

```

        VISIFNOT_EXCEP
        arguments->add_value("customer", customer2, CORBA::ARG_IN);
    // Set result
    VISEND_IFNOT_EXCEP

    ...

} VISCATCH(const CORBA::Exception,excep)
{
    cout << "Error while creating request" << endl;
    cout << excep << endl;
}
VISEND_CATCH

// Create request sequence
CORBA::Request_ptr reqs[2];
reqs[0] = (CORBA::Request*) req1;
reqs[1] = (CORBA::Request*) req2;
CORBA::RequestSeq reqseq((CORBA::ULong)2, 2,
(CORBA::Request_ptr*) reqs);

// Send the request
VISTRY {
    orb->send_multiple_requests_deferred(reqseq);

    cout << "Send multiple deferred calls are made..." << endl;
}VISCATCH(const CORBA::Exception,excep)
{
    ...
}

```

## Receiving multiple requests

When a sequence of requests is sent using `send_multiple_requests_deferred()`, the `poll_next_response()` and `get_next_response()` methods are used to receive the response the server sends for each request.

The ORB method `poll_next_response()` can be used to determine if a response has been received from the server. This method returns true if there is at least one response available. This method returns false if there are no responses available.

The ORB method `get_next_response()` can be used to receive a response. If no response is available, this method will block until a response is received. If you do not wish your client program to block, use the `poll_next_response()` method to first determine when a response is available and then use the `get_next_response()` method to receive the result.

**Code example 168** ORB methods for sending multiple requests and receiving the results

```

class CORBA {
    class ORB {
        ...
        typedef sequence <Request_ptr> RequestSeq;
        void send_multiple_requests_oneway(const RequestSeq &);
        void send_multiple_requests_deferred(const RequestSeq &);
        Boolean poll_next_response();
        Status get_next_response();
        ...
    };
};

```

## Using the interface repository with the DII

The following example has built-in knowledge of a remote object's type (`Account`) and the name of one of its methods (`balance()`). An actual DII application would get that information from an outside source—for example, a user and then use the interface repository (IR) (see [“Using Interface Repositories”](#)) to obtain the parameters of an operation.

The example:

- Binds to the `Bank_Manager` `AccountManager` object.
- Builds an operation list.
- Creates argument and result components. Note that the `balance()` method does not return an exception.
- Invokes the `Request`, extracts and prints the result.

### Code example 169 Using DII

```
#include <vxWorks.h>
#include "corba.h"
#include "vport.h"

extern CORBA::ORB_var orb;

static void bank_client(void);

void start_bank_client(void)
{
    char *    taskName = "CLIENT";
    int      Prio = 100;
    int      option = VX_FP_TASK;
    int      stackSize = 20000;

    taskSpawn(taskName,
              Prio,
              option,
              stackSize,
              (FUNCPTR)bank_client,
              0,0,0,0,0,0,0,0,0,0);
}

void bank_client(void)
{
    VISTRY
    {
        // Get the manager Id
        PortableServer::ObjectId_var managerId =
            PortableServer::string_to_ObjectId("BankManager");

        // Locate an account manager
        CORBA::Object_var manager;

        VISIFNOT_EXCEP
            manager = orb->bind("IDL:Bank/AccountManager:1.0",
                               "/ bank_agent_poa", managerId);
        VISEND_IFNOT_EXCEP

        // Set the account name
        const char* name = "Jack B. Quick";

        // Create request that will be sent to the manager object
        CORBA::Request_var request;

        VISIFNOT_EXCEP
            request = manager->_request("open");
        VISEND_IFNOT_EXCEP

        // Create argument to request
        CORBA::Any customer;
```

```

customer <<= (const char *) name;
CORBA::NVList_ptr arguments = request->arguments();
arguments->add_value( "name", customer, CORBA::ARG_IN );

// Set result type VISIFNOT_EXCEP
request->set_return_type(CORBA::_tc_Object);
VISEND_IFNOT_EXCEP

// Creation of a new account can take some time
// Execute the deffered request to the manager object
VISIFNOT_EXCEP
{
    request->send_deferred();

    VISPortable::vsleep(1);
}
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
{
    while (!request->poll_response())
    {
        cout << "Waiting for response..." << endl;
        VISPortable::vsleep(1); // Wait one second between polls
    }
}
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
    request->get_response();
VISEND_IFNOT_EXCEP

// Get the return value CORBA::Object_var account;

VISIFNOT_EXCEP
{
    CORBA::Any& open_result = request->return_value();

    open_result >>= CORBA::Any::to_object(account.out());
}
VISEND_IFNOT_EXCEP

// Create request that will be sent to the account object
VISIFNOT_EXCEP
    request = account->_request("balance");
VISEND_IFNOT_EXCEP

// Set the result type
VISIFNOT_EXCEP
    request->set_return_type(CORBA::_tc_float);
VISEND_IFNOT_EXCEP

// Execute the request to the account object
VISIFNOT_EXCEP
    request->invoke();
VISEND_IFNOT_EXCEP

// Get the return balance
CORBA::Float balance;

VISIFNOT_EXCEP
    CORBA::Any& balance_result = request->return_value();

    // Print out the balance VISIFNOT_EXCEP
    {
        balance_result >>= balance;
        cout << "The balance in " << name << "'s account is $"
            << balance << endl;
    }
    VISEND_IFNOT_EXCEP
VISEND_IFNOT_EXCEP
}

```

```
VISCATCH(CORBA::Exception, e)
{
    cerr << e << endl;
}
VISEND_CATCH
return;
}
```



# Using the Dynamic Skeleton Interface

*This chapter describes how object servers can dynamically create object implementations at run time to service client requests.*

## NOTE

The library `liborb_dyn.o` is required when building a VisiBrokerRT60 application to support the use of Dynamic Invocation Interface. For a description of all the libraries provided by the VisiBroker-RT for C++ product, see [“Step 6: Integrating VisiBroker-RT with Tornado/VxWorks”](#).

The Dynamic Skeleton Interface (DSI) is not supported as part of the “minimum CORBA” version of VisiBroker-RT for C++ (i.e. `liborb_min.o`).

The “minimum CORBA” OMG specification identifies dynamic functionality which should be excluded from an ORB, in an effort to reduce the ORB footprint.

For details, see the minimum CORBA specification document, OMG document number orbos/ 98-08-04. This document is available for download using the URL <ftp://ftp.omg.org/pub/docs/orbos/98-08-04.pdf>.

## What is the Dynamic Skeleton Interface?

The Dynamic Skeleton Interface (DSI) provides a mechanism for creating an object implementation that does not inherit from a generated skeleton interface. Normally, an object implementation is derived from a skeleton class generated by the `idl2cpp` compiler. The DSI allows an object to register itself with the ORB, receive operation requests from a client, process the requests, and return the results to the client without inheriting from a skeleton class generated by the `idl2cpp` compiler.

### Note

From the perspective of a client program, an object implemented with the DSI behaves just like any other ORB object. Clients do not need to provide any special handling to communicate with an object implementation that uses the DSI.

The ORB presents client operation requests to a DSI object implementation by calling the object’s `invoke()` method and passing it a `ServerRequest` object. The object implementation is responsible for determining the operation being requested, interpreting the arguments associated with the request, invoking the appropriate internal method or methods to fulfill the request, and returning the appropriate values.

Implementing objects with the DSI requires more manual programming activity than using the normal language mapping provided by object skeletons. Nevertheless, an object implemented with the DSI can be very useful in providing inter-protocol bridging.

## Steps for creating object implementations dynamically

To create object implementations dynamically using the DSI, follow these steps:

- 1 Use the `-type_code_info` flag when compiling your IDL.
- 2 Define the macro `_VIS_INCLUDE_DSI` in your DSI server implementation. Note this in the file `<install_location>/VisiBrokerRT60/examples/bank_dynamic/server.cpp`.
- 3 Design your object implementation so that it is derived from the `PortableServer::DynamicImplementation` abstract class instead of deriving your object implementation from a skeleton class.
- 4 Declare and implement the `invoke()` method, which the ORB will use to dispatch client requests to your object.
- 5 Register your object implementation (POA servant) with the POA manager as the default servant.

### Location of an example program for using the DSI

An example program that illustrates the use of the DSI is included in the `<install_location>/VisiBrokerRT60/examples/basic/bank_dynamic` directory of the VisiBroker-RT for C++ distribution. This example is used to illustrate DSI concepts in this chapter. The `Bank.idl` file, shown in [IDL sample 24](#), illustrates the interfaces implemented in this example.

**IDL sample 24** Bank.idl file used in the DSI example

```
// Bank.idl module Bank {
    interface Account {
        float balance();
    };
    interface AccountManager {
        Account open(in string name);
    };
};
```

## Extending the DynamicImplementation class

To use the DSI, object implementations should be derived from the `DynamicImplementation` base class shown below. This class offers several constructors and the `invoke()` method, which you must implement.

