# Borland VisiBroker™ 8.0

# VisiBroker for C++ Developer's Guide

Borland Software Corporation
20450 Stevens Creek Blvd., Suite 800
Cupertino, CA 95014 USA
www.borland.com

Refer to the file deploy.html for a complete list of files that you can distribute in accordance with the License Statement and Limited Warranty.

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. Please refer to the product CD or the About dialog box for the list of applicable patents. The furnishing of this document does not give you any license to these patents.

Copyright 1992–2006 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. All other marks are the property of their respective owners.

Microsoft, the .NET logo, and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

For third-party conditions and disclaimers, see the Release Notes on your product CD.

VB 80 VisiBroker C++ API
April 2007

Borland®

# Contents

1

# Understanding the CORBA model

This section introduces VisiBroker, which comprises both the VisiBroker for C++ and the VisiBroker for Java ORBs. Both are complete implementations of the CORBA 3.0 specification (excluding CCM and GIOP 1.3). This section describes VisiBroker features and components.

## What is CORBA?

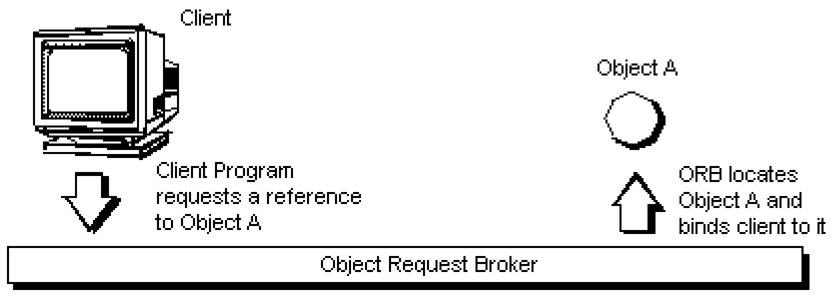
The Common Object Request Broker Architecture (CORBA) allows distributed applications to interoperate (application-to-application communication), regardless of what language they are written in or where these applications reside.

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 by presenting a well-defined interface. Use of these interfaces, themselves written in the standardized Interface Definition Language (IDL) 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 role of the Object Request Broker (ORB) is to track and maintain these interfaces, facilitate communication between them, and provide services to applications making use of them. The ORB itself is not a separate process. It is a collection of libraries and network resources that integrates within end-user applications, and allows your client applications to locate and use disparate objects.

The Object Request Broker in the following figure connects a client application with the objects it wants to use. The client program does not need to know whether the object it seeks 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.

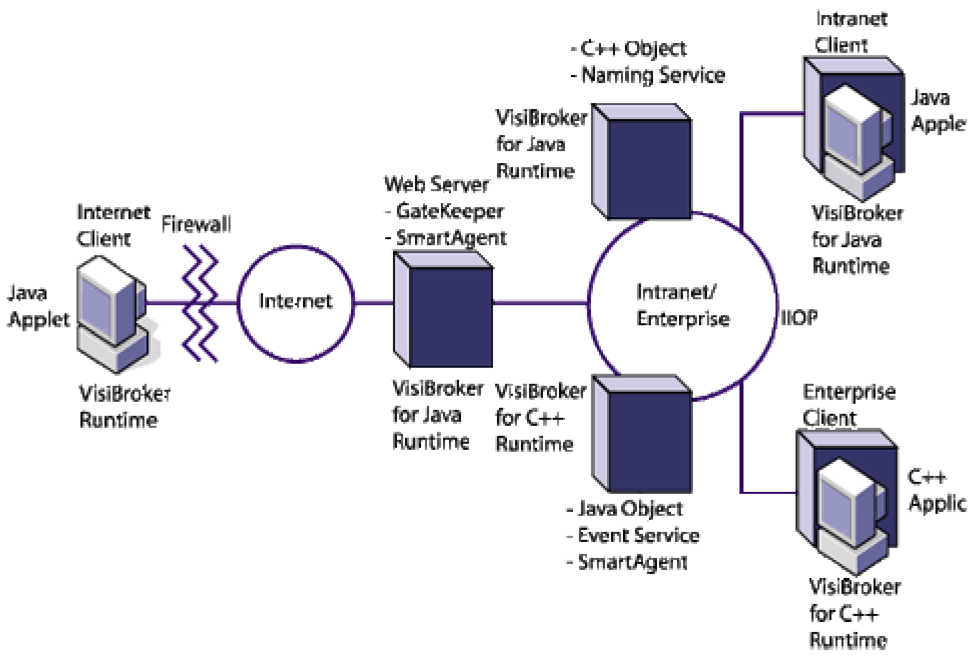
**Figure 1.1** Client program acting on an object


Client
Object A
Client Program requests a reference to Object A
ORB locates Object A and binds client to it
Object Request Broker

## What is VisiBroker?

VisiBroker provides a complete CORBA 3.0 ORB runtime and supporting development environment for building, deploying, and managing distributed applications for both C++ and Java that are open, flexible, and interoperable. Objects built with VisiBroker are easily accessed by Web-based applications that communicate using the Internet Inter-ORB Protocol (IIOP) standard for communication between distributed objects through the Internet or through local intranets. VisiBroker has a built-in implementation of IIOP that ensures high-performance and interoperability.

**Figure 1.2** VisiBroker Architecture



## VisiBroker Features

VisiBroker has several key features as described in the following sections.

### VisiBroker's Smart Agent (osagent) Architecture

VisiBroker's Smart Agent (`osagent`) is a dynamic, distributed directory service that provides naming 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 object-invocation time. VisiBroker 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 host, if it is so configured. For details on the Smart Agent see "Using the Smart Agent" and "Using Quality of Service (QoS)".

### Enhanced Object Discovery Using the Location Service

VisiBroker 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 "Using the Location Service."

### Implementation and Object Activation Support

The Object Activation Daemon (OAD) is the VisiBroker implementation of the Implementation Repository. The OAD can be used to automatically start object implementations when clients need to use them. Additionally, VisiBroker 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 on a server.

## Robust thread and connection management

VisiBroker provides native support for single- and multi-threaded thread management. With VisiBroker's thread-per-session model, threads are automatically allocated on the server (per client connection) to service multiple requests, and then are terminated when each connection ends. With the thread pooling model, threads are allocated based on the amount of request traffic to and from server objects. 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'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 “Managing threads and connections” for details on how VisiBroker manages threads and connections.

## IDL compilers

VisiBroker comes with three IDL compilers that make object development easier,

- `idl2java`: The `idl2java` compiler takes IDL files as input and produces the necessary client stubs and server skeletons in Java.
- `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. Unlike the previous two compilers, `idl2ir` functions with both the C++ and Java ORBs.

See “Using IDL” and “Using Interface Repositories” for details on these compilers.

## Dynamic invocation with DII and DSI

VisiBroker provides implementations of both the Dynamic Invocation Interface (DII) and the Dynamic Skeleton Interface (DSI) for dynamic invocation. The DII allows client applications to dynamically create requests for objects that were not defined at compile time. The DSI allows servers to dispatch client operation requests to objects that were not defined at compile time. See “Using the Dynamic Invocation Interface” and “Using the Dynamic Skeleton Interface” for more information.

## Interface and implementation repositories

The Interface Repository (IR) is an online database of meta information about the VisiBroker ORB objects. Meta information stored for objects includes information about modules, interfaces, operations, attributes, and exceptions. “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.

The Object Activation Daemon is a VisiBroker interface to the Implementation Repository that is used to automatically activate the implementation when a client references the object. See “Using the Object Activation Daemon (OAD)” for more information.

### Server-side portability

VisiBroker 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 additional functionality, such as the POA Manager and Servant Manager which create and manages instances of your objects. See "Using POAs" for more information.

### Customizing the VisiBroker ORB with interceptors and object wrappers

VisiBroker's Interceptors enable developers to view under-the-cover communications between clients and servers. The VisiBroker Interceptors are Borland's proprietary interceptors. Interceptors can be used to extend the VisiBroker ORB with customized client and server code that enables load balancing, monitoring, or security to meet the specialized needs of distributed applications. See "Using Portable Interceptors" for information.

VisiBroker also includes the Portable Interceptors, based on the OMG standardized feature, that allow you to write portable code for interceptors and use it with different vendor ORBs. For more information, refer to the *COBRA 3.0 specification*.

VisiBroker'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.

### Event Queue

The event queue is designed as a server-side only feature. A server can register the listeners to the event queue based on the event types that the server is interested in, and the server processes those events when the need arises.

### Backing stores in the Naming Service

The new interoperable Naming Service integrates with pluggable backing stores to make its state persistent. This ensures easy fault tolerance and failover functionality in the Naming Service. See "Pluggable backing store" for more information.

### GateKeeper

The GateKeeper allows client programs to issue operation requests to objects that reside on a web server and to receive callbacks from those objects, all the while conforming to the security restrictions imposed by web browsers. The Gatekeeper also handles communication through firewalls and can be used as an HTTP daemon. It is fully compliant with the OMG CORBA Firewall Specification. For more information see the VisiBroker "Introduction to GateKeeper."

## VisiBroker CORBA compliance

VisiBroker is fully compliant with the CORBA specification (version 3.0) (excluding CCM and GIOP 1.3) from the Object Management Group (OMG). For more details, refer to the CORBA specification located at http://www.omg.org/.

## VisiBroker Development Environment

VisiBroker can be used in both the development and deployment phases. The development environment includes the following components:

- Administration and programming tools
- VisiBroker ORB

### Programmer'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 for both the VisiBroker for Java and VisiBroker for C++. |
| `idl2cpp` | This tool generates C++ stubs and skeletons from an IDL file. |
| `idl2java` | This tool generates Java stubs and skeletons from an IDL file |
| `java2iiop` | Generates Java stubs and skeletons from a Java file. This tool allows you to define your interfaces in Java, rather than in IDL. |
| `java2idl` | Generates an IDL file from a file containing Java bytecode. |

### CORBA services tools

The following tools are used to administer the VisiBroker ORB during development:

| Tool | Purpose |
|---|---|
| `irep` | Used to manage the Interface Repository. See “Using Interface Repositories.” |
| `oad` | Used to manage the Object Activation Daemon (OAD). See “Using the Object Activation Daemon (OAD).” |
| `nameserv` | Used to start an instance of the Naming Service. See “Using the VisiNaming Service.” |

### Administration Tools

The following tools are used to administer the VisiBroker ORB during development:

| Tool | Purpose |
|---|---|
| `oadutil list` | Lists VisiBroker ORB object implementations registered with the OAD. |
| `oadutil reg` | Registers an VisiBroker ORB object implementation with the OAD. |
| `oadutil unreg` | Unregisters an VisiBroker ORB object implementation with the OAD. |
| `osagent` | Used to manage the Smart Agent. See “Using the Smart Agent.” |
| `osfind` | Reports on objects running on a given network. |

## Interoperability with VisiBroker

Applications created with VisiBroker for Java can communicate with object implementations developed with VisiBroker for C++. Likewise, for applications created with VisiBroker for C++, these applications can also communicate with objects implementations developed with VisiBroker for Java. For example, if you want to use Java application on VisiBroker for C++, use the same IDL you used to develop your Java application as input to the VisiBroker IDL compiler, supplied with VisiBroker for C++. You may then use the resulting C++ skeletons to develop the object implementation. To use the C++ application on VisiBroker for Java, repeat the process. However, you will use the VisiBroker IDL complier with VisiBroker for Java instead.

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

## 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's use of IIOP allows client and server applications you develop with VisiBroker to interoperate with a variety of ORB products from other vendors.

## IDL to C++ Mapping

VisiBroker conforms with the *OMG IDL/C++ Language Mapping Specification*. See the VisiBroker *Programmer's Reference* for a summary of VisiBroker's current IDL to C++ language mapping, as implemented by the `idl2cpp` compiler. For each IDL construct there is a section that describes the corresponding C++ construct, along with code samples.

For more information about the mapping specification, refer to the *OMG IDL/C++ Language Mapping Specification*.

2

# Developing an example application with VisiBroker

This section uses an example application to describe the development process for creating distributed, object-based applications for both Java and C++.

The code for the example application is provided in the `bank_agent.html` file. You can find this file in:

```
<install_dir>/examples/vbroker/basic/bank_agent/
```

## Development process

When you develop distributed applications with VisiBroker, you must first identify the objects required by the application. The following figure illustrates the steps to develop a sample bank application. Here is a summary of the steps taken to develop the bank sample:

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 compilers to generate the client stub code and server POA servant code.

   With the interface specification described in step 1, use the `idl2java` or `idl2cpp` compilers to generate the client-side stubs and the server-side classes for the implementation of the remote objects.

3. Write the client program code.

   To complete the implementation of the client program, initialize the VisiBroker 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.

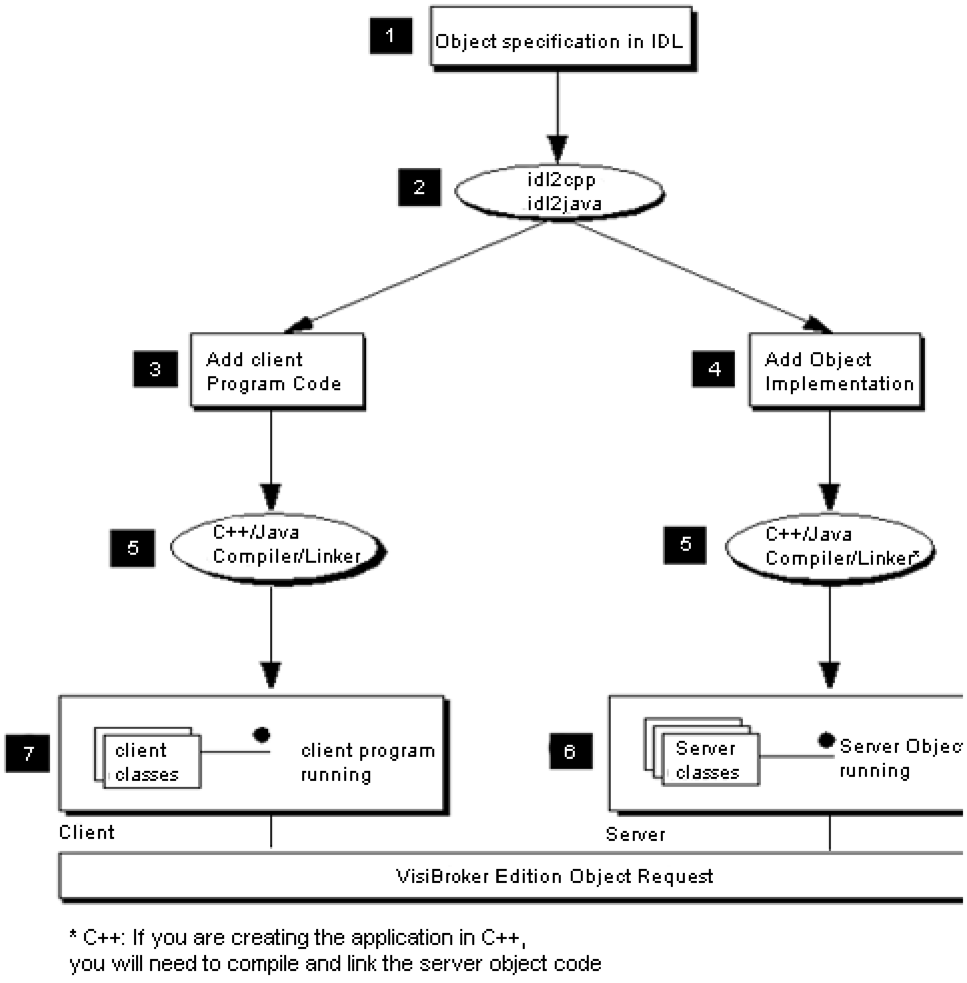
To complete the implementation of the server object code, we must derive from the `AccountPOA` and `AccountManagerPOA` classes, provide implementations of the interfaces' methods, and implement the server's `main` routine.

5 Compile the client and server code using the appropriate stubs and skeletons.

6 Start the server.

7 Run the client program.

**Figure 2.1** Developing the sample bank application


1
Object specification in IDL
2
idl2cpp
idl2java
3
Add client Program Code
4
Add Object Implementation
5
C++/Java Compiler/Linker
5
C++/Java Compiler/Linker*
7
client classes
client program running
Client
6
Server classes
Server Objec
running
Server
VisiBroker Edition Object Request
* C++: If you are creating the application in C++,
you will need to compile and link the server object code

## Step 1: Defining object interfaces

The first step to creating an application with VisiBroker 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.

You then use the `idl2cpp` compiler to generate stub routines and servant code compliant with 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.

### 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.

The sample below shows the contents of the `Bank.idl` file for the `bank_agent` 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.

```
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's `idl2cpp` to generate C++ stub routines for the client program, and skeleton code for the object implementation.

The client program uses the stub routines 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.

Because the `Bank.idl` file requires no special handling, you can compile the file with the following command.

```
prompt> idl2cpp Bank.idl
```

For more information on the command-line options for the `idl2cpp` compiler, see "Using IDL."

### 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 `AccountPOA` and `AccountManagerPOA` servant classes.
- `Bank_s.cpp`: Contains the internal routines used by the server.

You will use the `Bank_c.hh` and `Bank_c.cpp` files to build the client application. The `Bank_s.hh` and `Bank_s.cpp` files are for building the server object. All generated files have either a `.cpp` or `.hh` suffix to help you distinguish them from source files.

**Windows**

The default suffix for generated files from the `idl2cpp` compiler is `.cpp`. However, the Makefiles associated with the examples for VisiBroker use the `-src` suffix to change the output to the specified extension.

**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 `Bank` code generated by the `idl2cpp` compiler as shown in the previous example.

The `Client.C` file illustrates this example and is included in the `bank_agent` directory. Normally, you would create this file.

### Client.C

The `Client` program implements the client application which obtains the current balance of a bank account. The bank client program performs these steps:

1. Initializes the VisiBroker ORB.
2. Binds to an `AccountManager` object.
3. Obtains the balance of the `Account` using the object reference returned by `bind()`.
4. Obtains the balance by invoking `balance` on the `Account` object.

```
#include "Bank_c.hh"
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB.
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      // 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_ptr manager =
      Bank::AccountManager::_bind("/bank_agent_poa", managerId);
      // use argv[1] as the account name, or a default.
      const char* name = argc > 1 ? argv[1] : "Jack B. Quick";







      // Request the account manager to open a named account.
      Bank::Account_ptr account = manager->open(name);
      // Get the balance of the account.
      float balance = account->balance();
      // Print out the balance.
      cout << "The balance in " << name << "'s account is $" << balance <<
endl;
   } catch(const CORBA::Exception& e) {
      cerr << e << endl;
      }
   }
```

### Binding to the AccountManager object

Before your client program can invoke the `open(String name)` member function, the client 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 implemented automatically by `idl2cpp`. The `bind()` member function requests the VisiBroker ORB to locate and establish a connection to the server.

If the server is successfully located and a connection is established, a proxy object is created to represent the server's `AccountManagerPOA` object. A pointer 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.

Several other member functions are provided that allow your client program to manipulate an `AccountManager` object reference.

## 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 of `Bank` 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.

### Server programs

This file implements the Server class for the server side of our banking example. The code sample below ia an example of a server side program. The server program does the following:

- Initializes the Object Request Broker.
- Creates a Portable Object Adapter with the required policies.
- Creates the account manager servant object.
- Activates the servant object.
- Activates the POA manager (and the POA).
- Waits for incoming requests.