**Code example 170** DynamicImplementation base class

```
class PortableServer::DynamicImplementation : public virtual
PortableServer::ServantBase {
public:
    virtual void invoke(PortableServer::ServerRequest_ptr
request) = 0;
    ...
};
```

## Example of designing objects for dynamic requests

[Code example 171](#) shows the declaration of the `AccountImpl` class that is to be implemented with the DSI. It is derived from the `DynamicImplementation` class, which declares the `invoke()` method. The ORB will call the `invoke()` method to pass client operation requests to the implementation in the form of `ServerRequest` objects.

Also note the `Account` class constructor and `_primary_interface()` function are shown in [Code example 171](#).

### Code example 171 AccountImpl class from the dynamic example

```
// Implementation of Account default servant
class AccountImpl : public PortableServer::DynamicImplementation
{
public:
    AccountImpl(PortableServer::Current_ptr current,
                PortableServer::POA_ptr poa) :
        _poa_current(PortableServer::Current::_duplicate(current)),
        _poa(poa)
    {}

    CORBA::Object_ptr get(const char *name)
    {
        CORBA::Float balance;

        // Check if account exists
        if (!_registry.get(name, balance))
        {

            // simulate delay while creating new account
            VISPortable::vsleep(3);

            // Make up the account's balance, between 0 and $1000
            balance = abs(rand()) % 100000 / 100.0;
            // Print out the new account
            cout << "Created " << name << "'s account: " << balance <<
            endl

            _registry.put(name, balance);
        }

        // Return object reference
        PortableServer::ObjectId_var accountId =
            PortableServer::string_to_ObjectId(name);

        return _poa->create_reference_with_id(accountId, "IDL:Bank/
Account:1.0");
    }

private:
    AccountRegistry _registry;
    PortableServer::POA_ptr _poa;
    PortableServer::Current_var _poa_current;

    CORBA::RepositoryId _primary_interface(
        const PortableServer::ObjectId& oid,
        PortableServer::POA_ptr poa)
    {
        {
            return CORBA::string_dup((const char *)"IDL:Bank/
Account:1.0");
        }
    };

    void invoke(CORBA::ServerRequest_ptr request)
    {
        // Get the account name from the object id
        PortableServer::ObjectId_var oid =
        _poa_current->get_object_id();
    }
};
```

```

CORBA::String_var name;
VISTRY
{
    name = PortableServer::ObjectId_to_string(oid);
}
VISCATCH (CORBA::Exception, e)
{
    VISTHROW(CORBA::OBJECT_NOT_EXIST());
}
VISEND_CATCH

// Ensure that the operation name is correct
if (strcmp(request->operation(), "balance") != 0)
{
    VISTHROW(CORBA::BAD_OPERATION());
}

// Find out balance and fill out the result
CORBA::NVList_ptr params = new CORBA::NVList(0);
request->arguments(params);

CORBA::Float balance;
if (!_registry.get(name, balance))
    VISTHROW(CORBA::OBJECT_NOT_EXIST());

CORBA::Any result; result <=& balance;
request->set_result(result);

cout << "Checked " << name << "'s balance: " << balance <<
endl;
}
};

```

The code below shows the implementation of the `AccountManagerImpl` class that needs to be implemented with the DSI. It is also derived from the `DynamicImplementation` class, which declares the `invoke()` method. The ORB will call the `invoke()` method to pass client operation requests to the implementation in the form of `ServerRequest` objects.

**Code example 172** `AccountManagerImpl` class from the dynamic example

```

// Implementation of manager default servant
class AccountManagerImpl : public
PortableServer::DynamicImplementation
{
public:
    AccountManagerImpl(AccountImpl* accounts) { _accounts =
accounts; }

    CORBA::Object_ptr open(const char* name)
    {
        return _accounts->get(name);
    }

private:
    AccountImpl* _accounts;

    CORBA::RepositoryId _primary_interface(const
PortableServer::ObjectId& oid, PortableServer::POA_ptr poa)
    {
        return CORBA::string_dup((const char *)"IDL:Bank/
AccountManager:1.0");
    }

    void invoke(CORBA::ServerRequest_ptr request)
    {
        // Ensure that the operation name is correct
        if (strcmp(request->operation(), "open") != 0)
            VISTHROW(CORBA::BAD_OPERATION());

        // Fetch the input parameter
        char *name = NULL;
    }
};

```

```

VISTRY
{
    CORBA::NVList_ptr params = new CORBA::NVList(1);
    CORBA::Any any;
    any <<= (const char*) "";
    params->add_value("name", any, CORBA::ARG_IN);
    request->arguments(params);
    *(params->item(0)->value()) >>= name;
}
VISCATCH (CORBA::Exception, e)
{
    VISTHROW(CORBA::BAD_PARAM());
}
VISEND_CATCH

// Invoke the actual implementation and fill out the result
CORBA::Object_var account = open(name);
CORBA::Any result;
result <<= account;
request->set_result(result);
}
};

```

## Specifying repository ids

- 1 The `_primary_interface()` method should be implemented to return supported repository identifiers. To determine the correct repository identifier to specify, start with the IDL interface name of an object and use the following steps:
- 2 Replace all non-leading instances of the delimiter scope resolution operator (`::`) with a slash (`/`).
- 3 Add "IDL:" to the beginning of the string.
- 4 Add ":1.0" to the end of the string.

For example, [Code example 173](#) shows an IDL interface name and [Code example 174](#) shows the resulting repository identifier string.

**Code example 173** IDL interfacename

```
Bank::AccountManager
```

**Code example 174** Resulting repository identifier

```
IDL:Bank/AccountManager:1.0
```

## Looking at the ServerRequest class

A `ServerRequest` object is passed as a parameter to an object implementation's `invoke()` method. The `ServerRequest` object represents the operation request and provides methods for obtaining the name of the requested operation, the parameter list, and the context. It also provides methods for setting the result to be returned to the caller and for reflecting exceptions.

**Code example 175** ServerRequest base class

```

class CORBA::ServerRequest {
public:
    const char* op_name() const { return _operation; }
    void params(CORBA::NVList_ptr);
    void result(CORBA::Any_ptr);
    void exception(CORBA::Any_ptr exception);
    ...
    CORBA::Context_ptr ctx() {
        ...
    }
}

```

```

// POA spec methods
const char *operation() const { return _operation; }
void arguments(CORBA::NVList_ptr param) { params(param); }
void set_result(const CORBA::Any& a) { result(new
                                     CORBA::Any(a));
}
void set_exception(const CORBA::Any& a) {
    exception(new CORBA::Any(a));
}
};

```

All arguments passed into the `arguments()`, `set_result()`, or `set_exception()` methods are thereafter owned by the ORB. The memory for these arguments will be released by the ORB—you should not release them.

### Note

The following methods have been deprecated:

- `op_name`
- `params`
- `result`
- `exception`

## Implementing the Account object

The `Account` interface in declares only one method, so the processing done by the `AccountImpl` class' `invoke()` method is fairly straightforward.

The `invoke()` method first checks to see if the requested operation has the name "balance." If the name does not match, a `BAD_OPERATION` exception is raised. If the `Account` object were to offer more than one method, the `invoke()` method would need to check for all possible operation names and use the appropriate internal methods to process the operation request.

Since the `balance()` method does not accept any parameters, there is no parameter list associated with its operation request. The `balance()` method is simply invoked and the result is packaged in an `Any` object that is returned to the caller, using the `ServerRequest` object's `set_result()` method.

## Implementing the AccountManager object

Like the `Account` object, the `AccountManager` interface also declares one method. However, the `AccountManagerImpl` object's `open()` method does accept an account name parameter. This makes the processing done by the `invoke()` method a little more complicated. [Code example 172](#) shows the implementation of the `AccountManagerImpl` object's `invoke()` method.

The method first checks to see that the requested operation has the name "open." If the name does not match, a `BAD_OPERATION` exception is raised. If the `AccountManager` object were to offer more than one method, its `invoke()` method would need to check for all possible operation names and use the appropriate internal methods to process the operation request.

### Processing input parameters

Here are the steps the `AccountManagerImpl` object's `invoke()` method uses to process the operation request's input parameters.

- 1 Create an `NVList` to hold the parameter list for the operation.

- 2 Create `Any` objects for each expected parameter and add them to the `NVList`, setting their `TypeCode` and parameter type (`ARG_IN`, `ARG_OUT`, or `ARG_INOUT`).
- 3 Invoke the `ServerRequest` object's `arguments()` method, passing the `NVList`, to update the values for all the parameters in the list.

The `open()` method expects an account name parameter; therefore, an `NVList` object is created to hold the parameters contained in the `ServerRequest`. The `NVList` class implements a parameter list containing one or more `NamedValue` objects. The `NVList` and `NamedValue` classes are described in ["Using the Dynamic Invocation Interface"](#).

An `Any` object is created to hold the account name. This `Any` is then added to `NVList` with the argument's name set to "name" and the parameter type set to `ARG_IN`.

Once the `NVList` has been initialized, the `ServerRequest` object's `arguments()` method is invoked to obtain the values of all of the parameters in the list.

### Note

After invoking the `arguments()` method, the `NVList` will be owned by the ORB. This means that if an object implementation modifies an `ARG_INOUT` parameter in the `NVList`, the change will automatically be apparent to the ORB. This `NVList` should not be released by the caller.

An alternative to constructing the `NVList` for the input arguments is to use the ORB object's `create_operation_list()` method. This method accepts an `OperationDef` and returns an `NVList` object, completely initialized with all the necessary `Any` objects. The appropriate `OperationDef` object may be obtained from the interface repository, described in ["Using Interface Repositories"](#).

## Setting the return value

After invoking the `ServerRequest` object's `arguments()` method, the value of the `name` parameter can be extracted and used to create a new `Account` object. An `Any` object is created to hold the newly created `Account` object, which is returned to the caller by invoking the `ServerRequest` object's `set_result()` method.

## Server implementation

The implementation of the `main` routine, shown below, is almost identical to the original example introduced in ["Developing an Example Application with VisiBroker-RT for C++"](#).

### Code example 176 Server implementation

```
void bank_server()
{
    PortableServer::POA_var rootPOA;

    VISTRY
    {
        //get a reference to the root POA
        CORBA::Object_var obj =
            orb->resolve_initial_references("RootPOA");

        VISIFNOT_EXCEP
            rootPOA = PortableServer::POA::_narrow(obj);
        VISEND_IFNOT_EXCEP

        // Get the POA Manager
```

```

VISIFNOT_EXCEP
    PortableServer::POAManager_var poaManager =
        rootPOA->the_POAManager();
VISEND_IFNOT_EXCEP

// Create the account POA with the right policies
CORBA::PolicyList accountPolicies;
    accountPolicies.length(3);

VISIFNOT_EXCEP
accountPolicies[(CORBA::ULong)0] = rootPOA->
    create_servant_retention_policy(PortableServer::NON_RETAIN
);
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
    accountPolicies[(CORBA::ULong)1] =
        rootPOA->create_request_processing_policy(
PortableServer::USE_DEFAULT_SERVANT);
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
accountPolicies[(CORBA::ULong)2] =
    rootPOA->create_id_uniqueness_policy(
        PortableServer::MULTIPLE_ID); VISEND_IFNOT_EXCEP

PortableServer::POA_var accountPOA;

VISIFNOT_EXCEP
    accountPOA = rootPOA->create_POA("bank_account_poa",
        poaManager, accountPolicies);
VISEND_IFNOT_EXCEP

// Create the account default servant
PortableServer::Current_var current;

VISIFNOT_EXCEP
    current = PortableServer::Current::_instance();
VISEND_IFNOT_EXCEP

AccountImpl *accountServant = new AccountImpl(current,
        accountPOA);

VISIFNOT_EXCEP
    accountPOA->set_servant(accountServant);
VISEND_IFNOT_EXCEP

PortableServer::POA_var managerPOA;

VISIFNOT_EXCEP
{
    // Create the manager POA with the right policies
    CORBA::PolicyList managerPolicies;
    managerPolicies.length(3);

    VISIFNOT_EXCEP
        managerPolicies[(CORBA::ULong)0] =
rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);
VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
        managerPolicies[(CORBA::ULong)1] =
            rootPOA->create_request_processing_policy
                (PortableServer::USE_DEFAULT_SERVANT);
VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
        managerPolicies[(CORBA::ULong)2] = rootPOA->
            create_id_uniqueness_policy(PortableServer::MULTIPLE_ID);
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP

```

```

        managerPOA = rootPOA->create_POA("bank_agent_poa",
poaManager, managerPolicies);
        VISEND_IFNOT_EXCEP
    }
    VISEND_IFNOT_EXCEP

    // Create the manager default servant
    AccountManagerImpl *managerServant = new
        AccountManagerImpl(accountServant);

    VISIFNOT_EXCEP
        managerPOA->set_servant(managerServant);
    VISEND_IFNOT_EXCEP

    // Activate the POA Manager
    VISIFNOT_EXCEP
        poaManager->activate();
    VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
        cout << "AccountManager is ready" << endl;
    VISEND_IFNOT_EXCEP

    }
    VISCATCH(CORBA::Exception, e)
    {
        cerr << e << endl;
        return 0;
    }
    VISEND_CATCH

    return;
}

```

The DSI implementation is instantiated as a default servant and the POA should be created with the support of corresponding policies. For more information see [“Using POAs”](#).



# Using the Dynamically Managed Types

This chapter describes the *DynAny* feature of VisiBroker-RT for C++, which allows you to construct and interpret data types at runtime.

## NOTE

The library `liborb_dyn.o` is required when building a VisiBrokerRT60 application to support the use of Dynamic Invocation Interface. For a description of all the libraries provided by the VisiBroker-RT for C++ product, see [“Step 6: Integrating VisiBroker-RT with Tornado/VxWorks”](#).



The *DynAny* interface is not supported as part of the “minimum CORBA” version of VisiBroker-RT for C++ (i.e. `liborb_min.o`).

The “minimum CORBA” OMG specification identifies dynamic functionality which should be excluded from an ORB, in an effort to reduce the ORB footprint.

For details, see the minimum CORBA specification document, OMG document number orbos/98-08-04. This document is available for download using the URL <ftp://ftp.omg.org/pub/docs/orbos/98-08-04.pdf>.

## Overview

The *DynAny* interface provides a way to dynamically create basic and constructed data types at runtime. It also allows information to be interpreted and extracted from an *Any* object, even if the type it contains was not known to the server at compile-time. The use of the *DynAny* interface enables you to build powerful client and server applications that create and interpret data types at runtime.

Example client and server applications that illustrate the use of *DynAny* are included as part of the VisiBroker distribution. The `examples` are found in `<install_location>/VisiBrokerRT60/examples/dynany` directory. These example programs will be used to illustrate *DynAny* concepts in this chapter.

## DynAny types

A *DynAny* object has an associated value that may either be a basic data type (such as `boolean`, `int`, or `float`) or a constructed data type. The *DynAny* interface, described in detail in the VisiBroker-RT for C++ *Reference Guide*, provides methods for determining the type of the contained data as well as for setting and extracting the value of primitive data types.

Constructed data types are represented by the following interfaces, which are all derived from *DynAny*. Each of these interfaces provides its own set of methods that are appropriate for setting and extracting the values it contains.

**Table 34** Interfaces derived from `DynAny` that represent constructed data types

Interface	TypeCode	Description
<code>DynArray</code>	<code>_tk_array</code>	An array of values with the same data type that has a fixed number of elements.
<code>DynEnum</code>	<code>_tk_enum</code>	A single enumeration value.
<code>DynFixed</code>	<code>_tk_fixed</code>	Not supported.
<code>DynSequence</code>	<code>_tk_sequence</code>	A sequence of values with the same data type. The number of elements may be increased or decreased.
<code>DynStruct</code>	<code>_tk_struct</code>	A structure.
<code>DynUnion</code>	<code>_tk_union</code>	A union.
<code>DynValue</code>	<code>_tk_value</code>	Not supported.

## Usage restrictions

A `DynAny` object may only be used locally by the ORB instance which created it. Any attempt to use a `DynAny` object as a parameter on an operation request for a bound object or to externalize it using the `ORB::object_to_string` method will cause a `MARSHAL` exception to be raised.

Furthermore, any attempt to use a `DynAny` object as a parameter on DII request will cause a `NO_IMPLEMENT` exception to be raised.

This version does not support the long double and fixed types as specified in CORBA 2.3.

## Creating a `DynAny`

A `DynAny` object is created by invoking an operation on a `DynAnyFactory` object. First obtain a reference to the `DynAnyFactory` object, and then use that object to create the new `DynAny` object.

```

CORBA::Object var obj =
orb->resolve_initial_references("DynAnyFactory");
DynamicAny::DynAnyFactory var factory =
DynamicAny::DynAnyFactory::_narrow(obj);

// Create Dynamic struct

DynamicAny::DynAny var dynany =
factory->create_dyn_any_from_type_code(
    PrINTER::_tc_StructType);

DynamicAny::DynStruct var info =
DynamicAny::DynStruct::_narrow(dynany);

info->set_members(seq);

CORBA::Any var any = info->to_any();

```

## Initializing and accessing the value in a `DynAny`

The `DynAny::insert_<type>` methods allow you to initialize a `DynAny` object with a variety of basic data types, where `<type>` is `boolean`, `octet`, `char`, and so on. Any attempt to insert a type that does not match the `TypeCode` defined for the `DynAny` will cause an `TypeMismatch` exception to be raised.

The `DynAny::get_<type>` methods allow you to access the value contained in a `DynAny` object, where `<type>` is `boolean`, `octet`, `char`, and so on. Any attempt to access a value from a `DynAny` component which does not match the `TypeCode` defined for the `DynAny` will cause a `TypeMismatch` exception to be raised.

The `DynAny` interface also provide methods for copying, assigning, and converting to or from an `Any` object. The sample programs, described later in this chapter, provide examples of how to use some of these methods. The `VisiBroker-RT for C++ Reference Guide` provides a complete description of these methods.

## Constructed data types

The following types are derived from the `DynAny` interface and are used to represent constructed data types. These interfaces, and the methods they offer, all described in the `VisiBroker-RT for C++ Reference Guide`.