```
#include "BankImpl.h"
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB.
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
      // 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
      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
      myPOA->activate_object_with_id(managerId, &managerServant);
      // Activate the POA Manager
      poa_manager->activate();
      cout << myPOA->servant_to_reference(&managerServant) << " is ready" <<
endl;
      // Wait for incoming requests
      orb->run();
   } catch(const CORBA::Exception& e) {
      cerr << e << endl;
      return 1;
   }
   return 0;
}
```

### 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_c.hh` file and notice that it is derived from the `Account` class. The figure below shows the class hierarchy.

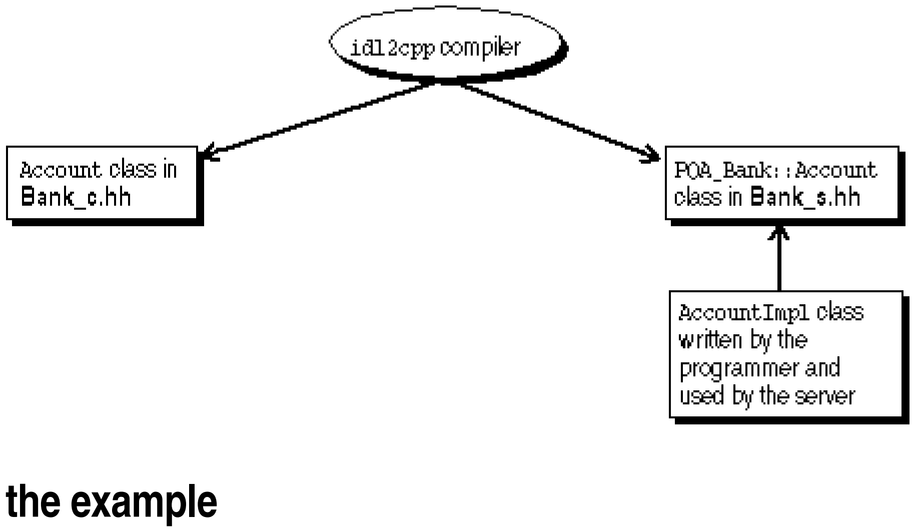

idl2cpp compiler
Account class in Bank_c.hh
POA_Bank::Account class in Bank_s.hh
AccountImpl class written by the programmer and used by the server

**Figure 2.2** Class hierarchy for the AccountImpl interface

## Step 5: Building the example

The `examples` directory of your VisiBroker release contains a `Makefile.cpp` for this example and other VisiBroker examples.

The `Client.C` that you created and the generated `Bank_c.cc` file are compiled and linked together to create the client program. The `Server.C` file that you created, along with the generated `Bank_s.cpp` and the `Bank_c.cpp` files, are compiled and linked to create the bank account server. Both the client program and the server must be linked with the VisiBroker ORB library.

**Note:**

VisiBroker for C++ 8.0 has a kernel library, the ORB Core, and six component libraries. The ORB Core must be linked to any VBC application, while the component libraries can be selected on demand by linking or using property.

To reduce the migration effort, VisiBroker for C++ 8.0 also provides a library having the same name as previous single ORB library. Although, the library suffix may be different on some platforms.

The examples directory also contains a file named `stdmk` (for UNIX) or `stdmk_nt` (for Windows NT), and defines file location and variable settings to be used by the `Makefile`.

You may need to customize the `stdmk` or `stdmk_nt` file if your compiler does not support the specified flags.

### Compiling the example

**Windows**

Assuming VisiBroker is installed in `C:\vbroker`, type the following to compile the example:

```
prompt> C:
prompt> cd vbroker\examples\basic\bank_agent
prompt> nmake -f Makefile.cpp
```

The Visual C++ `nmake` command runs the `idl2cpp` compiler and then compiles each file.

If you encounter some problems while running `make` , check that your path environment variable points to the `bin` directory where you installed the VisiBroker software.

Also, try setting the `VBROKERDIR` environment variable to the directory where you installed the VisiBroker software.

**UNIX**

Assuming VisiBroker is installed in `/usr/local`, type the following to compile the example:

```
prompt> cd /usr/local/vbroker/examples/basic/bank_agent
prompt> make cpp
```

In this example, `make` is the standard UNIX facility. If you do not have it in your `PATH`, see your system administrator.

## Step 6: Starting the server and running the example

Now that you have compiled your client program and server implementation, you are ready to run your first VisiBroker application.

### Starting the Smart Agent

Before you attempt to run VisiBroker client programs or server implementations, you must first start the Smart Agent on at least one host in your local network.

The basic command for starting the Smart Agent is as follows:

```
prompt> osagent
```

The Smart Agent is described in detail in “Using the Smart Agent.”

### Starting the server

#### Windows

Open a DOS prompt window and start your server by using the following DOS command:

```
prompt> start Server
```

#### UNIX

Start your `Account` server by typing:

```
prompt> Server&
```

### Running the client

#### Windows

Open a separate DOS prompt window and start your client by using the following DOS command:

```
prompt> Client
```

#### UNIX

To start your client program, type the following command:

```
prompt> Client
```

You should see output similar to that shown below (the account balance is computed randomly).

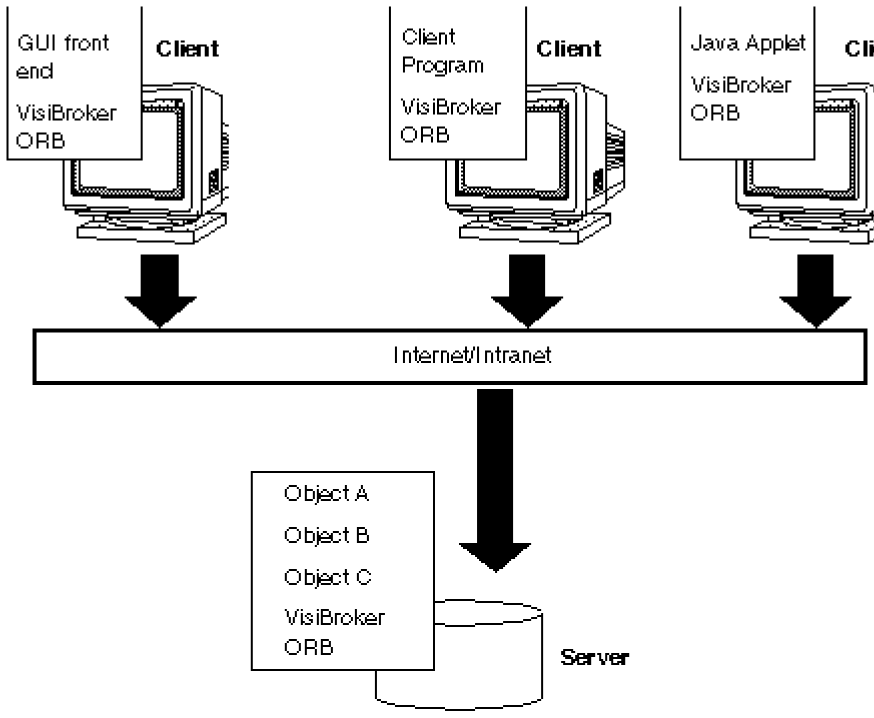
```
The balance in the account in $168.38.
```

## Deploying applications with VisiBroker

VisiBroker is also used in the deployment phase. This phase occurs when a developer has created client programs or server applications that have been tested and are ready for production. At this point a system administrator is ready to deploy the client programs on end-users' desktops or server applications on server-class machines.

For deployment, the VisiBroker ORB supports client programs on the front end. You must install the VisiBroker ORB on each machine that runs the client program. Clients (that make use of the VisiBroker ORB) on the same host share the VisiBroker ORB. The VisiBroker ORB also supports server applications on the middle tier. You must install the full VisiBroker ORB on each machine that runs the server application. Server applications or objects (that make use of the VisiBroker ORB) on the same server machine share the VisiBroker ORB. Clients may be GUI front ends, applets, or client programs. Server implementations contain the business logic on the middle tier.

**Figure 2.3** Client and server programs deployed with VisiBroker ORBs



## VisiBroker Applications

### Deploying applications

In order to deploy applications developed with VisiBroker, you must first set up a runtime environment on the host where the application is to be executed and ensure that the necessary support services are available on the local network.

The runtime environment required for applications developed with VisiBroker for C++ includes these components:

- The VisiBroker libraries, located in the `bin` sub-directory where the product is installed.
- The availability of the support services required by the application.

The VisiBroker ORB libraries must be installed on the host where the deployed application is to execute. The location of these libraries must be included in the PATH for the application's environment.

### Environment variables

If the deployed application is to use a Smart Agent (`osagent`) on a particular host, you must set the `OSAGENT_ADDR` environment variable before running the application. You can use the `ORBagentAddr` property as a command-line argument to specify a hostname or IP address. The table below lists the necessary command-line arguments.

If the deployed application is to use a particular UDP port when communicating with a Smart Agent, you must set the `OSAGENT_PORT` environment variable before running the application.

You can use the `ORBagentPort` (C++) command-line argument to specify the IP port number.

For more information about environment variables, see the Borland VisiBroker *Installation Guide*.

## Support service availability

A Smart Agent must be executing somewhere on the network where the deployed application is to be executed. Depending on the requirements of the application being deployed, you may need to ensure that other VisiBroker runtime support services are available, as well. These services include:

| Support services | Needed when: |
|---|---|
| Object Activation Daemon (`oad`) | A deployed application is a server that implements object which needs to be started on demand. |
| Interface Repository (`irep`) | A deployed application uses either the dynamic skeleton interface or dynamic implementation interface. See "Using Interface Repositories" for a description of these interfaces. |
| GateKeeper | A deployed application needs to execute in an environment that uses firewalls for network security. |

## Running the application

Before you attempt to run VisiBroker client programs or server implementations, you must first start the Smart Agent on at least one host in your local network. The Smart Agent is described in detail in "Starting the Smart Agent".

## Executing client applications

A client application is one that uses VisiBroker ORB objects, but does not offer any VisiBroker ORB objects of its own to other client applications.

The following table summarizes the command-line arguments that may be specified for a client application. These arguments also are applicable to servers.

| Options | Description |
|---|---|
| `-ORBagentAddr <hostname\|ip_address>` | Specifies the hostname or IP address of the host running the Smart Agent this client should use. If a Smart Agent is not found at the specified address or if this option is not specified, broadcast messages will be used to locate a Smart Agent. |
| `-ORBagentPort <port_number>` | Specifies the port number of the Smart Agent. This option can be useful if multiple VisiBroker ORB domains are required. If not specified, a default port number of `14000` will be used. |
| `-ORBbackcompat <0\|1>` | If set to 1, this option specifies that backward compatibility with VisiBroker for C++ version 2.0 should be provided. The default is `0`. |
| `-ORBbackdii <0\|1>` | If set to 1, this option specifies that support for the 1.0 IDL-to-C++ mapping should be provided. If set to `0` or not specified at all, the new 1.1 mapping will be used. The default setting is `0`. If `-ORBbackcompat` is set to `1`, this option will automatically be set to `1`. |
| `-ORBir_name <ir_name>` | Specifies the name of the Interface Repository to be accessed when the `Object::get_interface()` method is invoked on object implementations. |
| `-ORBir_ior <ior_string>` | Specifies the IOR of the Interface Repository to be accessed when the `Object::get_interface()` method is invoked on object implementations. |
| `-ORBnullstring <0\|1>` | If set to `1`, this option specifies that the VisiBroker ORB will allow C++ `NULL` strings to be streamed. The `NULL` strings will be marshalled as strings of length `0` opposed to the empty string ("") which is marshalled as a string of length `1`, with the sole character of `\0`. If set to `0`, attempts to marshal out a `NULL` string will throw `CORBA::BAD_PARAM`. Attempts to marshal in a `NULL` string will throw `CORBA::MARSHAL`. The default setting is `0`. If `-ORBbackcompat` is set to `1`, this option will automatically be set to `1`. |
| `-ORBrcvbufsize <buffer_size>` | Specifies the size of the TCP buffer (in bytes) used to receive responses. If not specified, a default buffer size will be used. This argument can be used to significantly impact performance or benchmark results. |
| `-ORBsendbufsize <buffer_size>` | Specifies the size of the TCP buffer (in bytes) used to send client requests. If not specified, a default buffer size will be used. This argument can be used to significantly impact performance or benchmark results. |
| `-ORBshmsize <size>` | Specifies the size of the send and receive segments (in bytes) in shared memory. If your client program and object implementation communicate via shared memory, you may use this option to enhance performance. This option is only supported on Windows platforms. |
| `-ORBtcpnodelay <0\|1>` | When set to `1`, it sets all sockets to immediately send requests. The default value of `0` allows sockets to send requests in batches as buffers fill. This argument can be used to significantly impact performance or benchmark results. |

3

# Programmer tools for C++

This chapter describes the programmer tools offered by VisiBroker for C++.

## VisiBroker for C++ Switches for Header Files

The following switches are used to point consumers of the header files to the proper code libraries.

### _VIS_STD

On platforms where VisiBroker for C++ supports development of both classical and standard C++ applications, defining `_VIS_STD` enables inclusion of the correct C++ header files for the C++ libraries in the VisiBroker for C++ header files. For developing standard C++ applications, use the `_VIS_STD` flag while compiling. For classical C++ application development, do not use this flag.

### _VIS_NOLIB

On Windows, a VisiBroker for C++ header file (`vdef.h`) automatically places the VisiBroker for C++ library search records in the object files. This is done using the `#pragma` comment for both MSVC and BCB compilers. Depending on certain other definition such as `_DEBUG`, `VISDEBUG` or `_VIS_STD`, appropriate library search records are selected. If this behavior is not required and VisiBroker for C++ library names are to be specified explicitly in the application link command, then `_VIS_NOLIB` should be defined. By default it is not defined.

## Arguments/Options

There is a set of arguments common to all VisiBroker programmer's tools and, in addition, each tool has its own arguments. The specific arguments and options for each tool are listed in the section for the tool. The general options are listed below.

### General options

The following options are common to all programmer tools:

| Option | Description |
|---|---|
| `-J<java option>` | Passes the java_option directly to the Java Virtual Machine. |
| `-VBJversion` | Prints the VisiBroker version. |
| `-VBJdebug` | Prints the VisiBroker debug information. |
| `-VBJclasspath` | Specifies the classpath, precedes the CLASSPATH environment variable. |
| `-VBJprop <name> [=<value>]` | Passes the name/value pair to the Java Virtual Machine. |
| `-VBJjavavm <jvmpath>` | Specifies the path of the Java Virtual Machine. |
| `-VBJaddJar <jarfile>` | Appends the jarfile to the CLASSPATH before executing the Java Virtual Machine. |

**Note**

On UNIX platforms, the `-J` option is only available with VisiBroker for Java on Solaris.

## General information

The syntax of the VisiBroker programming tools described in this chapter differs depending on whether you call them from a UNIX or a Windows environment. The UNIX version of each tool is listed first followed by the Windows version.

**UNIX**

To display the options of a command under UNIX, enter:

| Syntax | Example |
|---|---|
| `command name -\?` | idl2cpp -\? |

**Windows**

To display the options of a command under Windows, enter:

| Syntax | Example |
|---|---|
| `command name -?` | idl2cpp -? |

## idl2cpp

This command implements VisiBroker's IDL to C++ compiler, which generates client stubs and server skeleton code from an IDL file.

**Syntax**

```
idl2cpp [arguments] infile(s)
```

`idl2cpp` takes an IDL file as input and generates the corresponding C++ classes for the client and server side, client stubs, and server skeleton code.

The `infile` parameter represents the IDL file for which you wish C++ code to be generated and the arguments provide various controls over the resulting code.

**Example**

```
idl2cpp -hdr_suffix hx -server_ext _serv -no_tie -no_excep-spec bank.idl
```

**Windows** When linking implementations based on the stubs and skeletons `idl2cpp` generates, use the -`DSTRICT` preprocessor option. Otherwise, the linker may display an error message stating that a constructor is missing from `orb.lib`.

| Argument | Description |
|---|---|
| `-D, -defined foo[=bar]` | Defines a preprocessor macro `foo`, optionally with a value `bar`. To specify more than one preprocessor macro, use the `-D` option multiple times. For example: `-Dfoo=bar -Dhello=world` |
| `-H, -list_includes` | Prints the full paths of included files on the standard error output. The default is off. |
| `-I, -include <dir>` | Specifies an additional directory for `#include` searching. To specify more than one additional `#include` directory for searching, use the `-I` option multiple times. For example: `-I/home/include -I /app/include` |
| `-P, no_line_directives` | Suppresses the generation of line number information. The default is off. |
| `-U, -undefine foo` | Undefines a preprocessor macro `foo`. |
| `-client_ext <file_extension>` | Specifies the file extension to be used for client files that are generated. The default extension is `(_c)`. To generate client files without an extension, specify `none` as the value for <`file_extension`>. |
| `-[no_]back_compat_mapping` | In the current release this option does not do anything. It could change in the next release. |
| `_[no_]boa` | Specifies the generation of BOA compatible code. By default, this code is not generated. |
| `-[no_]comments` | Includes comments in the generated code. By default, the comments are displayed in the generated code. |
| `-[no_]idl_strict` | Specifies strict OMG standard interpretation of the IDL source. By default, the OMG standard interpretation is not used. |
| `-[no_]obj_wrapper` | Generates stubs and skeletons with object wrapper support. It also generates the base typed object wrapper from which all other object wrappers inherit, and a default object wrapper that performs the untyped object wrapper calls. When this option is not set, `idl2cpp` does not generate code for object wrappers. |
| `-[no_]preprocess` | Preprocesses the IDL file before parsing. The default value is set to on. |
| `-[no_]preprocess_only` | Stops parsing the IDL file after preprocess. This option causes the compiler to generate the result of the preprocess phase to stdout. The default is on. |
| `-[no_]pretty_print` | Generates the `_pretty_print` method. By default, this is set to on. |
| `-[no_]servant` | Specifies the generation of the server-side code. By default, the servant is generated. |
| `-[no_]stdstream` | Generates class stream operators with standard `iostream` classes in their signature. The default is on. |
| `-[no_]tie` | Generates the `_tie` template classes. By default, `_tie` classes are generated. |
| `-[no_]warn_all` | Suppresses all warnings. The default is off. |
| `-[no_]warn_missing_define` | Warns if any forward declared names were never defined. The default is on. |
| `-[no_]warn_unrecognized_pragmas` | Generates a warning if a `#pragma` is not recognized. |
| -`corba_inc <filename>` | Causes the `#include <filename>` directive to be inserted in generated code instead of the usual`#include <corba.h>` directive. By default, `#include <corba.h>` is inserted into generated code. |

| Argument | Description |
|---|---|
| `-[no_]examples` | Specifies the generation of sample implementations. By default, the sample implementations are not generated. |
| `-excep_spec` | Generates exception specifications for methods. By default, exception specifications are not generated. |
| **Windows:** -export <tag> | Defines a tag name to be inserted into every client-side declaration (class, function, etc.) that is generated. Specifying`-export _MY_TAG` when invoking `idl2cpp` results in a class definition like this: `class _MY_TAG Bank{...}` instead of `class Bank {...}` By default, no tag names for client-side declarations are generated. |
| **Windows:** -export_skel <tag> | Defines a tag name to be inserted into just the server-side declarations that are generated. Specifying `-export _MY_TAG` when invoking `idl2cpp` results in a class definition like this: `class _MY_TAG POA_Bank{...}` instead of `class POA_Bank {...}` By default, no tag names for server-side declarations are generated. |
| `-gen_include_files` | Specifies the generation of code for #include files. By default, this code is not generated. |
| `-h, -help, -usage, -?` | Specifies that help information be printed. |
| `-hdr_suffix <string>` | Specifies the header filename extension. The default is `.hh`. |
| `-impl_inherit` | Generates implementation inheritance. The default is off. |
| `-list_files` | Specifies that files written during code generation be listed. By default, this list is not created. |
| `-map_keyword <keywrd> <map>` | Adds <keywrd> as a keyword and associates with it the mapping indicated. Any IDL identifier that conflicts with <keywrd> will be mapped in C++ to <map>. This prevents clashes between keywords and names used in C++ code. All C++ keywords have default mappings—they do not need to be specified using this option. |
| `-namespace` | Implements modules as namespaces. The default is off. |
| `-root_dir <path>` | Specifies the directory where the generated code is to be written. By default, the code is written to the current directory. |
| `-server_ext <file_extension>` | Specifies the file extension to be used for server files that are generated. The default extension is `(_s)`. To generate server files without an extension, specify **none** as the value for <`file_extension`>. |
| `-src_suffix <string>` | Specifies the source filename extension. The default is `.cc`. |
| `-target <compiler>` | Specifies the compiler used to generate the C++ code. The default compiler used is `Solaris`. |
| `-type_code_info` | Enables the generation of type code information needed for client programs that intend to use the Dynamic Invocation Interface. By default, type code information is not generated. |
| `-version` | Displays the software version number of VisiBroker. |
| -corba_style | Requires `-type_code_info` flag. Generates pointer insertion/ extraction into/from `CORBA::Any`. By default, it is off. |
| -corba_style_tie | Requires `-tie` flag. Generate a tie class within same scope as skeleton class. By default, it is off. |
| `file1 [file2] ...` | Specifies one or more files to be processed, or "-" for stdin. |
| `CPP` | The orb.idl has conditional definitions which are specific to either VisiBroker for C++ or VisiBroker for Java. Therefore, if you want to include the `orb.idl` in your IDL, you must turn on the VisiBroker for C++-specific definitions using the `CPP` macro. For example, use the following: `idl2cpp -D` **`CPP`** `test.idl`. Alternately, you may put the following line at the top of your IDL file: `#define CPP` |

## idl2ir

This command allows you to populate an interface repository with objects defined in an Interface Definition Language source file.

**Syntax**

```
idl2ir [-ir <IR_name>] [-replace] <filename>.idl [<filename2>.idl ...]
```

**Example**

```
idl2ir -ir my_repository -replace bank/Bank.idl
```

**Description**

The `idl2ir` command takes the name of an IDL file as input, binds itself to an interface repository server, and populates the repository with the IDL constructs contained in <filename>.idl. If the `-replace` option is specified, if the repository already contains an item with the same name as an item in the IDL file, the old item is replaced.

**Note**

The `idl2ir` command does not handle anonymous arrays or sequences properly. To work around this problem, `typedefs` must be used for all sequences and arrays.

| Option | Description |
|---|---|
| `-D, -define foo[=bar]` | Defines a preprocessor macro `foo`, optionally with a value `bar`. |
| `-I, -include <dir>` | Specifies an additional directory for `#include` searching. |
| `-P, no_line_directives` | Suppresses the generation of line number information. The default is off; line numbering is not suppressed. |
| `-H, _list_includes` | Prints the names of included files on the standard error output. The default is off. |
| `-U, -undefine foo` | Undefines a preprocessor macro `foo`. |
| `-[no_]back_compat_mapping` | Specifies the use of mapping that is backward compatible with VisiBroker 3.x. |
| `-{no_]idl_strict` | Specifies strict OMG standard interpretation of the IDL source. By default, the OMG standard interpretation is not used. |
| `-[no_]preprocess` | Preprocesses the IDL file before parsing. The default value is set to on. |
| `-[no_]preprocess_only` | Stops parsing the IDL file after preprocess. This option causes the compiler to generate the result of the preprocess phase to stdout. The default is on. |
| `-[no_]warn_all` | Suppresses all warnings. The default is off. |
| `-[no_]warn_unrecognized_pragmas` | Generates a warning if a `#pragma` is not recognized. |
| `-deep` | Specifies deep (rather than shallow) merges. If you specify `-deep`, only differences between the new contents and the existing contents will be merged. In a `-shallow` merge, all existing content is replaced with new content if the new content defines the same names. The default is `off`. |
| `-h, -help, -usage, -?` | Prints help information. |
| `-irep <name>` | Specifies the instance name of the interface repository to which `idl2ir` will attempt to bind. If no name is specified, `idl2ir` will bind itself to the interface repository server found in the current domain. The current domain is defined by the `OSAGENT_PORT` environment variable. |
| `-replace` | Replaces definitions instead of updating them. |
| `-version` | Displays the software version number of VisiBroker. |
| `file1 [file2] ...` | Specifies one or more files to be processed, or “-” for stdin. |

## ir2idl

This command allows you to create an Interface Definition Language (IDL) source file with objects from an interface repository.

**Syntax**

```
ir2idl [options]
```

**Example**

The following example dumps the contents of the IR named `foo` into the file named `foo.idl`:

```
ir2idl -irep foo -o foo.idl
```

**Description**

The ir2idl command extracts the contents of an IR and prints it out as IDL.

**Options**

The following options are available for ir2idl.

| Option | Description |
|---|---|
| `-irep <irep name>` | Specifies the name of the interface repository. |
| `-o, <file>` | Specifies the name of the output file, or "-" for stdout. |
| `-strict` | Specifies strict adherence to OMG-standard code generation. The default is on. The compiler will complain upon occurrences of Borland-proprietary syntax extensions in input IDL. |
| `-version` | Displays or prints out the version of VisiBroker that you are currently running |
| `-h, -help, -usage, -?` | Prints help information. |

## idl2wsc

idl2wsc generates C++ code similar to Axis C++ v1.5 WSDL2Ws Server side generated code and it also generates the necessary CORBA calls to the CORBA server. This constitutes the C++ Web Services CORBA Bridge Code.

Given an IDL name "Foo.idl" by default the idl2wsc tool will generate the files "Foo_ws_s.cc, Foo_ws_c.hh, Foo.wsdl, corba.wsdl and Foo.wsdd". Note that the "*.cc, *.hh, *.wsdl" files should not be modified. The generated WSDD file can be modified by the user to point to the compiled shared library that will be loaded by the C++ Web Services Run-time Library.

The options available to idl2cpp are also available to idl2wsc. In addition to the idl2cpp options, the following are specific to idl2wsc

| Option | Description |
|---|---|
| `-encoding_wsi_only` | Generate specific WS–I encodings only. Defaults to OFF |
| `-encoding_soap_only` | Generate specific SOAP encodings only. Defaults to OFF |
| `-gen_cpp_bridge` | Generate VisiBroker for C++ bridge code. Defaults to OFF. |

### Usage of idl2wsc

Before passing any IDL file to idl2wsc to generate the C++ bridge code, you will have to pass the IDL file to idl2cpp to generate the CORBA stub code. Note that you should apply the same idl2cpp options that you use to generate the CORBA stub code to the idl2wsc tool because the idl2wsc tool references names of files and/or signatures that should have been generated by idl2cpp.

Note that any changes to the idl2cpp generated code or to a new version of "\include\ vbws.h" requires a recompilation of the idl2wsc-generated code.

## Limitation of idl2wsc

Note that the Axis C++ v1.5 WSDL2WS tool does not support a WSDL file that defines more than one "portType" and it will only generate only one of the "portTypes" defined. This itself is a limitation of Axis C++ v1.5 and therefore an IDL file containing more than one interface is not supported.

4

# IDL to C++ mapping

This section discusses the IDL to C++ language mapping provided by the VisiBroker for C++ `idl2cpp` compiler, which strictly complies with the CORBA C++ language mapping specification.

## Primitive data types

The basic data types provided by the Interface Definition Language are summarized in the table below. Due to hardware differences between platforms, some of the IDL primitive data types have a definition that is marked "platform dependent." On a platform that has 64–bit integral representations, for example, the `g` type, would still be only 32 bits. You should refer to the included file **orbtypes.h** for the exact mapping of these primitive data types for your particular platform.

| IDL type | VisiBroker type | C++ definition |
|---|---|---|
| short | CORBA::Short | short |
| long | CORBA::Long | platform dependent |
| unsigned short | CORBA::UShort | unsigned short |
| unsigned long | CORBA::ULong | unsigned long |
| float | CORBA::Float | float |
| double | CORBA::Double | double |
| char | CORBA::Char | char |
| wchar | CORBA::WChar | wchar_t |
| boolean | CORBA::Boolean | unsigned char |
| octet | CORBA::Octet | unsigned char |
| long long | CORBA::LongLong | platform dependent |
| ulong long | CORBA::ULongLong | platform dependent |

#### Caution

The IDL `boolean` type is defined by the CORBA specification to have only one of two values: `1` or `0`. Using other values for a `boolean` will result in undefined behavior.

## Strings

Both bounded and unbounded String types in IDL are mapped to the C++ type `char *`.

**Note**

All CORBA string types are null-terminated.

To ensure that your applications use the same memory management facilities as VisiBroker does, use the following functions to dynamically allocate and de-allocate strings:

```
class CORBA

{
   ...
   static char *string_alloc(CORBA::ULong len);
   static void string_free(char *data);
   ...
};
```

CORBA::char *string_alloc(CORBA::ULong len);

Dynamically allocates a string and returns a pointer to it. Returns a `NULL` pointer if the allocation fails.

| Parameter | Description |
|---|---|
| `len` | The length specified by the `len` parameter need not include the `NULL` terminator. |

CORBA::void *string_free(char *data);

Releases the memory associated with a string that was allocated with `CORBA::string_alloc`.

| Parameter | Description |
|---|---|
| `data` | Pointer to a string that was allocated with `CORBA::string_alloc`. |

### String_var Class

Whenever it maps an IDL `string` to a`char *`, the IDL compiler also generates a `String_var` class that contains a pointer to the memory allocated to hold the string. When a `String_var` object is destroyed or goes out of scope, the memory allocated to the string is automatically freed.

Following are the members and methods in the `String_var` class:

```
class CORBA      {

   class String_var {
         protected:
               char   *_p;
               ...
         public:
               String_var();
               String_var(char *p);
               ~String_var();
               String_var&      operator=(const char *p);
               String_var&      operator=(char *p);
               String_var&      operator=(const String_var& s);
               operator const char *() const;
               operator char *();



               char &operator[](CORBA::ULong index);
```

```
                char operator[](CORBA::ULong index) const;
                friend ostream&         operator<<(ostream&, const
String_var&);
                inline friend Boolean         operator==(const String_var& s1,
                                const String_var& s2);
                ...
    };
    ...
};
```

## Constants

IDL constants defined outside of any interface specification are mapped directly to a C++ constant declaration. For example:

This code sample shows the top-level definitions in IDL.

```
const string          str_example = "this is an example";

const long            long_example = 100;
const boolean         bool_example = TRUE;
```

This code sample shows the resulting C++ code for constants.

```
const char *              str_example = "this is an example";

const CORBA::Long         long_example = 100;
const CORBA::Boolean      bool_example = 1;
```

IDL constants defined within an interface specification are declared in the C++ include file and assigned values in the C++ source file. For example:

This code sample shows the IDL definitions from the example.idl file.

```
interface example {

   const string       str_example = "this is an example";
   const long         long_example = 100;
   const boolean      bool_example = TRUE;
};
```

This code sample shows the C++ code generated to the example_client.hh file.

```
class example :: public virtual CORBA::Object

{
   ...
   static const char *str_example;  /* "this is an example" */
   static const CORBA::Long         long_example; /* 100 */
   static const CORBA::Boolean      bool_example; /* 1 */
   ...
};
```

This code sample shows the C++ code generated to the example_client.cc file.

```
const char *example::str_example = "this is an example";

const CORBA::Long            example::long_example = 100;
const CORBA::Boolean         example::bool_example = 1;
```

### Special cases involving constants

Under some circumstances, the IDL compiler must generate C++ code that contains the value of an IDL constant rather than the name of the constant. For example, in the following code samples, the value of the constant `length` must be generated for the typedef `V` to allow the C++ code to compile properly.

The code sample shows the definition of an IDL constant with a value.

```
// IDL
interface foo {
   const long length = 10;
   typedef long V[length];
};
```

This code sample shows the generation of an IDL constant's value in C++.

```
class foo : public virtual CORBA::Object
{
   const CORBA::Long length;
   typedef CORBA::Long V[10];
};
```

## Enumerations

Enumerations in IDL map directly to C++ enumerations. For example:

```
// IDL
enum enum_type {
   first,
   second,
   third
};
```

This code sample shows the enumerations in IDL map directly to C++ enums.

```
// C++ code
enum enum_type {
   first,
   second,
   third
};
```

## Type definitions

Type definitions in IDL map directly to C++ type definitions. If the original IDL type definition maps to several C++ types, the IDL compiler generates the corresponding aliases for each type in C++. For example:

```
// IDL
typedef octet                example_octet;
typedef enum enum_values {
   first,
   second,
   third
} enum_example;
```

This code sample shows the mapping of simple type definitions from IDL to C++.

```
// C++
typedef octet                example_octet;
enum enum_values {
   first,
   second,
   third
};
typedef enum_values enum_example;
```

The following code samples provide other type definition mapping examples.

This code sample shows the IDL typedef of an interface.

```
// IDL
interface A1;
typedef A1 A2;
```

This code sample shows the mapping the IDL interface type definition in C++.

```
// C++
class A1;
typedef A1 *A1_ptr;
typedef A1_ptr A1Ref;
class A1_var;
typedef A1 A2;
typedef A1_ptr A2_ptr;
typedef A1Ref A2Ref;
typedef A1_var A2_var;
```

This code sample shows the IDL typedef of a sequence.

```
// IDL
typedef sequence<long> S1;
typedef S1 S2;
```

This code sample shows the mapping the IDL sequence type definition to C++.

```
// C++
class S1;
typedef S1 *S1_ptr;
typedef S1_ptr S1Ref;
class S1_var;
typedef S1 S2;
typedef S1_ptr S2_ptr;
typedef S1Ref S2Ref;
typedef S1_var S2_var;
```

## Modules

The OMG IDL to C++ language mapping specifies that each IDL `module` be mapped to a C++ `namespace` with the same name. However, few compilers currently support the use of `namespaces`. Therefore, VisiBroker currently supports module to class mapping only. The code samples below show how VisiBroker's IDL compiler maps a `module definition to` a `class`.

This code sample shows the IDL module definition.

```
// IDL
module ABC
{
   ...
};
```

This code sample shows the generated C++ class.

```
// C++
class ABC
{
   ...
};
```

## Complex data types

In this section, we discuss how the following complex data types are mapped from IDL to C++:

- `Any` type
- `string` type, bounded or unbounded
- `sequence` type, bounded or unbounded
- Object reference
- Other `structures` or `unions` that contain a variable-length member
- `array` with variable-length elements
- `typedef` with variable-length elements.

| IDL type | C++ mapping |
|---|---|
| `struct` (fixed length) | `struct` and `_var` class |
| `struct` (variable length) | `struct` and `_var` class<br>(variable length members are declared with their respective `T_var` class) |
| `union` | `class` and `_var` class |
| `sequence` | `class` and `_var` class |
| `array` | `array`, `array_slice`, `array_forany`, and `array_var` |

## Structures

### Fixed-length structures

For each fixed-length IDL structure mapped to C++, VisiBroker's IDL compiler generates a structure as well as a `_var` class for the structure. The code samples below show how this is done. For more information on the `_var` class, see "<class_name>_var" in the *VisiBroker for C++ API Reference*.

This code sample shows the fixed-length structure definition in IDL.

```
// IDL
struct example {
   short a;
   long b;
};
```

This code sample shows the mapping of a fixed-length IDL structure to C++.

```
// C++
struct example {
   CORBA::Short a;
   CORBA::Long b;
};
class example_var
{
   ...
   private:
        example *_ptr;
};
```

**Using fixed-length structures**

When accessing fields of the `_var` class, you must always use the `->` operator. For example, the code sample below shows that to access the fields of the `_var` class `ex2`, the `->` operator must always be used. When `ex2` goes out of scope, the memory allocated to it will be freed automatically.

This code sample shows the use of the example structure and the example_var class.

```
// Declare an example struct and initialize its fields.
example ex1 = { 2, 5 };
// Declare a _var class and assign it to a newly created example structure.
// The _var points to an allocated struct with un-initialized fields.
example_var ex2 = new example;
// Initialize the fields of ex2 from ex1
ex2->a = ex1.b;
```

## Variable length structures

The C++ code generated when a structure contains variable-length members is different than when the structure is of fixed length. For example, the code samples below show what would happen if in the `example` structure first described previously where the `long` member were replaced with a `string` and an object reference were added, so that `example` became a variable-length structure.

This code sample shows the variable length structure definitions in IDL.

```
// IDL
interface ABC {
   ...
};
struct vexample {
   short      a;
   ABC      c;
   string      name;
};
```

This code sample shows the mapping of a variable-length structure to C++.

```
// C++
struct vexample {
   CORBA::Short      a;
   ABC_var      c;
   CORBA::String_var      name;
   vexample&    operator=(const vexample& s);
};
class vexample_var {
   ...
};
```

Notice how the `ABC` object reference is mapped to an `ABC_var` class. In a similar fashion, the string `name` is mapped to a `CORBA::String_var` class. In addition, an assignment operator is generated for variable-length structures.

## Memory management for structures

The use of `_var` classes in variable-length structures ensures that memory allocated to the variable-length members is managed transparently.

– If a structure goes out of scope, all memory associated with variable-length members is freed automatically.

– If a structure is initialized or assigned and then re-initialized or reassigned, the memory associated with the original data is always freed.

– When a variable-length member is assigned to an object reference, a copy is always made of the object reference. If a variable-length member is assigned to a pointer, no copying takes place.

## Unions

Each IDL `union` is mapped to a C++ class with methods for setting and retrieving the value of the data members. Every member in the IDL union is mapped to a set of functions that serve as accessors and mutators. A *mutator* function sets the value of the data member. An *accessor* function returns the data in the data member.

A special, pre-defined data member, named`_d,` of the `discriminant` type is also generated. The value of this discriminant is not set when the union is first created, so an application must set it before using the `union`. Setting any data member using one of the methods provided automatically sets the discriminant. A special accessor function, `_d()`, provides access to the discriminant.

For example, the code samples below show how a union, `example_union`, would be generated in C++:

This code sample shows the IDL union containing a struct.

```
// IDL
struct example_struct
{
   long abc;
};
union example_union switch(long)
{
   case 1: long       x; // a primitive data type
   case 2: string       y; // a simple data type
   case 3: example_struct       z; // a complex data type
};
```

This code sample shows the mapping of an IDL union to a C++ class.

```
// C++
struct example_struct
{
         CORBA::Long       abc;
};
class example_union
{
   private:
         CORBA::Long       _disc;
         CORBA::Long       _x;
         CORBA::String_var       _y;
         example_struct       _z;
   public:
         example_union();
         ~example_union();
         example_union(const example_union& obj);
         example_union&    operator=(const example_union& obj);
         void    x(const CORBA::Long val);
         const CORBA::Long    x() const;
         void    y(char *val);
         void    y(const char *val);
         void    y(const CORBA::String_var& val);
         const char    *y() const;
         void z(const example_struct& val);
         const example_struct&    z() const;
```

```
            example_struct&    z();
            CORBA::Long _d();
            void _d(CORBA::Long);
            ...
};
```

The table below describes some of the methods in the `example_union` class.

| Method | Description |
|---|---|
| `_d()` | This Method returns the value of the discriminator. |
| `_d(CORBA::Long)` | This method is used for setting the value of the discriminator. (In the case of the example, the discriminator is of type `long`). Note that based on the data type of the discriminator, the input argument's type will be different. |
| `example_union()` | The default constructor sets the discriminant to the default value but does not initialize any of the other data members. |
| `example_union(const example_union& obj)` | The copy constructor performs a deep copy of the source object. |
| `~example_union()` | The destructor frees all memory owned by the union. |
| `operator=(const example_union& obj)` | The assignment operator performs a deep copy, releasing old storage, if necessary. |

### Managed types for unions

In addition to the `example_union` class shown in the following code sample, an `example_union_var` class would also be generated. See “<class_name>_var” in the *VisiBroker for C++ API Reference* for details on the `_var` classes.

### Memory management for unions

Here are some important points to remember about memory management of complex data types within a union:

- When you use an accessor method to set the value of a data member, a deep copy is performed. You should pass parameters to accessor methods by value for smaller types or by constant reference for larger types.
- When you set a data member using an accessor method, any memory previously associated with that member is freed. If the member being assigned is an object reference, the reference count of that object is incremented before the accessor method returns.
- A `char *` accessor method frees any storage before ownership of the passed pointer is assumed.
- Both `const char *` and `String_var` accessor methods free any old memory before the new parameter's storage is copied.
- Accessor methods for array data members return a pointer to the array slice. For more information, see “Array slices”.

## Sequences

IDL sequences, both bounded and unbounded, are mapped to a C++ class that has a current length and a maximum length. The maximum length of a bounded sequence is defined by the sequence's type. Unbounded sequences can specify their maximum length when their C++ constructor is called. The current length can be modified programmatically. The code samples below show how an IDL sequence is mapped to a C++ class with accessor methods.

**Note**

When the length of an unbounded sequence exceeds the maximum length you specify, VisiBroker transparently allocates a larger buffer, copies the old buffer to the new buffer, and frees the memory allocated to the old buffer. However, no attempt is made to free unused memory if the maximum length decreases.

This code sample shows the IDL unbounded sequence.

```
// IDL
typedef sequence<long> LongSeq;
```

This code sample shows the mapping of an IDL unbounded sequence to a C++ class.

```
// C++
class LongSeq
{
   public:
         LongSeq(CORBA::ULong max=0);
         LongSeq(CORBA::ULong max=0, CORBA::ULong length,
                           CORBA::Long *data, CORBA::Boolean release = 0);
         LongSeq(const LongSeq&);
         ~LongSeq();
         LongSeq&          operator=(const LongSeq&);
         CORBA::ULong          maximum() const;
         void        length(CORBA::ULong len);
         CORBA::ULong          length() const;
         const CORBA::ULong&          operator[](CORBA::ULong index) const;
         ...
         static LongSeq          *_duplicate(LongSeq* ptr);
         static void          _release(LongSeq *ptr);
         static CORBA::Long          *allocbuf(CORBA::ULong nelems);
         static void          freebuf(CORBA::Long *data);
   private:
         CORBA::Long      * _contents;
         CORBA::ULong       _count;
         CORBA::ULong       _num_allocated;
         CORBA::Boolean       _release_flag;
         CORBA::Long       _ref_count;
};
```

| Method | Description |
|---|---|
| `LongSeq(CORBA::ULong max=0)` | The constructor for an unbounded sequence takes a maximum length as an argument. Bounded sequences have a defined maximum length. |
| `LongSeq(CORBA::ULong max=0,` `CORBA::ULong length,` `CORBA::Long *data,` `CORBA::Boolean release=0)` | This constructor allows you to set the maximum length, the current length, a pointer to the data buffer associated and a release flag. If `release` is not zero, VisiBroker will free memory associated with the data buffer when increasing the size of the sequence. If `release` is zero, the old data buffer's memory is not freed. Bounded sequences have all of these parameters except for `max`. |
| `LongSeq(const LongSeq&)` | The copy constructor performs a deep copy of the source object. |
| `~LongSeq();` | The destructor frees all memory owned by the sequence only if the release flag had a non-zero value when constructed. |
| `operator=(const LongSeq&j)` | The assignment operator performs a deep copy, then releases old storage, if necessary. |
| `maximum()` | Returns the size of the sequence. |
| `length()` | Two methods are defined for setting and returning the length of the sequence. |
| `operator[]()` | Two indexing operators are provided for accessing an element within a sequence. One operator allows the element to be modified and one allows only read access to the element. |

| Method | Description |
|---|---|
| `_release()` | Releases the sequence. If the constructor's release flag was non-zero when the object was created and the sequence element type is a string or object reference, each element is released before the buffer is released. |
| `allocbuf()` `freebuf()` | You should use these two static methods to allocate or free any memory used by a sequence. |

## Managed types for sequences

In addition to the `LongSeq` class shown in the code sample below, a `LongSeq_var` class is also generated. See "<class_name>_var" in the *VisiBroker for C++ API Reference* for details on the classes. In addition to the usual methods, there are two indexing methods defined for sequences.

```
CORBA::Long& operator[](CORBA::ULong index);

const CORBA::Long& operator[](CORBA::ULong index) const;
```

## Memory management for sequences

You should carefully consider the memory management issues listed below. The code sample below contains sample C++ code that illustrates these points.

- If the release flag was set to a non-zero value when the sequence was created, the sequence assumes management of the user's memory. When an element is assigned, the old memory is freed before ownership of the memory on the right-hand side of the expression is assumed.
- If the release flag was set to a non-zero value when a sequence containing strings or object references was created, each element is released before the sequence's contents buffer is released and the object is destroyed.
- Memory management errors may occur if you assign a sequence element using the `[]` operator unless the release flag was set to one.
- Do not use sequences created with the release flag set to zero as input/output parameters because memory management errors in the object server may result.
- Always use `allocbuf` and `freebuf` to create and free storage used with sequences.

This code sample shows the IDL specification for an unbounded sequence.

```
// IDL

typedef sequence<string, 3> String_seq;
```

This code sample shows is an example of memory management with two bounded sequences.

```
// C++

char *static_array[] = {"1", "2", "3"};
char *dynamic_array = StringSeq::allocbuf(3);
```

// Create a sequence, release flag is set to **FALSE** by default

```
StringSeq static_seq(3, static_array);
// Create another sequence, release flag set to TRUE
StringSeq dynamic_seq(3, dynamic_array, 1);
```

static_seq[1] = "1"; // old memory **not** freed, no copying occurschar *str = string_alloc(2);

```
dynamic_seq[1] = str;              // old memory is freed, no copying occurs
```

## Arrays

IDL arrays are mapped to C++ arrays, which can be statically initialized. If the array elements are strings or object references, the elements of the C++ array are of type `_var`. The following code samples show three arrays with different element types.

This code sample shows the IDL array definitions.

```
// IDL

interface Intf
{
   ...
};
typedef long L[10];
typedef string S[10];
typedef Intf A[10];
```

This code sample shows the mapping of IDL arrays to C++ arrays.

```
// C++

typedef CORBA::Long L[10];
typedef CORBA::String_var S[10];
typedef Intf_var A[10];
```

The use of the managed type `_var` for strings and object references allows memory to be managed transparently when array elements are assigned.

### Array slices

The array`_slice` type is used when passing parameters for multi-dimensional arrays. VisiBroker's IDL compiler also generates a `_slice` type for arrays that contains all but the first dimension of the array. The array `_slice` type provides a convenient way to pass and return parameters. The following code samples show two examples of the `_slice` type.

This code sample shows the IDL definition of multi-dimensional arrays.

```
// IDL

typedef long L[10];
typedef string str[1][2][3];
```

This code sample shows the generation of the _slice type.

```
// C++
typedef CORBA::Long L_slice;
typedef CORBA::String_var str_slice[2][3];
```

### Managed types for arrays

In addition to generating a C++ array for IDL arrays, VisiBroker's IDL compiler will also generate a `_var` class. This class offers some additional features for array.

– `operator[]` is overloaded to provide intuitive access to array elements.

– Constructor and assignment operator are provided that take a pointer to an array `_slice` object as an argument.

This code sample shows the IDL definition of an array.

```
// IDL

typedef long L[10];
```

This code sample shows the _var class generated for arrays.

```
// C++
class L_var
{
   public:
         L_var();
         L_var(L_slice *slice);
         L_var(const L_var& var);
         ~L_var();
         L_var&    operator=(L_slice *slice);
         L_var&    operator=(const L_var& var);
         CORBA::Long&    operator[](CORBA::ULong index);
         operator L_slice *();
         operator L &() const;
         ...
   private:
         L_slice   *_ptr;
};
```

## Type-safe arrays

A special _forany class is generated to handle arrays with elements mapped to the type any. As with the _var class, the _forany class allows you to access the underlying array type. The _forany class does not release any memory upon destruction because the _any type maintains ownership of the memory. The _forany class is not implemented as a typedef because it must be distinguishable from other types if overloading is to function properly.

This code sample shows the IDL array definition.

```
// IDL

typedef long L[10];
```

This code sample shows _for any class generated for an IDL array.

```
// C++

class L_forany
{
     public:
          L_forany();
          L_forany(L_slice *slice);
          ~L_forany();
          CORBA::Long&    operator[](CORBA::ULong index);
          const CORBA::Long&    operator[](CORBA::ULong index) const;
          operator L_slice *();
          operator L &() const;
          operator const L & () const;
          operator const L& () const;
          L_forany& operator=(const L_forany obj);
          ...
     private:
          L_slice   *_ptr;
};
```

## Memory management for arrays

VisiBroker's IDL compiler generates four functions for allocating, duplicating, copying, and releasing the memory associated with arrays. These functions allow the VisiBroker ORB to manage memory without having to override the new and delete operators.

This code sample shows the IDL array definition.