### Traversing the components in a constructed data type

Several of the interfaces that are derived from `DynAny` actually contain multiple components. The `DynAny` interface provides methods that allow you to iterate through these components. The `DynAny`-derived objects that contain multiple components maintain a pointer to the current component.

DynAny method	Description
<code>rewind</code>	Resets the current component pointer to the first component. Has no effect if the object contains only one component.
<code>next</code>	Advances the pointer to the next component. If there are no more components or if the object contains only one component, <code>false</code> is returned.
<code>current_component</code>	Returns a <code>DynAny</code> object, which may be narrowed to the appropriate type, based on the component's <code>TypeCode</code> .
<code>seek</code>	Sets the current component pointer to the component with the specified, zero-based index. Returns <code>false</code> if there is no component at the specified index. Sets the current component pointer to <code>-1</code> (no component) if specified with a negative index.

## DynEnum

This interface represents a single enumeration constant. Methods are provided for setting and obtaining the value as a string or as an integral value.

## DynStruct

This interface represents a dynamically constructed `struct` type. The members of the structure can be retrieved or set using a sequence of `NameValuePair` objects. Each `NameValuePair` object contains the member's name and an `Any` containing the member's Type and value.

You may use the `rewind`, `next`, `current_component`, and `seek` methods to traverse the members in the structure. Methods are provided for setting and obtaining the structure's members.

## DynUnion

This interface represents a union and contains two components. The first component represents the discriminator and the second represents the member value.

You may use the `rewind`, `next`, `current_component`, and `seek` methods to traverse the components. Methods are provided for setting and obtaining the union's discriminator and member value.

## DynSequence and DynArray

A `DynSequence` or `DynArray` represents a sequence of basic or constructed data types without the need of generating a separate `DynAny` object for each component in the sequence or array. The number of components in a `DynSequence` may be changed, while the number of components in a `DynArray` is fixed.

You may use the `rewind`, `next`, `current_component`, and `seek` methods to traverse the members in a `DynArray` or `DynSequence`.

## Example IDL

The following code sample shows the IDL used in the example client and server applications. The `StructType` structure contains two basic data types and an enumeration value. The `PrinterManager` interface is used to display the contents of an `Any` without any static information about the data type it contains.

**Code example 177** IDL for the `DynAny` example clients

```
// Printer.idl

module Printer {
    enum EnumType {first, second, third, fourth};
    struct StructType {
        string str;
        EnumType e;
        float fl;
    };
    interface PrinterManager {
        void printAny(in any info);
        oneway void shutdown();
    };
};
```

## Example client application

[Code example 178](#) shows a client application that can be found in the `dynany` directory of the `examples` directory in the `VisiBroker-RT for C++` distribution. The path is `<install_location>/VisiBrokerRT60/examples/dynany`. The client application uses the `DynStruct` interface to dynamically create a `StructType` structure.

The `DynStruct` interface uses a sequence of `NameValuePair` objects to represent the structure members and their corresponding values. Each name-value pair consists of a string containing the structure member's name and an `Any` object containing the structure member's value.

After initializing the ORB in the usual manner and binding to an `PrintManager` object, the client performs these steps:

- 1 Create an empty `DynStruct` with the appropriate type.
- 2 Create a sequence of `NameValuePair` objects that will contain the structure members.
- 3 Create and initialize `Any` objects for each of the structure member's values.
- 4 Initialize each `NameValuePair` with the appropriate member name and value.
- 5 Initialize the `DynStruct` object with the `NameValuePair` sequence.
- 6 Invoke the `PrinterManager::printAny` method, passing the `DynStruct` converted to a regular `Any`.

### Note

You must use the `DynAny::to_any` method to convert a `DynAny` object, or one of its derived types, to an `Any` before passing it as a parameter on an operation request.

### Code example 178 Example client application that uses `DynStruct`

```
void client(void)
{
    VISTRY
    {
        CORBA::Object_var obj =
            orb->resolve_initial_references("DynAnyFactory");

        DynamicAny::DynAnyFactory_var factory;

        VISIFNOT_EXCEP
            factory = DynamicAny::DynAnyFactory::_narrow(obj);
        VISEND_IFNOT_EXCEP

        DynamicAny::NameValuePairSeq seq(3);
        seq.length(3);

        CORBA::Any strAny, enumAny, floatAny;

        strAny <<= CORBA::Any::from_string((const char*)"String", 0,
            OUL);
        enumAny <<= Printer::second;
        floatAny <<= (CORBA::Float)864.50;

        CORBA::NameValuePair nvpairs[3];

        nvpairs[0].id = CORBA::string_dup("str");
        nvpairs[0].value = strAny;

        nvpairs[1].id = CORBA::string_dup("e");
        nvpairs[1].value = enumAny;

        nvpairs[2].id = CORBA::string_dup("fl");
        nvpairs[2].value = floatAny;

        seq[0] = nvpairs[0];
        seq[1] = nvpairs[1];
        seq[2] = nvpairs[2];

        // Create Dynamic struct
        DynamicAny::DynAny_var dynany;

        VISIFNOT_EXCEP
            dynany = factory->
                create_dyn_any_from_type_code(Printer::_tc_StructType);
        VISEND_IFNOT_EXCEP

        DynamicAny::DynStruct_var info;

        VISIFNOT_EXCEP
```

```

        info = DynamicAny::DynStruct::_narrow(dynany);
    VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
        info->set_members(seq);
    VISEND_IFNOT_EXCEP

    CORBA::Any_var any;

    VISIFNOT_EXCEP
        any = info->to_any();
    VISEND_IFNOT_EXCEP

    // now bind to the server and pass the constructed CORBA::Any

    // Get the manager Id
    PortableServer::ObjectId_var managerId;

    VISIFNOT_EXCEP
        managerId =
            PortableServer::string_to_ObjectId("PrinterManager");
    VISEND_IFNOT_EXCEP

    // Locate an account manager. Give the full POA name and the
    servant ID.
    Printer::PrinterManager_var manager;

    VISIFNOT_EXCEP
        manager = Printer::PrinterManager::_bind("/serverPoa",
managerId);
    VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
        manager->printAny(*any);
    VISEND_IFNOT_EXCEP

    VISIFNOT_EXCEP
        manager->shutdown();
    VISEND_IFNOT_EXCEP

    }
    VISCATCH(CORBA::Exception, e)
    {
        cerr << "Caught Exception" << e << endl;
    }
    VISEND_CATCH

    return;
}}

```

## Example server application

The following code sample shows a server application that can be found in the `dynany` directory of the `examples` directory in the VisiBroker-RT for C++ distribution. The server application performs these steps.

- 1 Initialize the ORB.
- 2 Create the policies for the POA.
- 3 Create a `PrintManager` object.
- 4 Export the `PrintManager` object.
- 5 Print a message and wait for incoming operation requests.

**Code example 179** Example server application

```

...
void server()
{

```

```

PortableServer::POA_var rootPOA;

VISTRY
{
    // get a reference to the root POA
    CORBA::Object_var obj =
        orb->resolve_initial_references("RootPOA");

    VISIFNOT_EXCEP
        rootPOA = PortableServer::POA::_narrow(obj);
    VISEND_IFNOT_EXCEP

    CORBA::Boolean Verbose = 0UL;

    CORBA::PolicyList policies;
    policies.length(1);
    VISIFNOT_EXCEP
        policies[(CORBA::ULong)0] = rootPOA->
create_lifespan_policy(PortableServer::PERSISTENT);
    VISEND_IFNOT_EXCEP

    PortableServer::POAManager_var poa_manager;

    VISIFNOT_EXCEP
        poa_manager = rootPOA->the_POAManager();
    VISEND_IFNOT_EXCEP

    // Create serverPOA with the right policies
    PortableServer::POA_var serverPOA;

    VISIFNOT_EXCEP
        serverPOA = rootPOA->create_POA("serverPoa", poa_manager,
policies);
    VISEND_IFNOT_EXCEP

    // Resolve Dynamic Any Factory CORBA::Object_var fact_obj;

    VISIFNOT_EXCEP
        fact_obj =
            orb->resolve_initial_references("DynAnyFactory");
    VISEND_IFNOT_EXCEP

    DynamicAny::DynAnyFactory_var factory;

    VISIFNOT_EXCEP
        factory = DynamicAny::DynAnyFactory::_narrow(fact_obj);
    VISEND_IFNOT_EXCEP

    PortableServer::ObjectId_var managerId;

    VISIFNOT_EXCEP
        managerId =
            PortableServer::string_to_ObjectId("PrinterManager");
    VISEND_IFNOT_EXCEP

    // Create the printer manager object.
    PrinterManagerImpl *manager;

    VISIFNOT_EXCEP
        manager = new PrinterManagerImpl(orb, factory, serverPOA,
managerId);
    VISEND_IFNOT_EXCEP

    // Export the newly create object.
    VISIFNOT_EXCEP
        serverPOA->activate_object_with_id(managerId,manager);
    VISEND_IFNOT_EXCEP

    // Activate the POA Manager
    VISIFNOT_EXCEP
        poa_manager->activate();
    VISEND_IFNOT_EXCEP

```

```

CORBA::Object_var reference;
VISIFNOT_EXCEP
    reference = serverPOA->servant_to_reference(manager);
VISEND_IFNOT_EXCEP

VISIFNOT_EXCEP
    cout << reference << " is ready" << endl;
VISEND_IFNOT_EXCEP

}
VISCATCH(CORBA::Exception, e)
{
    cerr << e << endl;
    taskSuspend(0);
}
VISEND_CATCH

return;
}

```

The following code sample shows how the `PrinterManager` implementation follows these steps in using a `DynAny` to process the `Any` object, without any compile-time knowledge of the type the `Any` contains.

- 1 Create a `DynAny` object, initializing it with the received `Any`.
- 2 Perform a `switch` on the `DynAny` object's type.
- 3 If the `DynAny` contains a basic data type, simply print out the value.
- 4 If the `DynAny` contains an `Any` type, create a `DynAny` for it, determine it's contents, and then print out the value.
- 5 If the `DynAny` contains an `enum`, create a `DynEnum` for it and then print out the string value.
- 6 If the `DynAny` contains a union, create a `DynUnion` for it and then print out the union's discriminator and the member.
- 7 If the `DynAny` contains a `struct`, array, or sequence, traverse through the contained components and print out each value.

#### Code example 180 The PrinterManager Implementation

```

#define _VIS_VXWORKS_LONG_LONG_IOSTREAMS

// PrinterManager Implementation
class PrinterManagerImpl : public POA_Printer::PrinterManager
{
public:
    PrinterManagerImpl(CORBA::ORB_ptr orb,
        DynamicAny::DynAnyFactory_ptr dynfactory,
        PortableServer::POA_ptr poa,
        PortableServer::ObjectId_ptr oid)
        : _orb(orb)
    {
        _factory =
            DynamicAny::DynAnyFactory::_duplicate(dynfactory);
        _poa = PortableServer::POA::_duplicate(poa);
        _oid = PortableServer::ObjectId::_duplicate(oid);
    }

    void printAny(const CORBA::Any& info)
    {
        VISTRY
        {
            // Create a DynAny object
            DynamicAny::DynAny_var dynAny =
                _factory->create_dyn_any(info);
            display(dynAny);
        }
        VISCATCH (CORBA::Exception, e)
        {

```

```

        cout << "Unable to create Dynamic Any from factory" <<
endl;
    }
    VISEND_CATCH
}

void shutdown()
{
    VISTRY
    {
        _poa->deactivate_object(_oid);

        cout << "Server shutting down..." << endl;
    }
VISCATCH (CORBA::Exception, e)
{
    cerr << e << endl;
    return 0;
}
VISEND_CATCH
}

void display(DynamicAny::DynAny_ptr value)
{
    CORBA::TypeCode_var type = value->type();
    while (type->kind() == CORBA::tk_alias)
        type = type->content_type();

    switch(type->kind())
    {
        case CORBA::tk_null:
        case CORBA::tk_void:
            break;
        case CORBA::tk_short:
            {
                cout << value->get_short() << endl;
                break;
            }
        case CORBA::tk_ushort:
            {
                cout << value->get_ushort() << endl;
                break;
            }
        case CORBA::tk_long:
            {
                cout << value->get_long() << endl;
                break;
            }
        case CORBA::tk_ulong:
            {
                cout << value->get_ulong() << endl;
                break;
            }
        case CORBA::tk_float:
            {
                cout << value->get_float() << endl;
                break;
            }
        case CORBA::tk_double:
            {
                cout << value->get_double() << endl;
                break;
            }
        case CORBA::tk_boolean:
            {
                cout << value->get_boolean() << endl;
                break;
            }
        case CORBA::tk_char:
            {
                cout << value->get_char() << endl;
                break;
            }
    }
}

```



```

        break;
    }
    case CORBA::tk_ulonglong:
    {
#ifdef _VIS_VXWORKS_LONG_LONG_IOSTREAMS
        cout << value->get_ulonglong() << endl;
#else
        cout << "received unsigned long long";
        cout << "unsigned long long IOStreams currently not
supported" << endl;
#endif
        break;
    }
    default:
        cout << "Invalid Type" << endl;
    }
}

private:
CORBA::ORB_var _orb;
DynamicAny::DynAnyFactory_var _factory;
PortableServer::POA_var _poa;
PortableServer::ObjectId_var _oid;
};

```



# Part 8

## Backward Compatibility

### **In this part**

This part contains the following chapters:

<a href="#">Using the BOA in VisiBroker-RT for C++ 6.0</a>	<a href="#">page 389</a>
<a href="#">Migrating VisiBroker Code</a>	<a href="#">page 391</a>



# Using the BOA in VisiBroker-RT for C++ 6.0

*This chapter describes how to use the BOA with VisiBroker-RT for C++ 6.0.*

## NOTE

*The library `libboa.o` is required when building a VisiBrokerRT60 application to support the use of Basic Object Adapter (BOA). For a description of all the libraries provided by the VisiBroker-RT for C++ product, see [“Step 6: Integrating VisiBroker-RT with Tornado/VxWorks”](#).*

## Compiling your BOA code with VisiBroker-RT for C++ 6.0

If you have existing BOA code that you developed with a previous version of VisiBroker-RT for C++, you can continue to use them with the current version as long as you keep the following points in mind.

- To generate the necessary BOA base code, you must use the `“-boa”` option with the `idl2cpp` tool. For more information on using `idl2cpp` to generate the code, see the chapter “Programmer tools” in the VisiBroker-RT for C++ *Reference Guide*.

## Supporting BOA options

All OA command line options supported by VisiBroker-RT for C++ 3.x are still supported.

## Using object activators

BOA object activators are no longer supported with VisiBroker-RT for C++ 6.0.

In this release of VisiBroker, the Portable Object Adaptor (POA) supports the features that were provided by the BOA in VisiBroker 3.x releases. The POA uses servant activators and servant locators in place of object activators. Please refer to [“Using servants and servant managers”](#) for details on using POA servant managers.

## Naming Objects under the BOA

Though the BOA is deprecated in VisiBroker 6.0, you may still use it in conjunction with the Smart Agent to specify a name for your server objects which may be bound to in your client programs.

### Object names

When creating an object, a server must specify an object name if the object is to be made available to client applications through the `osagent`. When the server calls the BOA `obj_is_ready` method, the object's interface name will

only be registered with the VisiBroker osagent if the object is named. Objects that are given an object name when they are created return persistent object references, while objects which are not given object names are created as transient.

#### **NOTE**

If you pass an empty string for the object name to the object constructor in VisiBroker for C++, a persistent object is created, (that is, an object which is registered with the Smart Agent). If you pass a null reference to the constructor, a transient object is created.

The use of an object name by your client application is required if it plans to bind to more than one instance of an object at a time. The object name distinguishes between multiple instances of an interface. If an object name is not specified when the bind method is called, the osagent will return any suitable object with the specified interface.

#### **NOTE**

In VisiBroker 3.x, it was possible to have a multiple CORBA objects that provided different interfaces, all of which had the same object name, but in VisiBroker 6.0, different interfaces may not have string-equivalent names.

# Migrating VisiBroker Code

This chapter describes how to migrate your VisiBroker code from previous versions to VisiBroker-RT for C++ 6.0. In particular, it provides information on the following:

- Instructions on using BOA with VisiBroker-RT for C++ 6.0, changing your BOA code to POA, and using servant activators.
- List of changes to class names, and API calls in VisiBroker 6.0.

## Migrating BOA to POA

Class names have changed from previous versions of VisiBroker-RT for C++. Be sure to update your source files to point to the most recent class names. The following tables illustrates these name changes using an example class name.

**Table 35** Class name changes

Old class name	New class name
<code>_sk_Account</code>	<code>POA_Account</code>
<code>_sk_AccountManager</code>	<code>POA_AccountManager</code>
<code>_tie_Account</code>	<code>POA_Account_tie</code>
<code>_tie_AccountManager</code>	<code>POA_AccountManager_tie</code>

### Looking at an example

The `<install_location>/VisiBrokerRT60/examples/boa/boa2poa` directory contains an example of updating your BOA to the equivalent POA code.

In this example, the BOA code in `server.C` was updated to POA by:

- Obtaining a reference to the root POA instead of initializing the BOA
- Setting the appropriate POA policies to mimic the BOA characteristics
- Defining the servant (the POA has a different definition of a servant than the BOA)
- Activating the POA manager (no equivalent step for the BOA)

### Obtaining a reference to the root POA

When using the BOA, a reference to the BOA was obtained through `orb->BOA_init()`.

With the POA, however, you obtain a reference to the root POA. You do this by using `orb->resolve_initial_references("RootPOA")`. `resolve_initial_references` returns a value of type `CORBA::object` which you then narrow to the desired type.

**Code example 181** Obtaining a reference to the rootPOA

```
CORBA::object_var obj = resolve_initial_references("RootPOA");
PortableServer::POA_var rootPOA =
PortableServer::POA::_narrow(obj);
```

## Setting the POA policies

The characteristics of a POA are defined by the policies set for that POA. Each POA has its own set of policies; POAs can not inherit policies from other POAs.