```
// IDL

typedef long L[10];
```

This code sample shows the methods generated for allocating and releasing array memory.

```
// C++
inline L_slice *L_alloc();
// Dynamically allocates array. Returns
// NULL on failure.
inline void L_free(L_slice *data);
// Releases array memory allocated with
// L_alloc.
inline void L_copy(L:slice *_to, L_slice *_from)
//Copies the contents of the _from array to the _to array
inline L_slice *L_dup(const L_slice *_date)
//Returns a new copy of _date array
```

## Principal

A `Principal` represents information about client applications that are making operation requests on an `object` implementation. The IDL interface of `Principal` does not define any operations. The `Principal` is implemented as a sequence of octets. The `Principal` is set by the client application and checked by the VisiBroker ORB implementation. VisiBroker for C++ treats the `Principal` as an opaque type and its contents are never examined by the VisiBroker ORB.

## Valuetypes

An IDL valuetype is mapped to a C++ class with the same name as the IDL valuetype. This class is an abstract base class with pure virtual accessor and modifier functions corresponding to the state members of the valuetype and pure virtual functions corresponding to the operations of valuetype.

A C++ class whose name is formed by adding an "`OBV_`" to the fully scoped name of the `valuetype` provides default implementations for the accessors and modifiers of the abstract base class.

Applications are responsible for the creation of valuetype instances. After creation, these applications deal with those instances using only pointers. Unlike object references which map to C++ `_ptr types` that may be implemented either as actual C++ pointers or as C++ pointer-like objects, handles to C++ valuetype instances are actual C++ pointers. This helps to distinguish them from object references.

Unlike mapping for interfaces, reference counting for `valuetype` must be implemented by the instance of the valuetypes. The `_var` type for a valuetype automates the reference counting. The code sample below illustrates these features.

```
 valuetype Example {

  Short op1();
 Long  op2( in Example x );
 Private short val1;
 Public long  val2;
};
```

The code sample below shows the C++ mapping of the IDL definition for the following three classes.

```
class  Example  : public virtual CORBA::ValueBase   {
 public:
    virtual CORBA::Short op1() = 0;
    virtual CORBA::Long op2(Example_ptr _x) = 0;
 // pure virtual getter/setters for all public state
 // These accessors are just like C++ union members since
 // by reference accessors allow read/write access
 virtual void val2(const CORBA::Long _val2) = 0;
 virtual const CORBA::Long val2() const = 0;
protected:
 Example() {}
 virtual ~Example() {}
```

```
    virtual void val1(const CORBA::Short _val1) = 0;
    virtual const CORBA::Short val1() const = 0;
       private:
    void operator=(const Example&);
    };
   class OBV_Example: public virtual Example{
    public:
    virtual void val2(const CORBA::Long _val2) {
    _obv_val2 = _val2;
    }
    virtual const CORBA::Long val2() const {
    return _obv_val2;
    }
   protected:
    virtual void val1(const CORBA::Short _val1) {
    _obv_val1 = _val1;
    }
    virtual const CORBA::Short val1() const {
    return _obv_val1;   }
    OBV_Example() {}
    virtual ~OBV_Example() {}
    OBV_Example(const CORBA::Short _val1,
         const CORBA::Long _val2) {
       _obv_val1 = _val1;
       _obv_val2 = _val2;
    }
    CORBA::Short _obv_val1;
    CORBA::Long _obv_val2;
    };
   class Example_init : public CORBA::ValueFactoryBase {
    };
```

The `_init` class provides a way to implement a factory for the valuetypes. Since valuetypes are passed by value over the wire, the receiving end of a streamed out valuetype usually implements a factory to create a valuetype instance from the stream. Both the server and the client should implement it if there is a possibility of receiving a valuetype over the stream. The `_init` class, as shown in the following code sample, which must also implement `create_for_unmarshal` that returns a `CORBA::ValueBase *`.

This code sample shows the -init class example.

```
   class Example_init_impl: public Example_init{

   public:
      Example_init; _impl();
      virtual ~Example_init();
   CORBA::ValueBase * create_for_unmarshal() {
    ...// return an Example_ptr
    }
   };
```

A valuetype can derive from other valuetypes as follows:

This code sample shows the IDL for the valuetype derived from other valuetypes.

```
   valuetype DerivedExample: Example{

    Short op3();
   };
```

The C++ interfaces for the `DerivedExample` class are as follows:

```
   // IDL valuetype: DerivedExample

   class  DerivedExample  : public virtual Example   {
    public:
      virtual CORBA::Short op3() = 0;
```

```
 protected:
   DerivedExample() {}
   virtual ~DerivedExample() {}
 private:
   void operator=(const DerivedExample&);
};
class OBV_DerivedExample: public virtual DerivedExample, public virtual
OBV_Example{
 protected:
   OBV_DerivedExample() {}
   virtual ~OBV_DerivedExample() {}
};
class DerivedExample_init : public CORBA::ValueFactoryBase { };
```

A derived valuetype can be truncated to the base valuetype as shown in the following code sample. This is required if the receiving end of the stream does not know how to construct a derived valuetype but can construct only the base valuetype.

This code sample shows the truncated derived valuetype.

```
valuetype DerivedExample : truncatable Example { };
```

The mapping is similar to regular derived valuetypes except that extra information is added to the `Type` information of the `DerivedExample` class to indicate the truncatability to the base class Example.

A valuetype can not derive from an interface but it can support one or more interfaces by providing all the operations of the interfaces. An IDL keyword, **supports,** is introduced for this purpose.

This code sample shows the IDL keyword support for the derived valuetype.

```
interface myInterface{
long op5();
};
valuetype IderivedExample supports myInterface {
 Short op6();
};
```

The C++ mapping for this will be as follows:

This code sample shows the C++ for the derived valuetype.

```
// IDL valuetype: DerivedExample
class  IderivedExample  : public virtual CORBA::ValueBase   {
 public:
   virtual CORBA::Short op6() = 0;
   virtual CORBA::Long op5() = 0;
 protected:
   IderivedExample() {}
   virtual ~IderivedExample() {}
 private:
   void operator=(const IderivedExample&);
};
class OBV_IderivedExample: public virtual IderivedExample{
 protected:
   OBV_IderivedExample() {}
   virtual ~OBV_IderivedExample() {}
};
```

For reference counting, the C++ mapping provides two standard classes. The first class is `CORBA::DefaultValueRefCountBase`, which serves as a base class for any application provided concrete valuetypes that do not derive from any IDL interfaces. For these kinds of valuetypes, the applications are also free to implement their own reference counting mechanisms. The second class is `PortableServer::ValueRefCountBase`, which must serve as a base class for any application provided a concrete valuetype class which does derive from one or more IDL interfaces.

## Valuebox

A valuebox is a valuetype applied to structures, unions, any, string, basic types, object references, enums, sequence, and array types. These types do not support method, inheritance, or interfaces. A valuebox is ref counted and is derived from `CORBA::DefaultValueRefCountBase`. The mapping is different for different underlying types. All valuebox C++ classes provide `_boxed_in(), boxed_out()`, and `_boxed_inout()` for mapping to the underlying types. The factory for a valuebox id automatically registered by the generated stub.

See the *OMG CORBA 2.3 idl2cpp specification*, Chapter 1.17, for more information. The factory for a valuebox is automatically registered by the generated stub.

## Abstract Interfaces

Abstract interfaces are used to determine at runtime, if an object is passed by reference (IOR) or by value (valuetype.) A prefix “abstract” is used for this purpose before an interface declaration.

This code sample shows the IDL code sample.

```
abstract interface foo {
   Void func():
}
```

A valuetype that supports an abstract interface, can be passes as that abstract interface. The abstract interface is declared as follows:

```
valuetype vt supports foo {
   ...
};
```

Similarly, an interface that needs to be passed as an abstract interface is declared as follows:

```
interface intf : foo {
}
```

The C++ mapping for the previously declared abstract interface foo, results in the following classes:

```
class foo_var : public CORBA::_var{
   ...
}
class foo_out{
   ...
};
class foo : public virtual CORBA:::AbstractBase{
   private:
       ...
      void operator=(const foo&) {}
   protected:
      foo();
      foo(const foo& ref) {}
      virtual ~foo() {}
   public:
      static CORBA::Object* _factory():
      foo_ptr _this();
      static foo_ptr _nil() { ... }
      static foo_ptr _narrow(CORBA::AbstractBase* _obj);
      static foo_ptr _narrow(CORBA::Object_ptr _obj);
      static foo_ptr _narrow(CORBA::ValueBase_ptr _obj);
      virtual void func() = 0;
      ...
};
```

```
class _vis_foo_stub : public virtual foo, public virtual CORBA_Object {
   public :
      _vis_foo_stub() {}
      virtual ~_vis_foo_stub() {}
      ...
      virtual void func():
}
```

There is a `_var` class, an `_out` class, and a class derived from `CORBA::AbstractBase` that implements the methods described in the previous code samples.

5

# VisiBroker properties

This section describes the Borland VisiBroker properties.

## Smart Agent properties

| Property | Default | Old property | Description |
|---|---|---|---|
| `vbroker.agent.addrFile` | `null` | `ORBagentAddrFile` | Specifies a file that stores the IP address or host name of a host running a Smart Agent. |
| `vbroker.agent.localFile` | `null` | `N/A` | Specifies which network interface to use on multi-home machines. This used to be the `OSAGENT_LOCAL_FILE` environment variable. |
| `vbroker.agent.clientHandlerPort` | `null` | `N/A` | Specifies the port that the Smart Agent uses to verify the existence of a client—in this case, a VisiBroker application. When you use the default value, `null`, the Smart Agent connects using a random port number. |
| `vbroker.agent.keepAliveTimer` | `120 seconds` | `N/A` | Smart agent will wake up after this timeout and based on the vbroker.agent.keepAliveThreshold value, will compute whether to do client verification. The logic is if the last received heart beat value is less than current time - (keepAliveTimer + keepAliveThreshold), then do client verification. The value of this property should be greater than 1 second and less than 120 seconds. The number of times the client verification is tried can be controlled by vbroker.agent.maxRetries property. |

| Property | Default | Old property | Description |
|---|---|---|---|
| `vbroker.agent.keepAliveThreshold` | 40 seconds | N/A | Refer to documentation on vbroker.agent.keepAliveTimer. This value should be greater than 0. |
| `vbroker.agent.maxRetries` | NA | N/A | The number of times the agent will do client verification on not receiving a heart beat from the client. Values can be 1 to 10. |
| `vbroker.agent.port` | 14000 | `ORBagentPort` | Specifies the port number that defines a domain within your network. VisiBroker applications and the Smart Agent work together when they have the same port number. This is the same property as the `OSAGENT_PORT` environment variable. |

## Smart Agent communication properties

The properties described in the table below are used by the ORB for Smart Agent communication.

| Property | Default | Old property | Description |
|---|---|---|---|
| `vbroker.agent.keepAliveTimer` | 120 | N/A | The duration in seconds during which the ORB will send keep-alive messages to the Smart Agent (applicable to both clients and servers). Valid values are integers between 1 and 120, inclusive. |
| `vbroker.agent.retryDelay` | 0 (zero) | N/A | The duration in seconds that the process will pause before trying to reconnect to the Smart Agent in the event of disconnection from the Smart Agent. If the value is `-1`, the process will exit upon disconnection from the Smart Agent. The default value of `0` (zero) means that reconnection will be made without any pause. |
| `vbroker.agent.addr` | null | `ORBagentAddr` | Specifies the IP address or host name of a host running a Smart Agent. The default value, `null`, instructs VisiBroker applications to use the value from the `OSAGENT_ADDR` environment variable. If this `OSAGENT_ADDR` variable is not set, then it is assumed that the Smart Agent is running on a local host. |
| `vbroker.agent.addrFile` | null | `ORBagentAddrFile` | Specifies a file that stores the IP address or host name of a host running a Smart Agent. |
| `vbroker.agent.debug` | false | `ORBdebug` | When set to `true`, specifies that the system will display debugging information about communication of VisiBroker applications with the Smart Agent. |

| Property | Default | Old property | Description |
|---|---|---|---|
| vbroker.agent.enableLocator | true | ORBdisableLocator | When set to `false`, does not allow VisiBroker applications to communicate with the Smart Agent. |
| vbroker.agent.port | 14000 | ORBagentPort | Specifies the port number that defines a domain within your network. VisiBroker applications and the Smart Agent work together when they have the same port number. This is the same property as the `OSAGENT_PORT` environment variable. |
| vbroker.agent.failOver | true | ORBagentNoFailOver | When set to `true`, allows a VisiBroker application to fail over to another Smart Agent. |
| vborker.agent.clientPort | 0 (zero) | N/A | Specifies the starting port number to bind to the dsuser socket. |
| vbroker.agent.clientPortRange | 0 (zero) | N/A | Specifies a range of port numbers to bind to the dsuser socket. This property should be used in conjunction with the `vbroker.agent.clientPort` property. |

## VisiBroker ORB properties

The following table describes the VisiBroker ORB properties.

| Property | Default | Description |
|---|---|---|
| vbroker.orb.defaultThreadStackSize | 0 (Zero) | This property indicates the default size to be used by any thread created by the ORB.<br>A default value of zero indicates a system default value. If the value is set and falls below the minimum system requirement, then the minimum system requirement value is used. Dispatchers may have their own threadStackSize property set and as such their values are used instead during thread creation. |
| vbroker.orb.enableFirewall | false | When set to true, application will dynamically load the "fw" library to make the Gatekeeper firewall component available.<br>By default, the property is false. when false, the "fw" library is not loaded, so the gatekeeper firewall component is not available |
| vbroker.orb.cacheDSQuery | true | When set to `true`, allows VisiBroker applications to cache IOR. |

| Property | Default | Description |
|---|---|---|
| vbroker.orb.rebindForward | 0 (zero) | This value determines the number of times a client will try to connect to a forwarded target. You can use this property when the client cannot communicate with the forwarded target (because of network failure, for example). The default value of 0 (zero) means that the client will keep trying to connect. |
| vbroker.orb.activationIOR | null | Allows the launched server to easily establish contact with the OAD that launched it. |
| vbroker.orb.oadUID | 0 (zero) | Used to ensure that the OAD that launched the server still exists. A value of 1 indicates that the OAD is still running. |
| vbroker.orb.propStorage | null | Specifies a property file that contains property values. |
| vbroker.orb.backCompat | FALSE | When set to TRUE, the server is operating in backward compatibility mode. |
| vbroker.orb.nullstring | FALSE | When set to TRUE, enables marshaling of null strings. Note that this property is no longer used, and has been replaced by the vbroker.orb.enableNullString property. |
| vbroker.orb.admDir | null | Specifies the administration directory at which various system files are located. This property can be set using the VBROKER_ADM environment variable. |
| vbroker.orb.isNTService | FALSE | When set to TRUE, this property coupled with the compile flag WIN32, enables any NT service/ COM+ app to stay running when a user logs out. |
| vbroker.orb.enableServerManager | FALSE | When set to TRUE, this property enables Server Manager when the server is started, so that clients can access it. |
| vbroker.orb.input.maxBuffers | 16 | Specifies the maximum number of input buffers retained in a pool. |
| vbroker.orb.input.buffSize | 255 | Specifies the size of the input buffer. |
| vbroker.orb.output.maxBuffers | 16 | Specifies the maximum number of output buffers retained in a pool. |
| vbroker.orb.output.buffSize | 255 | Specifies the size of the output buffer. |
| vbroker.orb.initRef | null | Specifies the initial reference. Object URL formats such as corbaloc can be used in addition to stringified IOR. “file://” URL as described below is also supported if the stringified IOR is in a file. |

| Property | Default | Description |
|---|---|---|
| `vbroker.orb.defaultInitRef` | `null` | Specifies the default initial reference. Object URL formats such as corbaloc can be used in addition to stringified IOR. "file://" URL as described below is also supported if the stringified IOR is in a file. |
| `vbroker.orb.boa_map.TSingle` | `boa_s` | Maps the BOA bid policy of a single thread to `boa_s`. |
| `vbroker.orb.boa_map.TPool` | `boa_tp` | Maps the BOA bid policy of a thread pool to `boa_tp`. |
| `vbroker.orb.boa_map.TSession` | `boa_ts` | Maps the BOA bid policy of a thread session to `boa_ts`. |
| `vbroker.orb.boa_map.TPool_LIOP` | `boa_ltp` | Maps the BOA bid policy of a local thread pool to `boa_ltp`. |
| `vbroker.orb.alwaysProxy` | `false` | When set to `true`, specifies that clients must always connect to the server using the GateKeeper. |
| `vbroker.orb.gatekeeper.ior` | `null` | Forces the client application to connect to the server through the GateKeeper whose IOR is provided. |
| `vbroker.locator.ior` | `null` | Specifies the IOR of the GateKeeper that will be used as proxy to the Smart Agent. If this property is not set, the GateKeeper specified by the `vbroker.orb.gatekeeper.ior` property is used for this purpose. For more information, go to the VisiBroker "Introduction to GateKeeper." |
| `vbroker.orb.exportFirewallPath` | `false` | Forces the server application to include firewall information as part of any servant's IOR which this server exposes (use `Firewall::FirewallPolicy` in your code to force it selectively per POA). |
| `vbroker.orb.proxyPassthru` | `false` | If set to `true`, forces `PASSTHROUGH` firewall mode globally in the application scope (use `QoSExt::ProxyModePolicy` in your code to force it selectively per object or per ORB). |
| `vbroker.orb.bids.critical` | `inprocess` | The critical bid has highest precedence no matter where it is specified in the bid order. If there are multiple values for critical bids, then their relative importance is decided by the `bidOrder` property. |

| Property | Default | Description |
|---|---|---|
| `vbroker.orb.bidOrder` | `inprocess:liop:ssl:iiop:proxy:hiop:locator` | You can specify the relative order of importance for the various transports. Transports are given precedence as follows: **1** `inprocess` **2** `liop` **3** `ssl` **4** `iiop` **5** `proxy` **6** `hiop` **7** `locator` The transports that appear first have higher precedence. For example, if an IOR contains both LIOP and IIOP profiles, the first chance goes to `LIOP`. Only if the `LIOP` fails is `IIOP` used. (The critical bid, specified by the `vbroker.orb.bids.critical` property, has highest precedence no matter where it is specified in the bid order.) |
| `vbroker.orb.dynamicLibs` | `null` | Specifies a list of available services used by the VisiBroker ORB. Each service is separated by a comma. |
| `vbroker.orb.embedCodeset` | `true` | When an IOR is created, the VisiBroker ORB embeds the codeset components into the IOR. This may produce problems with some non-compliant ORBs. By turning off the `embedCodeset` property, you instruct the Visibroker ORB not to embed codesets in IORs. When set to `false`, specifies that character and wide character conversions between the client and the server are not to be negotiated. |
| `vbroker.orb.enableVB4backcompat` | `false` | This property enables work-arounds to deal with behavior that is not GIOP 1.2-compliant in VisiBroker 4.0 and 4.1. Any VisiBroker client running on VisiBroker 4.1.1 or a release previous to 4.1.1 is affected, especially if GateKeeper is involved. To work with a Visibroker 4.0 or 4.1 client, this flag needs to be set to `true`. This is a server-side only flag. There is no corresponding flag on the client-side. |
| `vbroker.orb.enableNullString` | `false` | If set to `TRUE`, enables marshaling of null strings. |
| `vbroker.orb.procId` | `0` | Specifies the process ID of the server. |
| `vbroker.orb.usingPoll` | `true` | On UNIX platforms, the ORB uses the system calls select() or poll() for I/O multiplexing based on the value of this property. If the value is true, poll() is used. Otherwise, select() is used. True is the default value. |

The file URL conforms to the standard format of "file://*domain name*/*path*/*file*". However, there are some constraints in the format supported by VisiBroker for C++.

- The *protocol* part of the URL must be file://
- The *domain name* of the URL must be empty
- All path specifications are absolute (relative paths are not allowed)
- The path may not contain the character ":". The path separator must be "/"
- For Windows, the drive letter colon (":") must be replaced by the "|" symbol.

The following paths show examples of valid paths:

- `file:///home/user/appl.ior`
- `file:///C|/My Documents/User/root.txt`

## ServerManager properties

This table lists the Server Manager properties.

| Property | Default | Description |
|---|---|---|
| `vbroker.serverManager.name` | `null` | Specifies the name of the Server Manager. |
| `vbroker.serverManager.enableOperations` | `true` | When set to `true`, enables operations, exposed by the Server Manager, to be invoked. |
| `vbroker.serverManager.enableSetProperty` | `true` | When set to true, enables properties, exposed by the Server Manager, to be changed. |

### Additional Properties

The following section describes the new properties supported by the Server Manager. These properties can be queried through their containers.

#### Properties related to Server-side resource usage

| Property | Description |
|---|---|
| `vbroker.se.<SE_name>.scm.<SCM_name>.listener.preferIPv4Stack` | This is applicable to the Windows platforms only. It is a boolean value indicating whether the listener has to use IPv4 or IPv6. The default value is false, which will enforce usage of IPv6.<br>If the property "`vbroker.se.<SE_name>.host`" is given an IPv4 or IPv6 address value, then you can ignore the property setting. |
| `vbroker.se.<SE_name>.scm.<SCM_name>.manager.allocatedFileDescriptors` | The current number of file descriptors used by the Server Connection Manager (SCM). This value is typically equal to the current number of incoming connections plus two used by the listener. |
| `vbroker.se.<SE_name>.scm.<SCM_name>.manager.maxFileDescriptor` | The maximum value of the file descriptor with the SCM. |
| `vbroker.se.<SE_name>.scm.<SCM_name>.manager.inUseConnections` | The number of incoming connections for which there are requests executing in the ORB. |
| `vbroker.se.<SE_name>.scm.<SCM_name>.manager.idleConnections` | The number of incoming connections for which there are not any requests currently being executed in the ORB. |

| Property | Description |
|---|---|
| vbroker.se.<SE_name>.scm.<SCM_name>.manager.idledTimeoutConnections | The number of idle connections which have also idled past their idle timeout setting but have yet to be closed (due to garbage collection restrictions, for example). |
| vbroker.se.<SE_name>.scm.<SCM_name>.dispatcher.inUseThreads | The number of threads currently executing requests within the dispatcher. |
| vbroker.se.<SE_name>.scm.<SCM_name>.dispatcher.idleThreads | The number of threads which are currently idle waiting for work to be assigned. |

## Properties related to Client-side resource usage

| Property | Description |
|---|---|
| vbroker.ce.<CE_name>.ccm.maxFileDescriptor | The maximum number of file descriptors within the Client Connection Manager (CCM). |
| vbroker.ce.<CE_name>.ccm.activeConnections | The number of connections in the active pool; that is, object references are using these connections. |
| vbroker.ce.<CE_name>.ccm.cachedConnections | The number of connections in the cache pool; no object references are using these connections. |
| vbroker.ce.<CE_name>.ccm.inUseConnections | The number of outgoing connections with pending requests. |
| vbroker.ce.<CE_name>.ccm.idleConnections | The number of outgoing connections with no pending requests. |
| vbroker.ce.<CE_name>.ccm.idledTimeoutConnections | The number of idle connections which have idled past their timeout setting, but have not been closed. |

## Properties related to the Smart Agent (Smart Agent)

| Property | Description |
|---|---|
| vbroker.agent.currentAgentIP | The IP address of the current ORB's Smart Agent (Smart Agent). |
| vbroker.agent.currentAgentClientPort | The port of the Smart Agent to which the ORB is sending requests. |

## Miscellaneous Properties

| Property | Description |
|---|---|
| vbroker.env.path | The value of the `PATH` environment variable under which the ORB is running. |
| vbroker.env.shlibPath | The value of the shared library path environment variable. In HP-UX, it corresponds to the `SHLIB_PATH` environment variable. |
| vbroker.env.orbVersion | This is the ORB version of the currently loaded ORB. It can also be obtained by running `vbver liborb_r.sl` in HP-UX. |
| vbroker.process.fileDescriptorLimit | The maximum number of file descriptors for the current process. |
| vbroker.orb.uid | The user ID of the user who started the VisiBroker server application. |
| vbroker.orb.commandLine | The command-line argument passed to the `CORBA::ORB_init` method. |

## Location Service properties

The following table lists the Location Service properties.

| Property | Default | Description |
|---|---|---|
| vbroker.locationservice.debug | false | When set to true, allows the Location Service to display debugging information.<br>Note: This property has been deprecated. Refer to the new Debug Logger Properties. |
| vbroker.locationservice.verify | false | When set to true, allows the Location Service to check for the existence of an object referred by an object reference sent from the Smart Agent. Only objects registered BY_INSTANCE are verified for existence. Objects that are either registered with OAD, or those registered BY_POA policy are not verified for existence. |
| vbroker.locationservice.timeout | 1 | Specifies the connect/receive/send timeout, in seconds, when trying to interact with the Location Service. |

## Event Service properties

The following table lists the Event Service properties.

| Property | Default | Description |
|---|---|---|
| vbroker.events.maxQueueLength | 100 | Specifies the number of messages to be queued for slow consumers. |
| vbroker.events.factory | false | When set to true, allows the event channel factory to be instantiated, instead of an event channel. |
| vbroker.events.debug | false | When set to true, allows output of debugging information.<br>Note: This property is deprecated. Refer to the new Debug logger properties. |
| vbroker.events.interactive | false | When set to true, allows the event channel to be executed in a console-driven, interactive mode. |

## Naming Service (VisiNaming) properties

The following tables list the VisiNaming Service properties.

**Table 5.1** Core Visinaming Service properties

| Property | Default | Description |
|---|---|---|
| vbroker.naming.adminPwd | inprise | Password required by administrative VisiBroker naming service operations. |
| vbroker.naming.enableSlave | 0 | If 1, enables master/slave naming services configuration. See "VisiNaming Service Clusters for Failover and Load Balancing" for information about configuring master/slave naming services. |
| vbroker.naming.iorFile | ns.ior | This property specifies the full path name for storing the naming service IOR. If you do not set this property, the naming service will try to output its IOR into a file named ns.ior in the current directory. The naming service silently ignores file access permission exceptions when it tries to output its IOR. |

Table 5.1 Core Visinaming Service properties

| Property | Default | Description |
|---|---|---|
| `vbroker.naming.logLevel` | `emerg` | This property specifies the level of log messages to be output from the naming service. Acceptable values are:<br>emerg (0): indicates some panic condition.<br>alert (1): a condition that requires user attention—for example, if security has been disabled.<br>crit (2): critical conditions, such as a device error.<br>err (3): error conditions.<br>warning (4): warning conditions—these may include some troubleshooting advice.<br>notice (5): conditions that are not errors but may require some attention, such as the opening of a connection.<br>info (6): informational, such as binding in progress.<br>debug (7): debug messages for developers.<br>Note: This property is deprecated. Refer to the new Debug logger properties |
| `vbroker.naming.logUpdate` | `false` | This property allows special logging for all of the update operations on the `CosNaming::NamingContext`, `CosNamingExt::Cluster`, and `CosNamingExt::ClusterManager` interfaces.<br>The `CosNaming::NamingContext` interface operations for which this property is effective are: `bind, bind_context, bind_new_context, destroy, rebind, rebind_context, unbind`.<br>The `CosNamingExt::Cluster` interface operations for which this property is effective are: `bind, rebind, unbind, destroy`.<br>The `CosNamingExt::ClusterManager` interface operation for which this property is effective is: `create_cluster`.<br>When this property value is set to `true` and any of the above methods is invoked, the following log message is printed (the output shows a bind operation being executed):<br>`00000007,5/26/04 10:11 AM,127.0.0.1,00000000,`<br>`VBJ-Application,VBJ ThreadPool Worker,INFO,`<br><br>`OPERATION NAME : bind`<br>`CLIENT END POINT : Connection[socket=Socket`<br>`[addr=/127.0.0.1, port=2026, localport=1993]]`<br>`PARAMETER 0 : [(Tom.LoanAccount)]`<br>`PARAMETER 1 : Stub[repository_id=IDL:Bank/`<br>`LoanAccount:1.0, key=TransientId[poaName=/,`<br>`id={4 bytes: (0)(0)(0)(0)},sec=505,usec=990917734,`<br>`key_string=%00VB%01%00%00%00%02/%00%20%20%00%00%00%`<br>`04%00%00%00%00%00%01%f9;%104f],codebase=null]` |

For more information see the Object Clusters section.

Table 5.2 Object Clustering Related properties

| Property | Default | Description |
|---|---|---|
| `vbroker.naming.enableClusterFailover` | `true` | When set to `true`, it specifies that an interceptor be installed to handle fail-over for objects that were retrieved from the VisiNaming Service. In case of an object failure, an attempt is made to transparently reconnect to another object from the same cluster as the original. |

**Table 5.2** Object Clustering Related properties

| Property | Default | Description |
|---|---|---|
| `vbroker.naming.propBindOn` | 0 | If `1`, the implicit clustering feature is turned on. |
| `vbroker.naming.smrr.pruneStaleRef` | 1 | This property is relevant when the name service cluster uses the Smart Round Robin criterion. When this property is set to `1`, a stale object reference that was previously bound to a cluster with the Smart Round Robin criterion will be removed from the bindings when the name service discovers it. If this property is set to `0`, stale object reference bindings under the cluster are not eliminated. However, a cluster with Smart Round Robin criterion will always return an active object reference upon a `resolve()` or `select()` call if such an object binding exists, regardless of the value of the `vbroker.naming.smrr.pruneStaleRef` property. By default, the implicit clustering in the name service uses the Smart Round Robin criterion with the property value set to `1`. If set to 2, this property disables the clearing of stale references completely, and the responsibility of cleaning up the bindings belongs to the application, rather than to VisiNaming. |

For more information see "VisiNaming Service Clusters for Failover and Load Balancing".

**Table 5.3** Visinaming Service Cluster Related properties

| Property | Default | Description |
|---|---|---|
| `vbroker.naming.enableSlave` | 0 | See "VisiNaming Service properties". |
| `vbroker.naming.slaveMode` | No default. Can be set to `cluster` or `slave`. | This property is used to configure VisiNaming Service instances in the cluster mode or in the master/slave mode. The `vbroker.naming.enableSlave` property must be set to `1` for this property to take effect.<br>Set this property to `cluster` to configure VisiNaming Service instances in the cluster mode. VisiNaming Service clients will then be load balanced among the VisiNaming Service instances that comprise the cluster. Client failover across these instances are enabled.<br>Set this property to `slave` to configure VisiNaming Service instances in the master/slave mode. VisiNaming Service clients will always be bound to the master server if the master is running but failover to the slave server when the master server is down. |
| `vbroker.naming.serverClusterName` | null | This property specifies the name of a VisiNaming Service cluster. Multiple VisiNaming Service instances belong to a particular cluster (for example, `clusterXYZ`) when they are configured with the cluster name using this property. |

**Table 5.3** Visinaming Service Cluster Related properties

| Property | Default | Description |
|---|---|---|
| `vbroker.naming.serverNames` | `null` | This property specifies the factory names of the VisiNaming Service instances that belong to a cluster. Each VisiNaming Service instance within the cluster should be configured using this property to be aware of all the instances that constitute the cluster. Each name in the list must be unique. This property supports the format: `vbroker.naming.serverNames= Server1:Server2:Server3` See the related property, `vbroker.naming.serverAddresses`. |
| `vbroker.naming.serverAddresses` | `null` | This property specifies the host and listening port for the VisiNaming Service instances that comprise a VisiNaming Service cluster. The order of VisiNaming Service instances in this list must be identical to that of the related property `vbroker.naming.serverNames`, which specifies the names of the VisiNaming Service instances that comprise a VisiNaming Service Cluster. This property supports the format: `vbroker.naming. serverAddresses=host1:port1; host2:port2;host3:port3` |
| `vbroker.naming.anyServiceOrder` (To be set on VisiNaming Service clients) | `false` | This property must be set to `true` on the VisiNaming Service client to utilize the load balancing and failover features available when VisiNaming Service instances are configured in the VisiNaming Service cluster mode. The following is an example of how to use this property: `client -DVbroker. naming.anyServiceOrder=true` |

## Pluggable Backing Store Properties

The following tables show property information for the VisiNaming service pluggable backing store types.

### Default properties common to all adapters

| Property | Default | Description |
|---|---|---|
| `vbroker.naming.backingStoreType` | `InMemory` | Specifies the naming service adapter type to use. This property specifies which type of backing store you want the VisiNaming Service to use. The valid options are: `InMemory`, `JDBC`, `Dx`, `JNDI`. The default is `InMemory`. |
| `vbroker.naming.cacheOn` | `0` | Specifies whether to use the Naming Service cache. A value of `1` (one) enables caching. |
| `vbroker.naming.cache.connectString` | `N/A` | This property is required when the Naming Service cache is enabled (`vbroker.naming.cacheOn=1`) and the Naming Service instances are configured in Cluster or Master/Slave mode. It helps locate an Event Service instance in the format `<hostname>:<port>`. For example: `vbroker.naming.cache.connectString=127.0.0.1:14500` See "Caching facility" for details about enabling the caching facility and setting the appropriate properties. |
| `vbroker.naming.cache.size` | `2000` | This property specifies the size of the Naming Service cache. Higher values will mean caching of more data at the cost of increased memory consumption. |
| `vbroker.naming.cache.timeout` | `0 (no limit)` | This property specifies the time, in seconds, since the last time a piece of data was accessed, after which the data in the cache will be purged in order to free memory. The cached entries are deleted in LRU (Least Recently Used) order. |

## JDBC Adapter properties

| Property | Default | Description |
|---|---|---|
| `vbroker.naming.jdbcDriver` | `com.borland.datastore.jdbc.DataStoreDriver` | This property specifies the JDBC driver that is needed to access the database used as your backing store. The VisiNaming Service loads the appropriate JDBC driver specified. Valid values are: <br>■ `com.borland.datastore.jdbc.DataStoreDriver` JDataStore driver <br>■ `com.sybase.jdbc.SybDriver` Sybase driver <br>■ `oracle.jdbc.driver.OracleDriver` Oracle driver <br>■ `interbase.interclient.Driver` Interbase driver <br>■ `weblogic.jdbc.mssqlserver4.Driver` WebLogic MS SQLServer Driver <br>■ `COM.ibm.db2.jdbc.app.DB2Driver` IBM DB2 Driver |
| `vbroker.naming.resolveAutoCommit` | `True` | Sets Auto Commit on the JDBC connection when doing a "resolve" operation. |
| `vbroker.naming.loginName` | `VisiNaming` | The login name associated with the database. |
| `vbroker.naming.loginPwd` | `VisiNaming` | The login password associated with the database. |
| `vbroker.naming.poolSize` | `5` | This property specifies the number of database connections in your connection pool when using the JDBC Adapter as your backing store. |

| Property | Default | Description |
|---|---|---|
| `vbroker.naming.url` | `jdbc:borland:dslocal:rootDB.jds` | This property specifies the location of the database which you want the Naming Service to access. The setting is dependent upon the database in use. Acceptable values are:<br>■ `jdbc:borland:dslocal:<db-name>` JDataStore UTL<br>■ `jdbc:sybase:Tds:<host-name>:<port-number>/<db-name>` Sybase URL<br>■ `jdbc:oracle:thin@<host-name>:<port-number>:<sid>` Oracle URL<br>■ `jdbc:interbase://<server-name>/<full-db-path>` Interbase URL<br>■ `jdbc:weblogic:mssqlserver4:<db-name>@<host-name>:<port-number>` WebLogic MS SQLSever URL<br>■ `jdbc:db2:<db-name>` IBM DB2 URL<br>■ `<full-path-JDataStore-db>` DataExpress URL for the native driver |
| `vbroker.naming.minReconInterval` | `30` | This property sets the Naming Service's database reconnection interval time, in seconds. The default value is `30`. The Naming Service will ignore the reconnection request and throw a `CannotProceed` exception if the time interval between this request and the last reconnection time is less than the vset value. Valid values for this property are non-negative integers. If set to `0`, the Naming Service will try to reconnect to the database for every request. |

## DataExpress Adapter properties

The following table describes the DataExpress Adapter properties:

| Property | Description |
|---|---|
| `vbroker.naming.backingStoreType` | This property should be set to `Dx`. |
| `vbroker.naming.loginName` | This property is the login name associated with the database. The default is `VisiNaming`. |
| `vbroker.naming.loginPwd` | This property is the login password associated with the database. The default value is `VisiNaming`. |
| `vbroker.naming.url` | This property specifies the location of the database. |

## JNDI adapter properties

The following is an example of settings that can appear in the configuration file for a JNDI adapter:

| Setting | Description |
|---|---|
| `vbroker.naming.backingStoreType=JNDI` | This setting specifies the backing store type which is `JNDI` for the JNDI adapter. |
| `vbroker.naming.loginName=<user_name>` | The user login name on the JNDI backing server. |
| `vbroker.naming.loginPwd=<password>` | The password for the JNDI backing server user. |
| `vbroker.naming.jndiInitialFactory=com.sun.jndi.ldap.LdapCtxFactory` | This setting specifies the JNDI initial factory. |
| `vbroker.naming.jndiProviderURL=ldap://<hostname>:389/<initial root context>` | This setting specifies the JNDI provider URL |
| `vbroker.naming.jndiAuthentication=simple` | This setting specifies the JNDI authentication type supported by the JNDI backing server. |

## VisiNaming Service Security-related properties

| Property | Value | Default | Description |
|---|---|---|---|
| `vbroker.naming.security.disable` | `boolean` | `true` | This property indicates whether the security service is disabled. |
| `vbroker.naming.security.authDomain` | `string` | `""` | This property indicates the authorization domain name to be used for the naming service method access authorization. |
| `vbroker.naming.security.transport` | `int` | `3` | This property indicates what transport the Naming Service will use. The available values are: `ServerQoPPolicy.SECURE_ONLY=1` `ServerQoPPolicy.CLEAR_ONLY=0` `ServerQoPPolicy.ALL=3` |
| `vbroker.naming.security.requireAuthentication` | `boolean` | `false` | This property indicates whether naming client authentication is required. However, when the `vbroker.naming.security.disable` property is set to `true`, no client authentication will be performed regardless of the value of this requireAuthentication property. |
| `vbroker.naming.security.enableAuthorization` | `boolean` | `false` | This property indicates whether method access authorization is enabled. |
| `vbroker.naming.security.requiredRolesFile` | `string` | *`null`* | This property points to the file containing the required roles that are necessary for invocation of each method in the protected object types. For more information see "Method Level Authorization". |

## OAD properties

This following table lists the configurable OAD properties.

| Property | Default | Description |
|---|---|---|
| `vbroker.oad.spawnTimeOut` | `20` | After the OAD spawns an executable, specifies how long, in seconds, the system will wait to receive a callback from the desired object before throwing a `NO_RESPONSE` exception. |