In this example, persistent objects are used. With the BOA, persistent objects are those which have a specific instance name and are registered with the Smart Agent. A single BOA can support both persistent and transient objects. Under the POA, a persistent object is one that lives past the ORB instance that creates them. A single POA can support either persistent object or transient objects, not both. The supported object type is set by the POA policy. Since the root POA supports transient objects (by default), a new POA must be created to support persistent objects.

### Note

You cannot change the policies of a POA once it is created.

To support persistent objects, set the Lifespan policy to `PERSISTENT`. Once the appropriate policies have been set, a new POA can be created with

```
create_POA().
```

### Code example 182 Setting the POA policies

```
CORBA::PolicyList policies;
    policies.length(1);

    policies[(CORBA::ULong)0] =

    rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);

    // Create myPOA with the right policies
    PortableServer::POAManager_var mgr =
        rootPOA->the_POAManager();
    PortableServer::POA_var myPOA = rootPOA->create_POA(
        "bank_agent_poa", mgr, policies );
```

## Defining the servant

With the BOA, a servant is a CORBA object. In this example, the account manager object is created and then exported with `obj_is_ready()`.

With the POA, a servant is a programming object that provides the implementation of an abstract object. A servant is not a CORBA object. Under the POA scenario, the servant is created and then activated with a specific ID. You can use this ID to obtain the object reference.

### Code example 183 Defining and activating a servant

```
// Create the servant
    AccountManagerImpl *managerServant = new AccountManagerImp;
// Decide on the ID for the servant
    PortableServer::ObjectId var managerId =
        PortableServer::string_to_ObjectId("BankManager");
// Activate the servant with the ID on myPOA
    myPOA->activate_object_with_id(managerId,managerServant);
```

## Activating the POA manager

A POA Manager is an object that controls how a POA processes requests. By default, POA Managers are created in a holding state. In this state, all requests are routed to a holding queue and are not processed. To allow requests to be dispatched, the POA Manager associated with the POA must be changed from the holding state to an active state.

This is a new step required for the POA. There is no equivalent step for the BOA.

**Code example 184** Activating the POA manager

```
rootPOA->the_POAManager()->activate();
```

## Looking at the other classes

The `AccountImpl` and `AccountManagerImpl` class changes are much simpler. Most of the changes simply involve pointing to the new classes.

## Mapping BOA types to POA policies

The following table shows how to set your POA policies to mimic BOA behavior

**Table 36** Mapping BOA types to POA policies

	Transient BOA	Persistent BOA
TPOOL	Server Engine policy with TPOOL dispatcher LifeCycle property set to TRANSIENT	Server Engine policy with TPOOL dispatcher LifeCycle property set to PERSISTENTIDAssignment policy set to USER_ID BindSupport policy set to BY_INSTANCE
TSESSION	Server Engine policy with TSESSION dispatcher LifeCycle property set to TRANSIENT	Server Engine policy with TSESSION dispatcher LifeCycle property set to PERSISTENT IDAssignment policy set to USER_ID BindSupport policy set to BY_INSTANCE
Service-activated objects	LifeCycle property set to TRANSIENT Request Processing policy to USE_SERVANT_MANAGER Implicit Activation policy set to IMPLICIT_ACTIVATION	LifeCycle property set to PERSISTENT Request Processing policy to USE_SERVANT_MANAGER Implicit Activation policy set to IMPLICIT_ACTIVATION

## Migrating interceptors

The preferred method for migrating interceptors to VisiBroker-RT for C++ 6.0 is to use the new Portable Interceptors or the VisiBroker 6.0 interceptors.

### NOTE

Although VisiBroker 6.0 does provide wrappers that allow you to migrate your old interceptor code virtually unchanged (described below), the VisiBroker 6.0 wrappers for 3.x code do not provide functionality comparable to that of VisiBroker 6.0 interceptors.

### NOTE

The library `libmigrate.o` is required when building a VisiBrokerRT60 application to support the migration of Interceptors from VisiBroker 3.x to VisiBroker 6.x. For a description of all the libraries provided by the VisiBroker-RT for C++ product, see [“Step 6: Integrating VisiBroker-RT with Tornado/VxWorks”](#).

## Using VisiBroker 3.x interceptors

VisiBroker 6.0 ensures that method signatures of VisiBroker 3.x interceptors need not change, however installation and initialization procedures for old-style interceptors are changed.

### Installing VisiBroker 3.x interceptors

In order to use old-style interceptors with VisiBroker 6.0:

- 1 Add the include directive `#include "migration_c.hh"` to the files where you are going to use VisiBroker 3.x interceptors. The `migration_c.hh` header file contains the wrappers for the old interceptors.
- 2 Re-name the following interceptors as shown in the table:

**Table 37** Renaming interceptors

Old style name	New style name
<code>interceptor::BindInterceptor</code>	<code>interceptor_migration::BindInterceptorDelegate</code>
<code>interceptor::ChainBindInterceptor</code>	<code>interceptor_migration::BindInterceptorManager</code>
<code>interceptor::ChainClientInterceptor</code>	<code>interceptor_migration::ClientInterceptorDelegate</code>
<code>interceptor::ChainServerInterceptor</code>	<code>interceptor_migration::ServerInterceptorDelegate</code>
<code>interceptor::ClientInterceptorFactory</code>	<code>interceptor_migration::ClientInterceptorFactory</code>
<code>interceptor::ServerInterceptorFactory</code>	<code>interceptor_migration::ServerInterceptorFactory</code>

### Migrating BindInterceptors

The VisiBroker-RT for C++ 6.0 wrappers simulate the real `BindInterceptor`. In previous versions, to add a `BindInterceptor`, you would:

- 1 Get the reference to the `ChainBindInterceptor` by calling `ORB::resolve_initial_references("ChainBindInterceptor")`
- 2 Add the new interceptor to the chain.

To use your VisiBroker 3.x bind interceptor code in VisiBroker 6.0, you should instead:

- 1 Get the reference to `interceptor_migration::BindInterceptorManager` by calling `ORB::resolve_initial_references("ChainBindInterceptor");`
- 2 Then, create and add your `interceptor_migration::BindInterceptorDelegate` (rather than the `interceptor::BindInterceptor` that you used in VisiBroker 3.x) to the chain.

### Migrating client-side and server-side interceptors

In previous versions, to add a `ClientInterceptor` or a `ServerInterceptor`, you would:

First, implement the interface `Interceptor::ClientInterceptorFactory` or `Interceptor::ServerInterceptorFactory`. This interface provides methods for creating user-implemented `ClientInterceptors` and `ServerInterceptors`. You could then obtain a reference to the `ChainClientInterceptorFactory` or the `ChainServerInterceptorFactory` and, using these, you can add your own interceptors to the chain.

Under VisiBroker-RT for C++ 6.0, you should instead:

- 1 Implement `interceptor_migration::ClientInterceptorDelegate` or `interceptor_migration::ServerInterceptorDelegate`.

**2** Then, obtain a reference to the `Inteceptor_migration::ClientInterceptorFactory` or the `Inteceptor_migration::ServerInterceptorFactory`. These methods return the instance of the appropriate `InterceptorDelegate`.

After you have access to the client factory, server factory or both, you can install your client- or server-side interceptors into the appropriate factory chain. To do so, call

```
ORB::resolve_initial_references("ChainClientInterceptorFactory ")  
OR ORB::resolve_initial_references("ChainServerInterceptorFactory")
```

Once you have the references, you can use the `Add()` method to add to the chain. (This procedure is unchanged from VisiBroker 3.x.)

VisiBroker-RT for C++ 6.0 provides a sample application in `<install_location>/VisiBrokerRT60/examples/interceptors/ migration` which shows how to migrate an application that used older 3.x style interceptors to 6.0.



# CORBA exceptions

*This appendix provides information about CORBA exceptions that can be thrown by the VisiBroker-RT for C++ ORB, and explains possible causes for VisiBroker-RT for C++ throwing them.*

The following table lists CORBA exceptions, and explains reasons why the VisiBroker-RT for C++ ORB might throw them.

**Table 38** CORBA exceptions and possible causes

Exception	Explanation	Possible causes
<code>CORBA::BAD_CONTEXT</code>	An invalid context has been passed to the server.	An operation may raise this exception if a client invokes the operation but the passed context does not contain the context values required by the operation.
<code>CORBA::BAD_INV_ORDER</code>	The necessary prerequisite operations have not been called prior to the offending operation request.	<p>An attempt to call the <code>CORBA::Request::get_response()</code> or <code>CORBA::Request::poll_response()</code> methods may have occurred prior to actually sending the request.</p> <p>An attempt to call the <code>exception::get_client_info()</code> method may have occurred outside of the implementation of a remote method invocation. This function is only valid within the implementation of a remote invocation.</p>
<code>CORBA::BAD_OPERATION</code>	An invalid operation has been performed.	<p>An operation was called on an ORB that was already shut down.</p> <p>A server throws this exception if a request is received for an operation that is not defined on that implementation's interface. Ensure that the client and server were compiled from the same IDL.</p> <p>The <code>CORBA::Request::return_value()</code> method throws this exception if the request was not set to have a return value. If a return value is expected when making a DII call, be sure to set the return value type by calling the <code>CORBA::Request::set_return_type()</code> method.</p>

Exception	Explanation	Possible causes
<code>CORBA::BAD_PARAM</code>	A parameter passed to the ORB is invalid.	<p>Sequences throw <code>CORBA::BAD_PARAM</code> if an access is attempted to an invalid index. Make sure you use the <code>length()</code> method to set the length of the sequence before storing or retrieving elements of the sequence.</p> <p>ORB throws this exception if an invalid <code>Object_ptr</code> is passed as an <code>in</code> argument (for example, if a <code>nil</code> reference is passed).</p> <p>An attempt may have been made to send a <code>NULL</code> pointer where the IDL to C++ language mapping requires an initialized C++ object to be sent. For example, attempting to return <code>NULL</code> as a return value or <code>out</code> parameter from a method that should be returning a <code>sequence</code> will throw this exception. In this case a new <code>sequence</code> (probably of length 0) should be returned instead. The types which cannot be sent with the C++ <code>NULL</code> value include <code>Any</code>, <code>Context</code>, <code>struct</code>, or <code>sequence</code>.</p> <p>An attempt was made to send a value that is out of range for an enumerated data type.</p> <p>An attempt may have been made to construct a <code>TypeCode</code> with an invalid <code>kind</code> value.</p> <p>An attempt may have been made to insert a <code>nil</code> object reference into an <code>Any</code>.</p> <p>Using the DII and one way method invocations, an <code>OUT</code> argument may have been specified. An interface repository thrown this exception if an argument passed into an IR object's operation conflicts with its existing settings. See the compiler errors for more information.</p>
<code>CORBA::BAD_TYPECODE</code>	The ORB has encountered a malformed type code.	
<code>CORBA::CODESET_INCOMPATIBLE</code>	Communication between client and server native code sets fails because the code sets are incompatible.	The code sets used by the client and server cannot work together. For instance, the client uses ISO 8859-1 and the server uses the Japanese code set.
<code>CORBA::COMM_FAILURE</code>	Communication is lost while an operation is in progress, after the request was sent by the client but before the reply has been returned.	<p>An existing connection may have closed due to failure at the other end of the connection.</p> <p>A new connection request may have failed due to resource limits on the client or server machine (the maximum number of connections has been reached).</p> <p>When <code>COMM_FAILURES</code> occur due to system exceptions, the system error number is set in the minor code of the <code>COMM_FAILURE</code>. Check the minor code against the system-specific error numbers (for example in the <code>include/sys/errno.h</code> or <code>msdev\include\winerror.h</code> files).</p>
<code>CORBA::DATA_CONVERSION</code>	The ORB cannot convert the representation of marshaled data into its native representation or vice-versa.	An attempt to marshal Unicode characters with <code>Output.write_char()</code> or <code>Output.write_string</code> fails.

Exception	Explanation	Possible causes
CORBA::FREE_MEM	The ORB failed to free dynamic memory.	The memory segments that the ORB is trying to free may be locked. The heap could be corrupt.
CORBA::IMP_LIMIT	An implementation limit was exceeded in the ORB run time.	The ORB may have reached the maximum number of references it can hold simultaneously in an address space. The size of the parameter may have exceeded the allowed maximum. The maximum number of running clients and servers has been exceeded.
CORBA::INITIALIZE	A necessary initialization has not been performed.	The <code>ORB_init()</code> method may not have been called. All clients must call the <code>ORB_init()</code> method prior to performing any ORB-related operations. This call is typically made immediately upon program startup at the top of the main routine.
CORBA::INTERNAL	An internal ORB error has occurred.	An internal ORB error may have occurred. For instance, the internal data structures of the ORB may have been corrupted.
CORBA::INTF_REPOS	An instance of the Interface Repository could not be located.	If an object implementation cannot locate an interface repository during an invocation of the <code>get_interface()</code> method, this exception will be thrown to the client. Ensure that an Interface Repository is running, and that the requested object's interface definition has been loaded into the Interface Repository.
CORBA::INV_FLAG	An invalid flag was passed to an operation.	A Dynamic Invocation Interface request was created with an invalid flag.
CORBA::INV_IDENT	An IDL identifier is syntactically invalid.	An identifier passed to the interface repository is not well formed. An illegal operation name is used with the Dynamic Invocation Interface.
CORBA::INV_OBJREF	An invalid object reference has been encountered.	The ORB will throw this exception if an object reference is obtained that contains no usable profiles. The <code>ORB::string_to_object()</code> method will throw this exception if the stringified object reference does not begin with the characters <code>"IOR:"</code> .
CORBA::INV_POLICY	An invalid policy override has been encountered.	This exception can be thrown from any invocation. It can be raised when an invocation cannot be made due to an incompatibility between policy overrides that apply to the particular invocation.
CORBA::INVALID_TRANSACTION	A request carried an invalid transaction context.	See your transaction service documentation for more information on this exception.

Exception	Explanation	Possible causes
CORBA::MARSHAL	Error marshalling parameter or result.	A request or reply from the network is structurally invalid. This error typically indicates a bug in either the client-side or server-side run time. For example, if a reply from the server indicates that the message contains 1000 bytes, but the actual message is shorter or longer than 1000 bytes, the ORB raises this exception. A <code>MARSHAL</code> exception can also be caused by using the DII or DSI incorrectly. For example, if the type of the actual parameters sent does not agree with IDL signature of an operation.
CORBA::NO_IMPLEMENT	The requested object could not be located.	<p>A <code>bind()</code> call or some other remote operation fails because the target could not be found. To dynamically locate implementations through the VisiBroker-RT for C++ <code>bind()</code> call, a Smart Agent must be running in your ORB domain. In addition, an implementation of the requested interface must be available on the same ORB domain. To verify the presence of a Smart Agent, run the <code>osfind</code> utility. This utility prints the locations of all Smart Agents on your current domain (that is, all Smart Agents listening on your environment's <code>OSAGENT_PORT</code>).</p> <p>The <code>osfind</code> utility will also print the interface name and instance name of all available implementations. In summary, prior to running the client program:</p> <ol style="list-style-type: none"> <li>1 Verify that a Smart Agent is running and accessible on the network.</li> <li>2 Verify that the desired implementation is available on the network.</li> </ol> <p>If the <code>rebind()</code> method is enabled and an object implementation becomes unavailable, <code>NO_IMPLEMENT</code> will be thrown if another provider cannot be located.</p>
CORBA::NO_MEMORY	The ORB runtime has run out of memory.	
CORBA::NO_PERMISSION	The caller has insufficient privileges to complete an invocation.	
CORBA::NO_RESOURCES	A necessary resource could not be acquired.	<p>If a new thread cannot be created, this exception will be thrown.</p> <p>A server will throw this exception when a remote client attempts to establish a connection if the server cannot create a socket—for example, if the server runs out of file descriptors. The minor code contains the system error number obtained after the server's failed <code>::socket()</code> or <code>::accept()</code> call.</p> <p>A client will similarly throw this exception if a <code>::connect()</code> call fails due to running out of file descriptors. Running out of memory may also throw this exception.</p>

Exception	Explanation	Possible causes
<code>CORBA::NO_RESPONSE</code>	A client attempts to retrieve the result of a deferred synchronous call, but the response for the request is not yet available.	
<code>CORBA::OBJ_ADAPTER</code>	An administrative mismatch has occurred.	<p>A server has attempted to register itself with an implementation repository under a name that already is in use, or is unknown to the repository.</p> <p>The POA has raised an <code>OBJ_ADAPTER</code> error due to problems with the application's servant managers.</p>
<code>CORBA::OBJECT_NOT_EXIST</code>	The requested object does not exist.	A server throws this exception if an attempt is made to perform an operation on an implementation that does not exist within that server. This will be seen by the client when attempting to invoke operations on deactivated implementations.
<code>CORBA::PERSIST_STORE</code>	A persistent storage failure has occurred.	Attempts to establish a connection to a database has failed, or the database is corrupt.
<code>CORBA::REBIND</code>	The client has received an IOR which conflicts with QOS policies.	Thrown anytime the client gets an IOR which will conflict with the QOS policies that have been set. If the <code>RebindPolicy</code> has a value of <code>NO_REBIND</code> , <code>NO_CONNECT</code> , or <code>VB_NOTIFY_REBIND</code> and an invocation on a bound object reference results in an object forward or a location forward message.
<code>CORBA::TRANSACTION_REQUIRED</code>	The request carried a null transaction context, but an active transaction is required.	See your transaction service documentation for more information on this exception.
<code>CORBA::TRANSACTION_ROLLEDBACK</code>	The transaction associated with a request has already been rolled back, or marked for roll back.	See your transaction service documentation for more information on this exception.
<code>CORBA::TRANSIENT</code>	An error has occurred, but the ORB believes it is possible to retry the operation.	A communications failure may have occurred and the ORB is signalling that an attempt should be made to rebind to the server with which communications have failed. This exception will not occur if the <code>BindOptions</code> are set to false with the <code>enable_rebind()</code> method, or the <code>RebindPolicy</code> is properly set.