| `vbroker.oad.verbose` | `false` | Allows the OAD to print detailed information about its operations. |
| `vbroker.oad.readOnly` | `false` | When set to true, does not allow you to register, unregister, or change the OAD implementation. |
| `vbroker.oad.iorFile` | `Oadj.ior` | Specifies the filename for the OAD's stringified IOR. |
| `vbroker.oad.quoteSpaces` | `false` | Specifies whether to quote a command. |
| `vbroker.oad.killOnUnregister` | `false` | Specifies whether to kill spawned server processes, once they are unregistered. |
| `vbroker.oad.verifyRegistration` | `false` | Specifies whether to verify the object registration. |

This table list the OAD properties that cannot be overridden in a property file. They can however be overridden with environment variables or from the command line.

| Property | Default | Description |
|---|---|---|
| `vbroker.oad.implName` | `impl_rep` | Specifies the filename for the implementation repository. |
| `vbroker.oad.implPath` | `null` | Specifies the directory where the implementation repository is stored. |
| `vbroker.oad.path` | `null` | Specifies the directory for the OAD. |
| `vbroker.oad.systemRoot` | `null` | Specifies the root directory. |
| `vbroker.oad.windir` | `null` | Specifies the Windows directory. |

## Interface Repository properties

The following table lists the Interface Repository (IR) properties.

| Property | Default | Description |
|---|---|---|
| `vbroker.ir.debug` | `false` | When set to true, allows the IR resolver to display debugging information.<br>Note: This property is deprecated. Refer to the new Debug logger properties. |
| `vbroker.ir.ior` | `null` | When the vbroker.ir.name property is set to the default value, `null`, the VisiBroker ORB will try to use this property to locate the IR. |
| `vbroker.ir.name` | `null` | Specifies the name that is used by the VisiBroker ORB to locate the IR. |

## TypeCode properties

The table below lists the VisiBroker for C++ TypeCode properties.

| Property | Default | Description |
|---|---|---|
| `vbroker.typecode.debug` | `FALSE` | When set to `TRUE`, this property allows the typecode code to display debugging.<br>Note: This property is deprecated. Refer to the new Debug logger properties. |
| `vbroker.typecode.noIndirection` | `FALSE` | When set to `TRUE`, this property does not allow the use of indirection when writing a recursive typecode. |
| `vbroker.typecode.marshalName` | `TRUE` | Marshalling of names inside typecode data can now be suppressed by replacing these with empty strings, since the OMG spec allows. This will save network bandwidth by reducing the length of GIOP messages. However, the API functions relying on this data will not function correctly when compression is used. By default, the compression is not done. To enable this, set this property to false. |

## Client-Side LIOP Connection properties

The table below lists the VisiBroker for C++ client-side LIOP connection properties.

**Table 5.4** Client-side LIOP connection properties

| Property | Default | Description |
|---|---|---|
| `vbroker.ce.liop.ccm.connectionCacheMax` | `5` | Specifies the maximum number of cached connections on a client. The connection is cached when a client releases it. Therefore, the next time a client needs a new connection, it can retrieve one from the cache instead of creating a new one. |
| `vbroker.ce.liop.ccm.disableConnectionCache` | `false` | When set to `true`, this property disables connection caching on the client side. |
| `vbroker.ce.liop.ccm.connectionMax` | `0` | Specifies the maximum number of total connections for a client. This includes the active connections, plus the ones that are cached. The default value of `0` (zero) specifies that the client will not try to close any of the old active or cached connections. |
| `vbroker.ce.liop.ccm.connectionMaxIdle` | `360` | Specifies the time, in seconds, that the client uses to determine if a cached connection should be closed. If a cached connection has been idle longer than this time, then the client will close the connection. |
| `vbroker.ce.liop.ccm.type` | `Pool` | Specifies the type of client connection management used by a client. The default value `Pool` means connection pool. This is currently the only valid value for this property. |

**Table 5.4** Client-side LIOP connection properties

| Property | Default | Description |
|---|---|---|
| `vbroker.ce.liop.connection.rcvBufSize` | `0` | Specifies the size of the receive socket buffer. The default value `0` (zero) implies a system dependent value. |
| `vbroker.ce.liop.connection.sendBufSize` | `0` | Specifies the size of the send socket buffer. The default value `0` (zero) implies a system dependent value. |
| `vbroker.ce.liop.connection.shmSize` | `4096` | Specifies the size, in bytes, of shared memory. If your client program and object implementation communicate via shared memory, you may use this option to enhance performance. |
| `vbroker.se.default.local.listener.doorMaxMsgSize` | `1,000,000` | Specifies the maximum message size which will be sent through the fast IPC (door) mechanism in Solaris (when the client and server are running on the same machine). If the message size is greater than the default value (1,000,000), it will not be sent using the IPC, and will default to the next available mechanism (UNIX domain socket or TCP/IP socket). |

## Client-side IIOP connection properties

The table below lists the VisiBroker for C++ Client-side IIOP Connection properties.

**Table 5.5** Client-side IIOP Connection properties

| Property | Default | Description |
|---|---|---|
| `vbroker.ce.iiop.ccm.connectionCacheMax` | `5` | Specifies the maximum number of cached connections for a client. The connection is cached when a client releases it. Therefore, the next time a client needs a new connection, it first tries to retrieve one from the cache, instead of just creating a new one. |
| `vbroker.ce.iiop.ccm.disableConnectionCache` | `false` | If you set this property to `true`, it disables connection caching on the client side. |
| `vbroker.ce.iiop.ccm.connectionMax` | `0` | Specifies the maximum number of total connections for a client. This is equal to the number of active connections plus cached connections. The default value of zero specifies that the client will not try to close any of the old active or cached connections. |
| `vbroker.ce.iiop.ccm.connectionMaxIdle` | `0` | Specifies the time, in seconds, that the client uses to determine if a cached connection should be closed. If a cached connection has been idle longer than this time, then the client closes the connection. |

**Table 5.5** Client-side IIOP Connection properties

| Property | Default | Description |
|---|---|---|
| `vbroker.ce.iiop.ccm.type` | `Pool` | Specifies the type of client connection management used by a client. The value `Pool` means connection pool. This is currently the only valid value for this property. |
| `vbroker.ce.iiop.connection.rcvBufSize` | `0` | Specifies the size of the receive socket buffer. The default value `0` (zero) implies a system dependent value. |
| `vbroker.ce.iiop.connection.sendBufSize` | `0` | Specifies the size of the send socket buffer. The default value `0` (zero) implies a system dependent value. |
| `vbroker.ce.iiop.connection.tcpNoDelay` | `FALSE` | When set to `TRUE`, the server's sockets are configured to send any data written to them immediately instead of batching the data as the buffer fills. |
| `vbroker.ce.iiop.host` | `none` | Binds the client side sockets to the desired interface. If the value is null, the wild-card interface is used. |
| `vbroker.ce.iiop.connection.noCallback` | `FALSE` | When set to `TRUE`, this property allows the server to call back to the client. |
| `vbroker.ce.iiop.connection.socketLinger` | `0` | A TCP/IP setting. |
| `vbroker.ce.iiop.connection.keepAlive` | `TRUE` | A TCP/IP setting. |

## QoS-related Properties

| Property | Default | Description |
|---|---|---|
| `vbroker.qos.cache` | `True` | Specifies if QoS policies should be cached per delegate, instead of being checked prior to every request made by the client. |
| `vbroker.qos.defaultRRTTimeout` | `0 milli-secs` | Sets the default value of relative round trip request timeout. Default 0 means no timeout. |
| `vbroker.qos.defaultRRQTimeout` | `0 milli-secs` | Sets the default value of relative request timeout. Default 0 means no timeout |
| `vbroker.qos.defaultConnectTimeout` | `0 milli-secs` | Sets the default value of connection timeout. Default 0 means no timeout. |

## Server-side server engine properties

This table lists the server-side server engine properties.

| Property | Default | Description |
|---|---|---|
| `vbroker.se.default` | `iiop_tp` | Specifies the default server engine. |

## Server-side thread session IIOP_TS/IIOP_TS connection properties

The following table lists the server-side thread session IIOP_TS/IIOP_TS connection properties.

| Property | Default | Description |
|---|---|---|
| vbroker.se.iiop_ts.scm.iiop_ts.listener.preferIPv4Stack | false | This is applicable to the Windows platforms only. It is a boolean value indicating whether the listener has to use IPv4 or IPv6. The default value is false, which will enforce usage of IPv6. If the property "vbroker.se.<SE_name>.host" is given an IPv4 or IPv6 address value, then you can ignore the property setting. |
| vbroker.se.iiop_ts.host | null | Specifies the host name used by this server engine. The default value, null, means use the host name from the system. |
| vbroker.se.iiop_ts.proxyHost | null | Specifies the proxy host name used by this server engine. The default value, null, means use the host name from the system. |
| vbroker.se.iiop_ts.scms | iiop_ts | Specifies the list of Server Connection Manager name(s). |
| vbroker.se.iiop_ts.scm.iiop_ts.manager.type | Socket | Specifies the type of Server Connection Manager. |
| vbroker.se.iiop_ts.scm.iiop_ts.manager.connectionMax | 0 | Specifies the maximum number of connections the server will accept. The default value, 0 (zero), implies no restriction. |
| vbroker.se.iiop_ts.scm.iiop_ts.manager.connectionMaxIdle | 0 | Specifies the time in seconds the server uses to determine if an inactive connection should be closed. |
| vbroker.se.iiop_ts.scm.iiop_ts.manager.garbageCollectTimer | 30 | The number of seconds between garbage-collection for connection objects. |
| vbroker.se.iiop_ts.scm.iiop_ts.listener.type | IIOP | Specifies the type of protocol the listener is using. |
| vbroker.se.iiop_ts.scm.iiop_ts.listener.port | 0 | Specifies the port number that is used with the host name property. The default value, 0 (zero), specifies that the system will pick a random port number. |
| vbroker.se.iiop_ts.scm.iiop_ts.listener.proxyPort | 0 | Specifies the proxy port number used with the proxy host name property. The default value, 0 (zero), specifies that the system will pick a random port number. |
| vbroker.se.iiop_ts.scm.iiop_ts.listener.rcvBufSize | 0 | Specifies the size of the receive socket buffer. The default value 0 implies system dependent value. |
| vbroker.se.iiop_ts.scm.iiop_ts.listener.sendBufSize | 0 | Specifies the size of the send buffer. The default value 0 implies a system dependent value. |
| vbroker.se.iiop_ts.scm.iiop_ts.listener.socketLinger | 0 | A TCP/IP setting |
| vbroker.se.iiop_ts.scm.iiop_ts.listener.keepAlive | true | A TCP/IP setting |

| Property | Default | Description |
|---|---|---|
| vbroker.se.iiop_ts.scm.iiop_ts.listener.giopVersion | 1.2 | This property can be used to resolve interoperability problems with older VisiBroker ORBs that cannot handle unknown minor GIOP versions correctly. Legal values for this property are `1.0`, `1.1` and `1.2`. For example, to make the nameservice produce a GIOP 1.1 ior, start it like this: `nameserv -VBJprop vbroker.se.iiop_tp.scm.iiop_tp.listener.giopVersion=1.1` |
| vbroker.se.iiop_ts.scm.iiop_ts.dispatcher.type | "ThreadSession" | Specifies the type of thread dispatcher used in the Server Connection Manager. |
| vbroker.se.iiop_ts.scm.iiop_ts.dispatcher.threadStackSize | 0 | The size of the thread stack. The default value, 0, indicates system default. However, on the HP-UX platform, the default value is 128 KB. |
| vbroker.se.iiop_ts.scm.iiop_ts.dispatcher.coolingTime | 3 | Time duration in seconds when a connection is considered hot (expecting more requests). After the time is elapsed, the connection is returned back from the dispatcher. |
| vbroker.se.iiop_ts.scm.iiop_ts.connection.rcvBufSize | 0 | Specifies the size of the receive socket buffer. The default value 0 implies system dependent value. |
| vbroker.se.iiop_ts.scm.iiop_ts.connection.sendBufSize | 0 | Specifies the size of the send buffer. The default value 0 implies a system dependent value. |
| vbroker.se.iiop_ts.scm.iiop_ts.connection.socketLinger | 0 | A TCP/IP setting |
| vbroker.se.iiop_ts.scm.iiop_ts.connection.keepAlive | true | A TCP/IP setting |
| vbroker.se.iiop_ts.scm.iiop_ts.connection.tcpNoDelay | true | When this property is set to false, this turns on buffering for the socket. The default value, true, turns off buffering, so that all packets are sent as soon as they are ready. |

## Server-side thread session BOA_TS/BOA_TS connection properties

This protocol has the same set of properties as the Server-side thread session IIOP_TS/IIOP_TS connection properties, by replacing all`iiop_ts` with `boa_ts` in all the properties. For example, the `vbroker.se.iiop_ts.scm.iiop_ts.manager.connectionMax` will become `vbroker.se.boa_ts.scm.boa_ts.manager.connectionMax`. Also, the default value for `vbroker.se.boa_ts.scms` is `boa_ts`.

## Server-side thread pool IIOP_TP/IIOP_TP connection properties

The following table lists the server-side thread pool IIOP_TP/IIOP_TP connection properties.

| Property | Default | Description |
|---|---|---|
| vbroker.se.iiop_tp.scm.iiop_tp.listener.preferIPv4Stack | false | This is applicable to the Windows platforms only. It is a boolean value indicating whether the listener has to use IPv4 or IPv6. The default value is false, which will enforce usage of IPv6.<br>If the property "vbroker.se.<SE_name>.host" is given an IPv4 or IPv6 address value, then you can ignore the property setting. |
| vbroker.se.iiop_tp.host | null | Specifies the host name that can be used by this server engine. The default value, null, means use the host name from the system. Host names or IP addresses are acceptable values. |
| vbroker.se.iiop_tp.proxyHost | null | Specifies the proxy host name that can be used by this server engine. The default value, null, means use the host name from the system. Host names or IP addresses are acceptable values. |
| vbroker.se.iiop_tp.scms | iiop_tp | Specifies the list of Server Connection Manager name(s). |
| vbroker.se.iiop_tp.scm.iiop_tp.manager.type | Socket | Specifies the type of Server Connection Manager. |
| vbroker.se.iiop_tp.scm.iiop_tp.manager.connectionMax | 0 | Specifies the maximum number of cache connections on the server. The default value, 0 (zero), implies no restriction. |
| vbroker.se.iiop_tp.scm.iiop_tp.manager.connectionMaxIdle | 0 | Specifies the time, in seconds, that the server uses to determine if an inactive connection should be closed. |
| vbroker.se.iiop_tp.scm.iiop_tp.manager.garbageCollectTimer | 30 | The garbage-collection timer (in seconds) for connections. |
| vbroker.se.iiop_tp.scm.iiop_tp.listener.type | IIOP | Specifies the type of protocol the listener is using. |
| vbroker.se.iiop_tp.scm.iiop_tp.listener.port | 0 | Specifies the port number used with the host name property. The default value, 0 (zero), means that the system will pick a random port number. |
| vbroker.se.iiop_tp.scm.iiop_tp.listener.proxyPort | 0 | Specifies the proxy port number used with the proxy host name property. The default value, 0 (zero), means that the system will pick a random port number. |
| vbroker.se.iiop_tp.scm.iiop_tp.listener.rcvBufSize | 0 | Specifies the size of the receive socket buffer. The default value 0 implies a system dependent value. |
| vbroker.se.iiop_tp.scm.iiop_tp.listener.sendBufSize | 0 | Specifies the size of the send buffer. The default value 0 implies a system dependent value. |
| vbroker.se.iiop_tp.scm.iiop_tp.listener.socketLinger | 0 | A TCP/IP setting |
| vbroker.se.iiop_tp.scm.iiop_tp.listener.keepAlive | true | A TCP/IP setting |
| vbroker.se.iiop_tp.scm.iiop_tp.listener.giopVersion | 1.2 | This property can be used to resolve interoperability problems with older VisiBroker ORBs, that cannot handle unknown minor GIOP versions correctly. Acceptable values for this property are 1.0, 1.1 and 1.2. |

| Property | Default | Description |
|---|---|---|
| vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.type | ThreadPool | Specifies the type of thread dispatcher used in the Server Connection Manager. |
| vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMin | 0 | Specifies the minimum number of threads that the Server Connection Manager can create. |
| vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMax | 0 | Specifies the maximum number of threads that the Server Connection Manager can create. The default value, `0` (zero) implies the ORB will control the thread generation using an internal algorithm based on heuristics.<br>Setting the property `vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.unlimitedConcurrency=true` will imply that setting this property to 0 will enable unlimited number of threads in the thread pool to be created. |
| vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMaxIdle | 300 | Specifies the time in seconds before an idle thread will be destroyed. |
| vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadStackSize | 0 | The size of the thread stack. The default value 0 indicates the system default. However, on the HP-UX platform, the default value is 128 KB. |
| vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.coolingTime | 3 | Time duration, in seconds, when a connection is considered hot (expecting more requests). After the time is elapsed, the connection is returned back from the dispatcher. |
| vbroker.se.iiop_tp.scm.iiop_tp.connection.rcvBufSize | 0 | Specifies the size of the receive socket buffer. The default value 0 implies a system dependent value. |
| vbroker.se.iiop_tp.scm.iiop_tp.connection.sendBufSize | 0 | Specifies the size of the send buffer. The default value 0 implies a system dependent value. |
| vbroker.se.iiop_tp.scm.iiop_tp.connection.socketLinger | 0 | A TCP/IP setting |
| vbroker.se.iiop_tp.scm.iiop_tp.connection.keepAlive | true | A TCP/IP setting |
| vbroker.se.iiop_tp.scm.iiop_tp.connection.tcpNoDelay | true | When this property is set to `false`, this turns on buffering for the socket. The default value, `true`, turns off buffering, so that all packets are sent as soon as they are ready. |

## Server-side thread pool BOA_TP/BOA_TP connection properties

This protocol has the same set of properties as the Server-side thread pool IIOP_TP/IIOP_TP connection properties, by replacing all `iiop_tp` with `boa_tp` in all the properties. For example, the `vbroker.se.iiop_tp.scm.iiop_tp.manager.connectionMax` will become `vbroker.se.boa_tp.scm.boa_tp.manager.connectionMax`. Also, the default value for `vbroker.se.boa_tp.scms` is `boa_tp`.

## Server-side thread pool LIOP_TP/LIOP_TP connection properties

The following table lists the server-side thread pool LIOP_TP/LIOP_TP connection properties.

| Property | Default | Description |
|---|---|---|
| `vbroker.se.liop_tp.host` | `null` | Specifies the host name that can be used by this server engine. The default value, null, means use the host name from the system. Host names or IP addresses are acceptable values. |
| `vbroker.se.liop_tp.proxyHost` | `null` | Specifies the proxy host name that can be used by this server engine. The default value, null, means use the host name from the system. Host names or IP addresses are acceptable values. |
| `vbroker.se.liop_tp.scms` | `liop_tp` | Specifies the list of Server Connection Manager name(s). |
| `vbroker.se.liop_tp.scm.liop_tp.manager.type` | `Local` | Specifies the type of Server Connection Manager. |
| `vbroker.se.liop_tp.scm.liop_tp.manager.connectionMax` | `0` | Specifies the maximum number of cache connections on the server. The default value, 0 (zero), implies no restriction. |
| `vbroker.se.liop_tp.scm.liop_tp.manager.connectionMaxIdle` | `0` | Specifies the time, in seconds, that the server uses to determine if an inactive connection should be closed. |
| `vbroker.se.liop_tp.scm.liop_tp.manager.garbageCollectTimer` | `30` | The garbage-collection timer (in seconds) for connections. |
| `vbroker.se.liop_tp.scm.liop_tp.listener.type` | `LIOP` | Specifies the type of protocol the listener is using. |
| `vbroker.se.liop_tp.scm.liop_tp.listener.port` | `0` | Specifies the port number used with the host name property. The default value, 0 (zero), means that the system will pick a random port number. |
| `vbroker.se.liop_tp.scm.liop_tp.listener.proxyPort` | `0` | Specifies the proxy port number used with the proxy host name property. The default value, 0 (zero), specifies that the system will pick a random port number. |
| `vbroker.se.default.local.listener.door` | `true` | Specifies whether the Door API has to be used for the Client and Server to communicate when running on the same machine. When set to true, the Door API is used for the LIOP. When set to false, the LIOP uses a UNIX Domain Socket for IPC. This property is only for Solaris operating systems. |
| `vbroker.se.xxx.scm.yyy.listener.shmSize` | `4096` | The size, in bytes, of the shared memory allocation. If your client program and object implementation communicate via shared memory, you may use this option to enhance performance. |
| `vbroker.se.xxx.scm.yyy.listener.userConstrained` | `0` | When set to true, the file is hidden in a directory accessible only by the owner. |
| `vbroker.se.liop_tp.scm.liop_tp.listener.giopVersion` | `1.2` | This property can be used to resolve interoperability problems with older VisiBroker ORBs, that cannot handle unknown minor GIOP versions correctly. Acceptable values for this property are 1.0, 1.1 and 1.2. |

| Property | Default | Description |
|---|---|---|
| `vbroker.se.liop_tp.scm.liop_tp.listener.allowedGroups` | `null` | Allows server applications to control the trustees to the securable synchronization objects, used for Local IPC communication on Windows. Allows semicolon-separated Windows User groups to access servers using LIOP. Users not belonging to the groups specified will not be allowed to connect and will failover to IIOP. |
| `vbroker.se.liop_tp.scm.liop_tp.dispatcher.type` | `ThreadPool` | Specifies the type of thread dispatcher used in the Server Connection Manager. |
| `vbroker.se.liop_tp.scm.liop_tp.dispatcher.threadMin` | `0` | Specifies the minimum number of threads that the Server Connection Manager can create. |
| `vbroker.se.liop_tp.scm.liop_tp.dispatcher.threadMax` | `0` | Specifies the maximum number of threads that the Server Connection Manager can create. The default value, 0 (zero), implies the ORB will control the thread generation using an internal algorithm based on heuristics. |
| `vbroker.se.liop_tp.scm.liop_tp.dispatcher.threadMaxIdle` | `300` | Specifies the time, in seconds, before an idle thread will be destroyed. |
| `vbroker.se.liop_ts.scm.liop_ts.dispatcher.threadStackSize` | `0` | The size of the thread stack. The default value, 0, indicates a system default. However, on the HP-UX platform, the default value is 128 KB. |
| `vbroker.se.liop_tp.scm.liop_tp.dispatcher.coolingTime` | `3` | Time duration, in seconds, when a connection is considered hot (expecting more requests). After the time is elapsed, the connection is returned back from the dispatcher. |

## Server-side thread pool BOA_LTP/BOA_LTP connection properties

This protocol has the same set of properties as the thread pool `liop_tp/liop_tp` connection properties, by replacing all `liop_tp` with `boa_ltp` in all the properties. For example, the `vbroker.se.liop_tp.scm.liop_tp.manager.connectionMax` will become `vbroker.se.boa_ltp.scm.boa_ltp.manager.connectionMax`. Also, the default value for `vbroker.se.boa_ltp.scms` is `boa_ltp`.

## Properties that support bi-directional communication

The following table lists the properties that support bi-directional communication. These properties are evaluated only once—when the SCMs are created. In all cases, the `exportBiDir` and `importBiDir` properties on the SCMs are given priority over 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 bi-directionality in individual SCMs.

| Property | Default | Description |
|---|---|---|
| `vbroker.orb.enableBiDir` | none | You can selectively make bi-directional connections. If the client defines `vbroker.orb.enableBiDir=client` and the server defines `vbroker.orb.enableBiDir=server` the value of `vbroker.orb.enableBiDir` at the GateKeeper determines the state of the connection. Values of this property are: `server`, `client`, `both` or `none`. |
| `vbroker.se.<se>.scm.<scm>.manager.exportBiDir` | By default, this property is not set by the ORB. | This is a client-side property. Setting it to `true` enables creation of a bi-directional callback POA on the specified server engine. Setting it to `false` disables creation of a bidirectional POA on the specified server engine. |
| `vbroker.se.<se>.scm.<scm>.manager.importBiDir` | By default, not set by the ORB. | This is a server-side property. 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. |

## Debug Logging properties

This section details the properties that can be used to control and configure the output of debug log statements.

The debug log statements are categorized according to the areas of the ORB from where they are logged. These categories are called *source names*. Currently the following source names are logged:

- `connection` – logs from the connection-related source areas such as client side connection, server side connection, connection pool etc.
- `client` – logs from the client side invocation path
- `agent` – logs for Osagent communication
- `cdr` – logs for GIOP areas
- `se` – logs from the server engine, such as dispatcher, listener etc.
- `server` – logs from the server side invocation path.
- `orb` – logs from the ORB.

**For VisiNotify, the following source names are logged:**

- v_vntfy - logs from the process.
- v_vnchnl - logs from the channel object.
- v_vn_pxsup - logs from proxy supplier objects.
- v_vnper - logs from persistency module.

- v_vndb - logs from low level circular file based db layer
- v_vnlogdiscard - Details of any events that are discarded

**For VisiTelcoLog, the following source names are logged:**

- v_vtlog - logs from the process.
- v_vtlper - logs from the log persistence layer.
- v_vndb - logs from low level circular file based db layer.

**For VisiTransact, the following source names are logged:**

- v_ots_txncontext - logs from transaction factory, control and coordinator.
- v_ots_interceptor - logs from client and server interceptors and transaction current related operations.
- v_ots_completion - logs related to transaction completion.
- v_ots_pc - logs related to resource and synchronization objects registered.

**For VisiSecure C++, the following source names are logged:**

- v_secauthn - logs from authentication related code (i.e. login module, callback handler, identity services and alike).
- v_secauthz - logs from authorization related code (i.e. Authorization provider, Authorization domain, role map and alike).
- v_secssl - logs from SSL transport related code (i.e. SecureSocketProvider, CertificateFactory and alike).
- v_seccsiv2 - logs from CSIV2 service context protocol related code (i.e. Security context management code and alike).
- v_secmisc - logs from the rest of code.

## Enabling and Filtering

The following table describes the properties used to enable logging and filtering.

| Property | Default | Description |
|---|---|---|
| `vbroker.log.enable` | `false` | When set to `true`, all logging statements will be produced unless the log is being filtered.<br>Values are `true` or `false`. |
| `vbroker.log.logLevel` | `debug` | Specifies the logging level of the log message. When set at a level, the logs with log levels equal to the specified level or above are forwarded. This property is applied at the global level.<br>Values are `emerg`, `alert`, `crit`, `err`, `warning`, `notice`, `info` and `debug` ranking from the highest to the lowest.<br>The meaning of the log levels are:<br>■ `emerg`—indicates a panic condition.<br>■ `alert`—a condition that requires user attention—for example, if security has been disabled.<br>■ `crit`—critical conditions, such as a device error.<br>■ `err`—error conditions.<br>■ `warning`—warning conditions—these may accompany some troubleshooting advice, such as on the opening of a connection.<br>■ `info`—informational, such as binding in progress.<br>■ `debug`—debug conditions used by developers. |
| `vbroker.log.default.filter.register` | `null` | Register source name for controlling (filtering) the logs from that source.<br>Values are `client`, `server`, `connection`, `cdr`, `se`, `agent` and `orb`. Multiple values can be provided as a comma-separated string.<br>Note: The source names must be registered using this property before they can be explicitly controlled using<br>vbroker.log.default.filter.<source-name>.enableand<br>vbroker.log.default.filter.<source-name>.logLevel properties.. |
| `vbroker.log.default.filter.<source-name>.enable` | `true` | Once a source name is registered, log output from the source can be explicitly controlled using this property.<br>Values are `true` or `false`. |

| Property | Default | Description |
|---|---|---|
| `vbroker.log.default.filter.<source-name>.logLevel` | `debug` | This property provides finer-grained control over the global log level property. The log level specified using this property explicitly applies to the given source name.<br>The possible values are similar to the global logLevel values. |
| `vbroker.log.default.filter.all.enable` | `true` | This is a special case of the previous property where an inbuilt source name “all” is being used. “all” here denotes all the source names that have not been registered. |
| `vbroker.log.enableSigHandler` | `false` | When set to `true`, installs a signal handler based on SIGUSR2 to allow toggling of logging at runtime.<br>Values are `true` or `false`.<br>**Note:** This applies only to UNIX platforms. |

## Appending and Logging

The output of the logs can be appended (forwarded) to either the Console or a rolling local file system file (or both), either in a simple layout or in a more complicated Log4J XML event layout (format). By default, the logs are appended to the Console in a simple layout. The names of the various appenders and layouts supported are:

- `stdout` – Name of the Console appender.
- `rolling` – Name of the rolling file appender.
- `simple` – Name of a simple predefined output layout.
- `xml` – Name of Log4J XML event layout.
- `full` – Name of a full record fields printout layout.

The following table describes the properties used to configure the destination of the log output and its format.

| Property | Default | Description |
|---|---|---|
| `vbroker.log.default.appenders` | | List of comma-separated appenders instance names for specifying log output destination. |
| `vbroker.log.default.appender.<appender-inst-name>.appenderType` | `stdout` | Type of the appender instance that needs to be configured with the logger. Values could be stdout or rolling or a custom appender type. |
| `vbroker.log.default.appender.<appender-inst-name>.layoutType` | `simple` | Type of layout (format) to be associated with the registered appender destination.<br>Values are `simple` or `xml` or a custom layout type. |

## Examples

For the built-in rolling appender type, you can create the following configurations. The properties are described below, assuming that for each appender instance, the appender type is specified as “rolling”.

| Property | Default | Description |
|---|---|---|
| `vbroker.log.default.appender.<appender-inst-name>.logDir` | `<current_directory>` | Directory for the rolling log file to reside in. |
| `vbroker.log.default.appender.<appender-inst-name>.fileNam` | `vbrolling.log` | Name of rolling log file. |
| `vbroker.log.default.appender.<appender-inst-name>.maxFileSize` | `10` | Size in MB for each backup before rolling over.<br>Values >= `1`. |
| `vbroker.log.default.appender.<appender-inst-name>.maxBackupIndex` | `1` | Number of backups needed. When set to `0` (zero), no backup is created and logging will keep on appending to the file.<br>Values >= `0`. |

The following properties can be used to define custom appender and layout types.

| Property | Default | Description |
|---|---|---|
| `vbroker.log.appender.register` | | Comma-separated new appender type names being introduced to the logger framework |
| `vbroker.log.appender.<appender-type-name>.sharedLib` | | Complete path including file name of the shared library or the DLL containing the custom appender |
| `vbroker.log.layout.register` | | Comma-separated new layout type names being introduced to the logger framework |
| `vbroker.log.appender.<layout-type-name>.sharedLib` | | Complete path including file name of the shared library or the DLL containing the custom layout |

## Examples

The following examples explain some of the debug logging properties' usage scenarios. In the example commands, *vbapp* is a VisiBroker for C++ application.

**1** To turn on logging with default log level.