Exception	Explanation	Possible causes
CORBA::UNKNOWN	The ORB could not determine the thrown exception.	<p>The server throws something other than a correct exception, such as a Java runtime exception.</p> <p>There is an IDL mismatch between the server and the client, and the exception is not defined in the client program.</p> <p>In DII, if the server throws an exception not known to the client at the time of compilation and the client did not specify an exception list for the <code>CORBA::Request</code>. Set the property <code>vbroker.orb.warn=2</code> on the server to see which runtime exception caused the problem.</p>
CORBA::UnknownUserException	A user exception has been received, but the client has no compile-time knowledge of that exception.	<p>When a client reads in a user exception from a server, it will generate this exception if it has no compile-time knowledge of the exception type. The client can see the type of the exception, and is given the marshalled buffer containing the contents of the exception. The ORB has no way to unmarshal the exception on its own.</p>

---

**Table 39** CORBA exception minor codes

<b>System exception</b>	<b>Minor code</b>	<b>Explanation</b>
BAD_PARAM	1	Failure to register, unregister, or lookup the value factory
	2	RID already defined in the interface repository
	3	Name already used in the context in the interface repository
	4	Target is not a valid container
	5	Name clash in inherited context
	6	Incorrect type for abstract interface
MARSHAL	1	Unable to locate value factory
NO_IMPLEMENT	1	Missing local value implementation
	2	Incompatible value implementation version
BAD_INV_ORDER	1	Dependency exists in the interface repository preventing the destruction of the object
	2	Attempt to destroy indestructible objects in the interface repository
	3	Operation would deadlock
	4	ORB has shut down
OBJECT_NOT_EXIST	1	Attempt to pass an unactivated (unregistered) value as an object reference

---



# Glossary

This is a glossary of terms used in VisiBroker-RT.

## **activation**

Process of preparing an object to receive requests.

## **API (application program interface)**

A set of operations which allows a (client) program to access functionality contained in a library or another program, possibly a server.

## **attribute**

An attribute is a property of an object. For example, a Point object might have two coordinate attributes, X and Y.

## **application**

A computer program designed to help people perform a certain type of work. Depending on the work for which it was designed, an application can manipulate text, numbers, graphics, or a combination of these elements.

## **bind (NamingService)**

The process of associating a Name with a remote object in a server application, so that a client application can resolve the Name and obtain a reference to the remote object.

## **bind (VisiBroker)**

The process of establishing a connection to a server hosting an object we are interested in.

## **class**

A class is a data type which declares what attributes and operations an instantiated object will have.

## **client/server**

A programming strategy in which two programs cooperate with one another using some common and conventional protocol. For example, on the worldwide web, the browser is the client software, the web server is the server software, and HTTP is the protocol. Clients send requests to servers, and servers send replies to clients.

## **component**

A chunk or object of a distributed application.

## **CORBA (common object request broker architecture)**

An open, object-oriented, standard architecture developed by the OMG for the interoperability of distributed objects on different platforms, under different operating systems and implemented in different programming languages.

### **distributed application**

An application whose components are distributed across multiple computers on a network but which seem to be running on the user's computer.

### **distributed objects**

Software modules that are designed to work together but reside in multiple computer systems throughout the organization. A program in one machine sends a message to an object in a remote machine to perform some processing. The results are sent back to the calling machine.

### **Dynamic Invocation Interface (DII)**

An API that allows a client to make dynamic invocations on remote CORBA objects. It is used if at compile time a client does not have knowledge about an object it wants to invoke. Once an object is discovered, the client program can obtain a definition of it, issue a parameterized call to it, and receive a reply from it, all without having a type-specific client stub for the remote object.

### **Dynamic Skeleton Interface (DSI)**

An API that provides a way to deliver requests from an ORB to an object implementation when the type of the object implementation is not known at compile time. DSI, which is the server side analog to the client side DII, makes it possible for the application programmer to inspect the parameters of an incoming request to determine a target object and method.

### **failover**

Having more than one system which may be used as backup in case one of the systems fail.

### **HTML (hypertext markup language)**

An SGML application used to specify the structure of a hypertext (web) document.

### **HTTP (hypertext transport protocol)**

A protocol used by worldwide web client/server applications to connect and transfer HTML documents.

### **IDL (interface definition language)**

A high-level, programming language independent, declarative language for defining the interface of a distributed object.

### **IDL compiler**

A compiler which translates an IDL specification into programming language specific stub and skeleton files which are used to implement distributed objects.

### **IDL file**

A plain text file which declares modules and interfaces in IDL.

## **IIOP (Internet Inter-ORB protocol)**

A TCP/IP-based protocol developed by the OMG. The IIOP enables two or more ORBs to work in conjunction to provide requests to objects.

## **interface**

The set of public attributes and operations (or signature) which a (server) object exposes to a (client) object.

## **interface repository**

A service that contains all the registered component interfaces, the methods they support, and the parameters they require. The IFR stores, updates, and manages object interface definitions. Programs may use the IFR APIs to access and update this information.

## **master/slave**

The Interoperable Naming service runs master and slave naming service for a failover purposes. The master is the primary service and the slave is the fallback service in general.

## **method**

An operation of an object (the server) which when called by another object (the client) performs some declared behavior.

## **multithreading**

A programming technique whereby an application can be divided into more than one asynchronous time-slice (or thread of execution).

## **Name**

A name is a predefined name, an alias, or a convenient handle which is associated with a server object. To bind a name to an object, you use the bind method. To resolve a name (i.e., to retrieve a pointer) use the resolve method.

## **namespace**

A collection of names, no two of which are identical.

## **naming service**

A CORBA service that allows CORBA objects to be named by means of binding a name to an object reference. The name binding may be stored in the naming service, and a client may supply the name to obtain the desired object reference.

## **n-tier**

A programming strategy in which  $n$  programs cooperate with one another using some common and conventional protocol. For example, a client/server application can also be described as a two-tier application.

## **object**

A programming entity which is defined by its properties (attributes) and behaviors (operations). Objects have unique identities and can be

distinguished from one another. An object is an instance of a particular class.

### **object adapter**

The ORB component which provides object reference, activation, and state related services to an object implementation.

### **object implementation**

A server process that offers one or more objects which client applications may use.

### **object reference**

A handle to an object, used by a client application to invoke methods on the object.

### **OMG (Object Management Group)**

A consortium of software companies which is charged with the development of the CORBA specification: (see <http://www.omg.org/>).

### **operation**

The function of an object (the server) which when called by another object (the client) performs some declared behavior.

### **ORB (object request broker)**

The ORB allows clients to make and receive requests and responses.

### **package**

A logical collection of Java classes that provide similar or related features.

### **protocol**

A language which defines the requests and replies of client/server objects or applications.

### **RMI (remote method invocation)**

A Java API which allows objects to be instantiated and used in a distributed application.

### **RPC (remote procedure call)**

A strategy which allows procedures to be called from outside the currently running program's memory. RPC allows two or more different programs to interoperate with one another.

### **scalability**

The degree to which a system or application can handle increasing or decreasing demand on system resources without significant performance degradation.

### **servant**

An instance of an object implementation for an IDL interface. The servant object is registered with the ORB so that the ORB knows where to send

invocations. It is the servant that performs the services requested when a CORBA object's method is invoked.

### **server**

An object or application which performs a service for other objects or applications (the clients). A server replies to a client's request using a protocol.

### **service**

The functionality of a given server.

### **SGML (standard generalized markup language)**

Abbreviation of Standard Generalized Markup Language, a system for organizing and tagging elements of a document. SGML was developed and standardized by the International Organization for Standards (ISO). SGML itself does not specify any particular formatting; rather, it specifies the rules for tagging elements. These tags can then be interpreted to format elements in different ways.

### **signature**

The set of parameters and their names of a given operation which uniquely identify the operation.

### **skeleton (file)**

An older construct (used prior to VisiBroker 4.0): a serverside file generated from IDL which is to be implemented by the object implementor.

### **stringification**

Converting an object reference to a character string format. Used when an object reference needs to be made persistent to a text file or stored in a database or sent to a client program.

### **stub (file)**

The portion of a client or server program that executes the data marshalling and network transportation routines.

### **TCP/IP (transport control protocol / internet protocol)**

TCP is one of the main protocols in TCP/IP networks. Whereas the IP protocol deals only with packets, TCP enables two hosts to establish a connection and exchange streams of data. TCP guarantees delivery of data and also guarantees that packets will be delivered in the same order in which they were sent.

### **thread**

A thread is a stream of execution within a process. In a multithreaded environment, multiple tasks can execute concurrently within the same application.

### **transaction server**

A server which supports transactional semantics, (e.g., commit or rollback).

**XML (extensible markup language)**

Extensible Markup Language. A specification developed by the World Wide Web Consortium (W3C). XML is a subset of the SGML document language, designed especially for Web documents.