```
prompt> vbapp -Dvbroker.log.enable=true
```

**2** To trace only info level and above.

```
prompt> vbapp -Dvbroker.log.enable=true -Dvbroker.log.logLevel=info
```

**3** To turn off agent-related component statements.

```
prompt> vbapp -Dvbroker.log.enable=true \
-Dvbroker.log.default.filter.register=agent \
-Dvbroker.log.default.filter.agent.enable=false
```

**4** To trace the client and connection-related area only.

```
prompt> vbapp -Dvbroker.log.enable=true \
-Dvbroker.log.default.filter.all.enable=false \
-Dvbroker.log.default.register=client,connection
```

5 To trace emerg on se and err on the cdr areas and the rest on info level.

```
prompt> vbapp -Dvbroker.log.enable=true \
-Dvbroker.log.logLevel=info \
-Dvbroker.log.default.filter.register=se,cdr \
-Dvbroker.log.default.filter.se.logLevel=emerg \
-Dvbroker.log.default.filter.cdr.logLevel=err
```

6 To set up output to local file systems with three backups.

```
prompt> vbapp -Dvbroker.log.enable=true -
Dvbroker.log.default.appenders=myappinst1 \

-Dvbroker.log.default.appender.myappinst1.appenderType=rolling \
-Dvbroker.log.default.appender.myappinst1.logDir=/opt/vbc \
-Dvbroker.log.default.appender.myappinst1.fileName=vbc.log \
-Dvbroker.log.default.appender.myappinst1.maxBackupIndex=3
```

7 To set up output to both console and local filesystems in xml format.

```
prompt> vbapp -Dvbroker.log.enable=true -
Dvbroker.log.default.appenders=myappinst1,myappinst2\

-Dvbroker.log.default.appender.myappinst1.appenderType=rolling \
-Dvbroker.log.default.appender.myappinst2.appenderType=stdout \
-Dvbroker.log.default.appender.myappinst1.logDir=/opt/vbc \
-Dvbroker.log.default.appender.myappinst1.fileName=vbc.log \
-Dvbroker.log.default.appender.myappinst1.layoutType=xml \
-Dvbroker.log.default.appender.myappinst2.layoutType=xml

To set the output to two appender instances, one of type stdout and the other
a custom appender, using a simple layout and a custom layout.

prompt> vbapp -Dvbroker.log.enable=true \
-Dvbroker.log.appender.register=mycustomapp \
-Dvbroker.log.appender.mycustomapp.sharedLib=libCustomApp.so \
-Dvbroker.log.layout.register=mycustomlyt \
-Dvbroker.log.layout.mycustomlyt.sharedLib=libCustomLyt.so \
-Dvbroker.log.default.appenders=myappinst1,myappinst2 \
-Dvbroker.log.default.appender.myappinst1.appenderType=mycustomapp \
-Dvbroker.log.default.appender.myappinst1.layoutType=simple \
-Dvbroker.log.default.appender.myappinst2.appenderType=stdout \
-Dvbroker.log.default.appender.myappinst2.layoutType=mycustomlyt
```

## Web Services Runtime Properties

### Enabling the runtime

| Property | Default | Description |
|---|---|---|
| `vbroker.ws.enable` | `false` | Takes in a Boolean true or false parameter. Setting this value to true will enable the VisiBroker Web Services Runtime. |

## Web Services HTTP Listener properties

| Property | Default | Description |
|---|---|---|
| vbroker.ws.listener.host | Null | Specify the host name to be used by the listener. Default null means the host name from the system |
| vbroker.ws.listener.port | <TO DO> | Specify the port number to be used by the listener socket. |

## Web Services Connection Manager properties

| Property | Default | Description |
|---|---|---|
| vbroke.ws.keepAliveConnection | False | HTTP server closes a connection after use. If set to true, it tries to maintain the connection. |
| vbroker.ws.connectionMax | 0 | If keepAliveConnection is true, this property specifies the maximum number of connections the server will accept. Default 0 indicates no restriction. |
| vbroker.ws.connectionMaxIdle | 0 | If keepAliveConnection is true, this property determines the maximum time an unused connection will remain alive. |
| vbroker.ws.garbageCollectTimer | 30 | If keepAliveConnection is true, this property determines the garbage collection cycle time for reaping unused connections. Default is 30 seconds. |
| vbroker.ws.connection.rcvBufSize | 0 | Receive Buffer Socket option for the client connection sockets. Default 0 implies system dependent value. |
| vbroker.ws.connection.sendBufSize | 0 | Send Buffer Socket option for the client connection sockets. Default 0 implies system dependent value. |
| vbroker.ws.connection.socketLinger | 0 | TCP Socket option for the client connection sockets. |
| vbroker.ws.connection.keepAlive | true | TCP Socket option for the client connection sockets. |

## SOAP Request Dispatcher properties

**Soap Request Dispatcher properties**

| Property | Default | Description |
|---|---|---|
| vbroker.ws.dispatcher.threadMax | 0 | Maximum number of threads to be present in the thread pool dispatcher. Default value 0 indicates unlimited number of threads. |
| vbroker.ws.dispatcher.threadMin | 0 | Minimum number of threads to be present in the thread pool dispatcher. |
| vbroker.ws.dispatcher.threadMaxIdle | 300 | Time in seconds before an idled thread in the thread pool is destroyed. |
| vbroker.ws.dispatcher.threadStackSize | 0 | Stack size of the thread pool dispatcher thread. Default value 0 indicates system dependent. |

## Real-time Extensions related properties

The properties in the following table can be used to configure individual internal ORB thread priorities.

| Property | Description |
|---|---|
| `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 Manager. 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.agent.threadPriority` | 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 a VisiBroker Smart Agent (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. |

6

# 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 VisiBroker ORB is responsible for reflecting this information back to the client.

## System exceptions

System exceptions are usually raised by the VisiBroker ORB, though it is possible for object implementations to raise them through interceptors discussed in "Using VisiBroker Interceptors." When the VisiBroker ORB raises a `SystemException`, one of the CORBA-defined error conditions is displayed as shown below.

For a listing of explanations and possible causes of these exceptions, see "CORBA 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_QOS` | Quality of service cannot be supported. |
| `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` | VisiBroker ORB initialization failure. |

| Exception name | Description |
|---|---|
| `INTERNAL` | VisiBroker 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. |
| `INVALID_TRANSACTION` | Specified transaction was invalid (used in conjunction with VisiTransact). |
| `MARSHAL` | Error marshalling parameter or result. |
| `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. |
| `TRANSIENT` | Transient failure. |
| `TRANSACTION_MODE` | Mismatch detected between the `TransactionPolicy` in the IOR and the current transaction mode (used in conjunction with VisiTransact). |
| `TRANSACTION_REQUIRED` | Transaction is required (used in conjunction with VisiTransact). |
| `TRANSACTION_ROLLEDBACK` | Transaction was rolled back (used in conjunction with VisiTransact). |
| `TRANSACTION_UNAVAILABLE` | Connection to the VisiTransact Transaction Service has been abnormally terminated. |
| `TIMEOUT` | Request timeout. |
| `UNKNOWN` | Unknown exception. |

For a listing of explanations and possible causes of the above exceptions, see "CORBA exceptions."

## 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 sample below illustrates the `CompletionStatus` enumerated values for the CompletionStatus. `COMPLETED_MAYBE` is returned when the status of the operation cannot be determined.

```
enum CompletionStatus {
   COMPLETED_YES = 0;
   COMPLETED_NO = 1;
```

```
    COMPLETED_MAYBE = 2;
};
```

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

```
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.

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

## Determining the type of a system exception

The design of the VisiBroker 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.

## Catching system exceptions

Your applications should enclose the VisiBroker ORB and remote calls in a try catch block. The code samples below illustrate how the account client program, discussed in "Developing an example application with VisiBroker," prints an exception.

```
#include "Bank_c.hh"
int main(int argc, char* const* argv) {
   try {
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
      PortableServer::ObjectId_var managerId =
            PortableServer::string_to_ObjectId("BankManager");
      Bank::AccountManager_var manager =
            Bank::AccountManager::_bind("/bank_agent_poa", managerId);
      const char* name = argc > 1 ? argv[1] : "Jack B. Quick";
      Bank::Account_var account = manager->open(name);
      CORBA::Float balance = account->balance();
      cout << "The balance in " << name << "'s account is $" << balance <<
endl;
   } catch(const CORBA::Exception& e) {
      cerr << e << endl;
      return 1;
   }
   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.

```
prompt>Client
Exception: CORBA::OBJECT_NOT_EXIST
         Minor: 0
         Completion Status: NO
```

## Downcasting exceptions to a system exception

You can modify the account client program to attempt to downcast any exception that is caught to a `SystemException`. The following code sample shows you how to modify the client program.

```
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB.
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      // Bind to an account.
      Account_var account = Account::_bind();
      // Get the balance of the account.
      CORBA::Float acct_balance = account->balance();
      // Print out the balance.
      cout << "The balance in the account is $"
            << acct_balance << endl;
      } catch(const CORBA::Exception& e) {
      CORBA::SystemException* sys_excep;
         sys_excep = CORBA::SystemException::_downcast((CORBA::Exception*)&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;
         }
      }
}
```

The following code sample displays the resulting output if a system exception occurs.

```
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. The following code sample show this technique.

```
...
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB.
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      // Bind to an account.
      Account_var account = Account::_bind();
      // Get account balance.
      CORBA::Float acct_balance = account->balance();

      // Print out the balance.
      cout << "The balance in the account is $"                << acct_balance
<< endl;
   }
   // Check for system errors
   catch(const CORBA::SystemException& sys_excep) {
      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;
   }
}
...
```

## User exceptions

When you define your object's interface in IDL, you can specify the user exceptions that the object may raise. The following code sample shows the `UserException` code from which the `idl2cpp` compiler will derive the user exceptions you specify for your object.

```
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 account application, introduced in “Developing an example application with VisiBroker,” 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.

```
// 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.

```
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 {
            raise *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.

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

### Catching user exceptions

When an object implementation raises an exception, the VisiBroker 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 to the code as shown below.

```
...
   try {
      // Initialize the ORB.
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      // Bind to an account.
      Account_var account = Account::_bind();
      // Get the balance of the account.
      CORBA::Float acct_balance = account->balance();
   }
   catch(const Account::AccountFrozen& e) {
      cerr << "AccountFrozen returned:" << endl;
      cerr << e << endl;
      return(0);
   }
   // Check for system errors
   catch(const CORBA::SystemException& sys_excep) {
   }
...
```

### Adding fields to user exceptions

You can associate values with user exceptions. The code sample below 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.

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

# 7

# Server basics

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

## Overview

The basic steps that you'll perform in setting up your server are:

- Initialize the VisiBroker ORB
- Create and setup the POA
- Activate the POA Manager
- Activate objects
- Wait for client requests

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

## Initializing the VisiBroker ORB

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

```
// Initialize the VisiBroker 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*) did not permit portable object server code. A new specification was developed by the OMG to address these issues and the *Portable Object Adapter* (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 *rootPOA*) is supplied by each VisiBroker 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.

```
// 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` which 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 you need a different behavior, such as different lifespan policy, you will need to create a new POA.

POAs are created as children of existing POAs using `create_POA`. You can create as many POAs as you think are required.

**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.

```
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_agent_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 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`

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

`The following shows the AccountManager` implementation on the server side.

```
class AccountManagerImpl : public POA_Bank::AccountManager {
   private:
      Dictionary _accounts;
   public:
      virtual Bank::Account_ptr open(const char* name) {
         // Lookup the account in the account dictionary.
         Bank::Account_ptr account = (Bank::Account_ptr) _accounts.get(name);
         if(account == Bank::Account::_nil()) {
            // Make up the account's balance, between 0 and 1000 dollars.
            float balance = abs(rand()) % 100000 / 100.0;
            // Create the account implementation, given the balance.
            AccountImpl *accountServant = new AccountImpl(balance);
            try {
               // Activate it on the default POA which is root POA for this
                           servant
               PortableServer::POA_var rootPOA = _default_POA();
               CORBA::Object_var obj =
                           rootPOA->servant_to_reference(accountServant);
               account = Bank::Account::_narrow(obj);
            } catch(const CORBA::Exception& e) {
               cerr << "_narrow caught exception: " << e << endl;
            }
            // Print out the new account.
            cout << "Created " << name << "'s account: " << account << endl;
            // Save the account in the account dictionary.
            _accounts.put(name, account);




         }
```

```
            // Return the account.
            return Bank::Account::_duplicate(account);
        }
};
```

## Creating and Activating the Servant

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*.

```
// 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
myPOA->activate_object_with_id(managerId,&managerServant);
```

### 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. You can specify a POA Manager to use, or let the system create a new one for you by passing a `null` value as the POA Manager name in `create_POA()`).

```
// Activate the POA manager
rootPOA.the_POAManager().activate();
```

## 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 "Using POAs." For now, just be aware that there are several means for activating objects.

## Waiting for client requests

Once your POA is set up, you can wait for client requests by using `orb.run()`. This process will run until the server is terminated.

```
// Wait for incoming requests
orb.run();
```

## Complete example

The samples below shows the complete example code.

```
// Server.C
#include "Bank_s.hh"
#include <math.h>
class Dictionary {
   private:
      struct Data {
         const char* name;
         void* value;
      };
      unsigned _count;
      Data* _data;
   public:
      Dictionary() {
         _count = 0;
      }
      void put(const char* name, void* value) {
         Data* oldData = _data;
         _data = new Data[_count + 1];
         for(unsigned i = 0; i < _count; i++) {
            _data[i] = oldData[i];
         }
         _data[_count].name = strdup(name);
         _data[_count].value = value;
         _count++;
      }
      void* get(const char* name) {
         for(unsigned i = 0; i < _count; i++) {
            if(!strcmp(name, _data[i].name)) {
               return _data[i].value;
            }
         }
         return 0;
      }
};
class AccountImpl : public POA_Bank::Account {
   private:
      float _balance;
   public:
      AccountImpl(float balance) {
         _balance = balance;
      }
      virtual float balance() {
         return _balance;
      }


};
class AccountManagerImpl : public POA_Bank::AccountManager {
   private:
      Dictionary _accounts;
   public:
      virtual Bank::Account_ptr open(const char* name) {
         // Lookup the account in the account dictionary.
         Bank::Account_ptr account = (Bank::Account_ptr) _accounts.get(name);
         if(account == Bank::Account::_nil()) {
            // Make up the account's balance, between 0 and 1000 dollars.
            float balance = abs(rand()) % 100000 / 100.0;
            // Create the account implementation, given the balance.
            AccountImpl *accountServant = new AccountImpl(balance);
            try {
            // Activate it on the default POA which is root POA for this
               servant
               PortableServer::POA_var rootPOA = _default_POA();
               CORBA::Object_var obj =
                        rootPOA->servant_to_reference(accountServant);
               account = Bank::Account::_narrow(obj);
            } catch(const CORBA::Exception& e) {
               cerr << "_narrow caught exception: " << e << endl;
            }
            // Print out the new account.
            cout << "Created " << name << "'s account: " << account << endl;
            // Save the account in the account dictionary.
            _accounts.put(name, account);
         }
         // Return the account.
         return Bank::Account::_duplicate(account);
      }
};
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB.
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      // 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);
      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_agent_poa",
            rootManager, policies );
      // 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
      myPOA->activate_object_with_id(managerId,&managerServant);
      // Activate the POA Manager
      rootPOA->the_POAManager()->activate();
      cout << myPOA->servant_to_reference(&managerServant) << " is ready" <<

endl;
      // Wait for incoming requests
      orb->run();
   } catch(const CORBA::Exception& e) {
      cerr << e << endl;
```

```
    }
}
```

# 8

# Using POAs

## What is a Portable Object Adapter?

Portable Object Adapters replace Basic Object Adapters; they provide portability on the server side.

A POA is the intermediary between the implementation of an object and the VisiBroker ORB. In its role as an intermediary, a POA routes requests to servants and, as a result may cause servants to run and create child POAs if necessary.

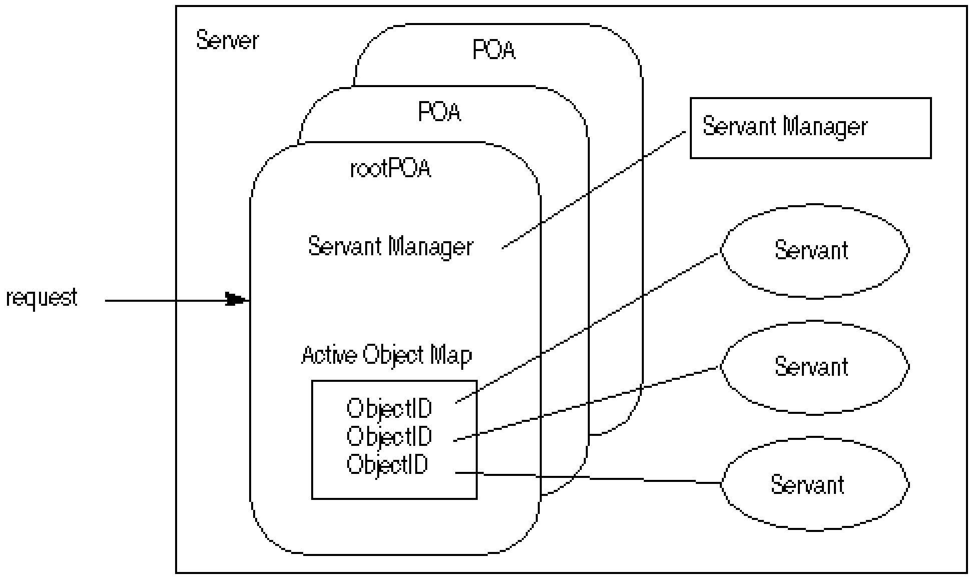
Servers can support multiple POAs. At least one POA must be present, which is called the rootPOA. The rootPOA is created automatically for you. The set of POAs is hierarchical; all POAs have the rootPOA as their ancestor.

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.

**Note**

Users familiar with versions of VisiBroker prior to 6.0 should note the change in inheritance hierarchy to support CORBA Specification 3.0, which requires local interfaces. For example, a ServantLocator implementation would now extend from `org.omg.PortableServer._ServantLocatorLocalBase` instead of `org.omg.PortableServer.ServantLocatorPOA`.

**Figure 8.1** Overview of the POA



## POA terminology

Following are definitions of some terms with which you will become more familiar as you read through this section.

| Term | Description |
|---|---|
| Active Object Map | Table that maps active VisiBroker 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. |
| incarnate | Associate 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 server process that created 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 VisiBroker 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 process that created it. |

## Steps for creating and using POAs

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

1. Define the POA's policies.
2. Create the POA.
3. Activate the POA through its POA manager.
4. Create and activate servants.
5. Create and use servant managers.
6. 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 lists the POA policies, their values, and the default value (used by the rootPOA).

- **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:** (Default) The POA is responsible for assigning requests to threads. In a multi-threaded environment, concurrent requests may be delivered using multiple threads. Note that VisiBroker uses multi-threading model.
  - **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.
  - **MAIN_THREAD_MODEL:** Calls are processed on a distinguished “main” thread. Requests for all main-thread POAs are processed sequentially. In a multi-threaded environment, all calls processed by all POAs with this policy are thread-safe. The application programmer designates the main thread by calling ORB::run() or ORB::perform_work(). For more information about these methods, see “Activating objects”.

- **Lifespan policy** The lifespan policy specifies the lifespan of the objects implemented in the POA.

  The lifespan policy can have the following values:

  - **TRANSIENT:** (Default) 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 process in which it was first created. Requests invoked on a persistent object may result in the implicit activation of a process, a POA and the servant that implements the object.

- **Object ID Uniqueness policy** The Object ID Uniqueness policy allows a single servant to be shared by many abstract objects.

  The Object ID Uniqueness policy can have the following values:

  - **UNIQUE_ID:** (Default) 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.
- **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:** (Default) 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 for persistent objects, and SYSTEM_ID is for transient objects. If you want to use SYSTEM_ID for persistent objects, you can extract them from the servant or object reference.
- **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:** (Default) 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. NON_RETAIN must be used with ServantLocators.

  ServantActivators and ServantLocators are types of servant managers. For more information on servant managers, see Using servants and servant managers.
- **Request Processing policy** The Request Processing policy specifies how requests are processed by the POA.
  - **USE_ACTIVE_OBJECT_MAP_ONLY:** (Default) 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.
  - **USE_DEFAULT_SERVANT:** If the Object ID is not listed in the Active Object Map or the NON_RETAIN policy is set, the request is dispatched to the default servant. If no default servant has been registered, an OBJ_ADAPTER exception is returned. The POA must also use the MULTIPLE_ID policy with this value.
  - **USE_SERVANT_MANAGER:** If the Object ID is not listed in the Active Object Map or the NON_RETAIN policy is set, the servant manager is used to obtain a servant.
- **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:

  - **IMPLICIT_ACTIVATION:** The POA supports implicit activation of servants. There are two ways to activate the servants as follows:
    - Converting them to an object reference with `POA::servant_to_reference()` .
    - Invoking `_this()` on the servant.

    The POA must also use the SYSTEM_ID and RETAIN policies with this value.
  - **NO_IMPLICIT_ACTIVATION:** (Default) The POA does not support implicit activation of servants.
- **Bind Support policy** The Bind Support policy (a VisiBroker-specific policy) controls the registration of POAs and active objects with the VisiBroker 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:** (Default) Only POAs are registered with the osagent. The POA must also use the PERSISTENT policy with this value.
- **NONE:** Neither POAs nor active objects are registered with the smart agent.

**Note**

The rootPOA is created with `NONE` activation policy.

## 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 VisiBroker ORB initialization. This POA uses the default POA policies described earlier in this section.

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 previous example) indicates the rootPOA. If the BindSupport: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 back slash (\\) when recording the names internally. For example, if you have coded two POAs in the following hierarchy,

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

then the 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.

```
// 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 policies

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".

```
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:

```
POA 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 will 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. The following code sample is an 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
PortableServer::POAManager_var rootManager = rootPOA->the_POAManager();
PortableServer::POA_var myPOA =
 rootPOA->create_POA("bank_agent_poa", rootManager, policies);
```

## Activating objects

When CORBA objects are associated with an active servant, if the POA's Servant Retention Policy is RETAIN, the associated object ID is recorded in the Active Object Map and the object is activated. 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 by 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.

This code sample is an 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
myPOA->activate_object_with_id(managerId,&managerServant);
// Activate the POA Manager
PortableServer::POAManager_var rootManager = rootPOA->the_POAManager();
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 below.

## 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:

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

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

If the POA has `IdUniquenessPolicy::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.

This is an example of activating all objects with the same servants

```
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB.
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      PortableServer::Current_var cur = PortableServer::Current::_instance();
      // 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(3);
      // Create policies for our persistent POA
      policies[(CORBA::ULong)0] =
            rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);
      policies[(CORBA::ULong)1] =
rootPOA>create_request_processing_policy(PortableServer::USE_DEFAULT_SERVANT);
      policies[(CORBA::ULong)2] =
            rootPOA->create_id_uniqueness_policy(PortableServer::MULTIPLE_ID);
      // Create myPOA with the right policies
      PortableServer::POAManager_var rootManager = rootPOA->the_POAManager();
      PortableServer::POA_var myPOA =
            rootPOA->create_POA("bank_default_servant_poa",
rootManager,policies);
      // Set the default servant
      AccountManagerImpl managerServant(cur);
      myPOA->set_servant( &managerServant );
      // Activate the POA Manager
      rootManager->activate();

      // Generate two references - one for checking and another for savings.
      //Note that we are not creating any
      // servants here and just manufacturing a reference which is not
      // yet backed by a servant
      PortableServer::ObjectId_var an_oid =
            PortableServer::string_to_ObjectId("CheckingAccountManager");
      CORBA::Object_var cref = myPOA->create_reference_with_id(an_oid.in(),
            "IDL:Bank/AccountManager:1.0");
      an_oid = PortableServer::string_to_ObjectId("SavingsAccountManager");
      CORBA::Object_var sref = myPOA->create_reference_with_id(an_oid.in(),
            "IDL:Bank/AccountManager:1.0");
      // Write out Checking reference
      CORBA::String_var string_ref = orb->object_to_string(cref.in());
      ofstream crefFile("cref.dat");
      crefFile << string_ref << endl;
      crefFile.close();
      // Now write out the Savings reference
      string_ref = orb->object_to_string(sref.in());
      ofstream srefFile("sref.dat");
      srefFile << string_ref << endl;
      srefFile.close();
      cout << "Bank Manager is ready" << endl;

 // Wait for incoming requests
orb->run();
    }
    catch(const CORBA::Exception& e) {
    cerr << e << endl;
    }
    return 1;
}
```

## Deactivating objects

A POA can remove a servant from its Active Object Map. This may occur, for example, as a 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.

This is an example of deactivating an object:

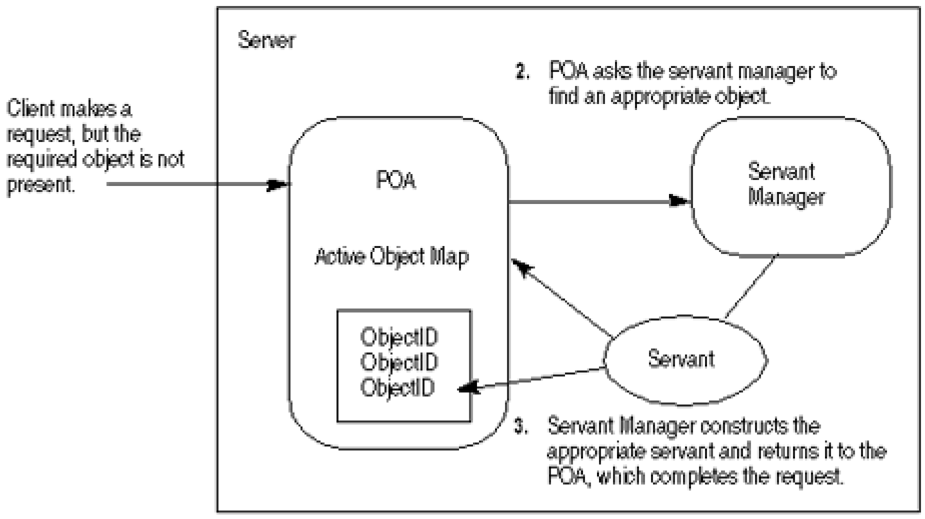
```
// DeActivatorThread
class DeActivatorThread: public VISThread {
   private :
      PortableServer::ObjectId _oid;
      PortableServer::POA_ptr  _poa;
   public :
      virtual ~DeActivatorThread(){}
      // Constructor
      DeActivatorThread(const PortableServer::ObjectId& oid,
            PortableServer::POA_ptr 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 );
      }
};
// 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
            throw CORBA::OBJECT_NOT_EXIST();
         // Create a deactivator thread
         new DeActivatorThread( oid, poa );
         // return the servant
         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;
      }
};
```

## Using servants and servant managers

Servant managers perform two types of operations: find and return a servant, and 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 server loads all objects at startup. Servant managers may also inform clients to forward requests to another object using the `ForwardRequest` exception.

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 8.2** Example servant manager function


Server
2. POA asks the servant manager to find an appropriate object.
Client makes a request, but the required object is not present.
POA
Active Object Map
Servant Manager
ObjectID
ObjectID
ObjectID
Servant
3. Servant Manager constructs the appropriate servant and returns it to the POA, which completes the request.

There are two types of servant managers: *ServantActivator* and *ServantLocator*. The type of policy already in place determines which type of servant manager is used. For more information on POA policy, see "POA policies". Typically, a Servant Activator activates persistent objects and a Servant Locator 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 Servant Activator or `ServantRetentionPolicy::NON_RETAIN` for Servant Locator.)

## 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 ServantActivators:

1. A client request is received (client request contains POA name, the object ID, and a few others.)
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 servant ID is entered into the active object map, and the response is returned to the client.

**Note**

The `etherealize` 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".

This code sample is an example of servant activator-type servant manager:

```
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB.
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      // 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(2);
      policies[(CORBA::ULong)0] =
            rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);
      policies[(CORBA::ULong)1] =
            rootPOA->create_request_processing_policy(
                  PortableServer::USE_SERVANT_MANAGER);
      // Create myPOA with the right policies
      PortableServer::POAManager_var rootManager = rootPOA->the_POAManager();
      PortableServer::POA_var myPOA =
            rootPOA->create_POA("bank_servant_activator_poa", rootManager,
                  policies);
      // Create a Servant activator
      AccountManagerActivator servant_activator_impl;
      // Set the servant activator
      myPOA->set_servant_manager(&servant_activator_impl);
      // Generate two references - one for checking and another for savings.
      // Note that we are not creating any
      // servants here and just manufacturing a reference which is not
      // yet backed by a servant
      PortableServer::ObjectId_var an_oid =
            PortableServer::string_to_ObjectId("CheckingAccountManager");
      CORBA::Object_var cref = myPOA->create_reference_with_id(an_oid.in(),
            "IDL:Bank/AccountManager:1.0");
      an_oid = PortableServer::string_to_ObjectId("SavingsAccountManager");
      CORBA::Object_var sref = myPOA->create_reference_with_id(an_oid.in(),
            "IDL:Bank/AccountManager:1.0");
      // Activate the POA Manager
      rootManager->activate();

      // Write out Checking reference
      CORBA::String_var string_ref = orb->object_to_string(cref.in());
      ofstream crefFile("cref.dat");
      crefFile << string_ref << endl;
      crefFile.close();
      // Now write out the Savings reference
      string_ref = orb->object_to_string(sref.in());
      ofstream srefFile("sref.dat");
      srefFile << string_ref << endl;
      srefFile.close();
      // Waiting for incoming requests
      cout << " BankManager Server is ready" << endl;
      orb->run();
   }
   catch(const CORBA::Exception& e) {
      cerr << e << endl;
   }
   return 1;
}
```

The servant manager for the servant activator example follows:

```
// 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
         throw CORBA::OBJECT_NOT_EXIST();
      // Create a deactivator thread
      new DeActivatorThread( oid, poa );
      // return the servant
      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 ServantLocators:

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.

This is some example server code illustrating servant locator-type servant managers:

```
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB.
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      // 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(3);
      // 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.
      policies[(CORBA::ULong)0] =
            rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);
      policies[(CORBA::ULong)1] =
            rootPOA->create_servant_retention_policy(PortableServer::
              NON_RETAIN);
      policies[(CORBA::ULong)2] =



   rootPOA->create_request_processing_policy(PortableServer::
        USE_SERVANT_MANAGER);
      PortableServer::POAManager_var rootManager = rootPOA->the_POAManager();
      PortableServer::POA_var myPOA =
            rootPOA-
>create_POA("bank_servant_locator_poa",rootManager,policies);
      // Create the servant locator
      AccountManagerLocator servant_locator_impl;
      myPOA->set_servant_manager(&servant_locator_impl);
      // Generate two references - one for checking and another for savings.
      // Note that we are not creating any
      // servants here and just manufacturing a reference which
      // is not yet backed by a servant
      PortableServer::ObjectId_var an_oid =
            PortableServer::string_to_ObjectId("CheckingAccountManager");
      CORBA::Object_var cref = myPOA->create_reference_with_id(an_oid.in(),
            "IDL:Bank/AccountManager:1.0");
      an_oid = PortableServer::string_to_ObjectId("SavingsAccountManager");
      CORBA::Object_var sref = myPOA->create_reference_with_id(an_oid.in(),
            "IDL:Bank/AccountManager:1.0");
      // Activate the POA Manager
      rootManager->activate();

      // Write out Checking reference
      CORBA::String_var string_ref = orb->object_to_string(cref.in());
      ofstream crefFile("cref.dat");
      crefFile << string_ref << endl;
      crefFile.close();
      // Now write out the Savings reference
      string_ref = orb->object_to_string(sref.in());
      ofstream srefFile("sref.dat");
      srefFile << string_ref << endl;
      srefFile.close();
      // Wait for incoming requests
      cout << "Bank Manager is ready" << endl;
      orb->run();
   }
   catch(const CORBA::Exception& e) {
      cerr << e << endl;
   }
   return 1;
}
```

The servant manager for this example follows:

```
// 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
            throw 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
         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;
      delete the_servant;
   }
};
```

## Managing POAs with the POA manager

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

A POA manager is associated with a POA when the POA is created. POA Managers can be created implicitly, by passing a nil `POAManager` reference to the create_POA operation

```
PortableServer::POAManager_var rootManager = rootPOA->the_POAManager();
PortableServer::POA_var myPOA = rootPOA->create_POA(
"bank_servant_locator_poa", rootManager, policies );
PortableServer::POA_var myPOA = rootPOA.create_POA( "bank_servant_locator_poa",
null, policies );
```

They can also be created explicitly using a POA Manager Factory

```
PortableServer::POAManagerFactory_var poaMgrFactory = rootPOA->
the_POAManagerFactory();
PortableServer::POAManager_var poaMgr = poaMgrFactory->
create_POAManager("MyPOAManager", null);
PortableServer::POA_var myPOA = rootPOA->create_POA("bank_agent_poa", poaMgr,
policies);
```

Explicit creation of a POA Manager permits application control of the POA Manager's identity, whereas implicit creation results in creation of a unique identity by the ORB run-time. There is a single instance of POA Manager Factory in an ORB and is created with root POA. It can also be used to get the list of all POA Managers in an ORB.

### Getting the current state

To get the current state of the POA manager, use

```
enum State{HOLDING, ACTIVE, DISCARDING, INACTIVE};
State get_state();
```

### 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 (in boolean 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. It is up to you to set-up thread monitoring if so desired.

To change the POA manager to the discarding state, use

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

The `wait_for_completion` option 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 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 cannot change to any other state.

To change the POA manager to the inactive state, use

```
void deactivate (in boolean etherealize_objects, in boolean
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 `ServantRetentionPolicy::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.

## Listening and Dispatching: Server Engines, Server Connection Managers, and their properties

**Note**

Policies that cover listener and dispatcher features previously supported by the BOA are not supported by POAs. In order to provide these features, a VisiBroker-specific policy (ServerEnginePolicy) can be used.

Visibroker provides a very flexible mechanism to define and tune endpoints for Visibroker servers. An endpoint in this context is a destination for a communication channel for clients to communicate with servers. A *Server Engine* is a virtual abstraction for connection endpoint provided as a configurable set of properties.

A ServerEngine abstraction can provide control in terms of:

– types of connection resources

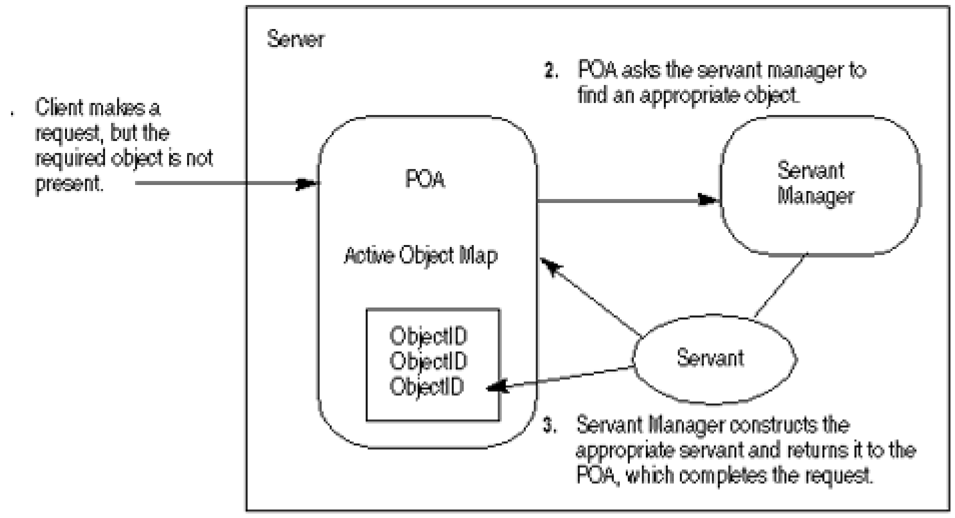
– connection management

– threading model and request dispatching

### Server Engine and POAs

A POA on Visibroker can have many-to-many relationship with a ServerEngine. A POA can be associated with many ServerEngines and vice-versa. The manifestation of this fact is that a POA, and hence the CORBA objects on the POA, can support multiple communication channels.

**Figure 8.3** Server engine overview


Server
. Client makes a request, but the required object is not present.
2. POA asks the servant manager to find an appropriate object.
POA
Active Object Map
ObjectID
ObjectID
ObjectID
Servant Manager
Servant
3. Servant Manager constructs the appropriate servant and returns it to the POA, which completes the request.

The simplest case is where POAs have their own unique single server engine. Here, requests for different POAs arrive on different ports. A POA can also have multiple server engines. In this scenario, a single POA supports requests coming from multiple input ports.

Notice that POAs can share server engines. When server engines are shared, the POAs listen to the same port. Even though the requests for (multiple) POAs arrive at the same port, they are dispatched correctly because of the POA name embedded in the request. This scenario occurs, for example, when you use a default server engine and create multiple POAs (without specifying a new server engine during the POA creation).

Server Engines are identified by a name and is defined the first time its name is introduced. By default Visibroker defines three Server Engine names. They are:

- iiop_tp: TCP transport with thread pool dispatcher
- iiop_ts: TCP transport with thread per session dispatcher
- iiop_tm: TCP transport with main thread dispatcher

Additionally, VisiBroker for C++ defines following Server Engines:

- liop_tp: Local ICP transport with thread pool dispatcher
- liop_ts: Local ICP transport with thread per session dispatcher
- liop_tm: Local ICP transport with main thread dispatcher

Two more Server Engines, boa_tp and boa_ts, are available for BOA backward compatibility.

## Associating a POA with a Server Engine

The default Server Engine associated with POA can be changed by using the property `vbroker.se.default`. For example, setting

```
vbroker.se.default=MySE
```

defines a new server engine with the name `MySE`. Root POA and all child POAs created will be associated with this Server Engine by default.

A POA can also be associated with a particular ServerEngine explicitly by using the `SERVER_ENGINE_POLICY_TYPE` POA policy. For example:

```
// create ServerEngine policy value
CORBA::Any_var se(new CORBA::Any);
CORBA::StringSequence_var engines =
  new CORBA::StringSequence(1UL);
engines->length(1UL);
engines[(CORBA::ULong)0] = CORBA::string_dup("MySE");
se <<= engines;

// create POA policies
CORBA::PolicyList_var policies =
  new CORBA::PolicyList(2UL);
policies->length(2UL);
policies[(CORBA::ULong)0] =
  orb->create_policy(
  PortableServerExt::SERVER_ENGINE_POLICY_TYPE,
  se);
policies[(CORBA::ULong)1] =
  rootPOA->create_lifespan_policy(
  PortableServer::PERSISTENT);

// create POA with policies
PortableServer::POA_var myPOA = rootPOA->create_POA(
  "bank_se_policy_poa", manager,
  policies);
```

The POA has an IOR template, profiles for which, are obtained from the Server Engines associated with it.

If you don't specify a server engine policy, the POA assumes a server engine name of `iiop_tp` and uses the following default values:

```
vbroker.se.iiop_tp.host=null
vbroker.se.iiop_tp.proxyHost=null
vbroker.se.iiop_tp.scms=iiop_tp
vbroker.se.liop_tp.host=null
vbroker.se.liop_tp.proxyHost=null
vbroker.se.liop_tp.scms=liop_tp
```

To change the default server engine policy, enter its name using the `vbroker.se.default` property and define the values for all the components of the new server engine. For example:

```
vbroker.se.default=abc,def
vbroker.se.abc.host=cob
vbroker.se.abc.proxyHost=null
vbroker.se.abc.scms=cobscm1,cobscm2
vbroker.se.def.host=gob
vbroker.se.def.proxyHost=null
vbroker.se.def.scms=gobscm1
```

### Defining Hosts for Endpoints for the Server Engine

Since Server Engines help define a connection's endpoints, the following properties are provided to specify their hosts:

- `vbroker.se.<se-name>.host=<host-URL>`: `vbroker.se.mySE.host=host.borland.com`, for example.
- `vbroker.se.<se-name>.proxyHost=<proxy-host-URL-or-IP-address>`: `vbroker.se.mySE.proxyHost=proxy.borland.com`, for example.

The `proxyHost` property can also take an IP address as its value. Doing so replaces the default hostname in the IOR with this IP address.

The endpoint abstraction of ServerEngine is further fine-grained in terms of configurable set of entities referred to as Server Connection Managers (SCM). A ServerEngine can have multiple SCMs. SCMs are not shareable between ServerEngines. SCMs are also identified using a name and are defined for a ServerEngine using:

```
vbroker.se.<se-name>.scms=<SCM-name>[,<SCM-name>,...]
```

**Note**

the `iiop_tp` and `liop_tp` Server Engines have SCMs named iiop_tp and liop_tp created for them, respectively.

## Server Connection Managers

The Server Connection Manager defines the configurable components of an endpoint. Its responsibilities are connection resource management, listening for requests, and dispatching requests to its associated POA. Three logical entities, defined through property groups, are provided by the SCM to fulfill these responsibilities:

- Manager
- Listener
- Dispatcher

Each SCM has one Manager, Listener, and Dispatcher. All three, when defined, form a single endpoint definition allowing clients to contact servers.

### Manager

Manager is a set of properties defining the configurable portions of a connection resource. VisiBroker provides a manager of type Socket.

Additionally, VisiBroker for C++ defines another manager of type `Local`. The `Local` type corresponds to `Local` IPC connections, while the `Socket` manager type expects TCP connections. To select either `Local` or `Socket`, set the following property:

```
vbroker.se.<se-name>.scm.<scm-name>.manager.type=Local|Socket
```

You can specify the maximum number of concurrent connections acceptable to the server endpoint using the `connectionMax` property:

```
vbroker.se.<se-name>.scm.<scm-name>.manager.connectionMax=<integer>
```

Setting `connectionMax` to `0` (zero) indicates that there is no restriction on the number of connections, which is the default setting.

You specify the maximum number of idle seconds using the `connectionMaxIdle` property:

```
vbroker.se.<se-name>.scm.<scm-name>.manager.connectionMaxIdle=<seconds>
```

Setting `connectionMaxIdle` to `0` (zero) indicates that there is no timeout, which is the default setting.

Garbage collection time can also be specified for the manager to garbage-collect idled connections. (Connections can idle after the `connectionMaxIdle` time until they are garbage-collected.) You can use the `garbageCollectTimer` property to specify the period of garbage collection in seconds:

```
vbroker.se.<se-name>.scm.<scm-name>.manager.garbageCollectTimer=<seconds>
```

A value of `0` (zero) means that the connection will never be garbage collected.

## Listener

The Listener is the SCM component that determines how and where the SCM listens for messages. Like the Manager, the Listener is also a set of properties. VisiBroker defines a IIOP listener for the TCP connections.

Additionally, VisiBroker for C++ defines a LIOP listerner for local IPC connections. You specify which type of listener you want to use with the property:

```
vbroker.se.<se-name>.scm.<scm-name>.listener.type=LIOP|IIOP
```

Since listeners are close to the actual underlying transport mechanism, their properties are not portable across listener types. Each listener type has its own set of properties, defined below.

### LIOP listener properties

For systems using shared memory Local IPC, the `shmSize` property is used to control the shared memory size, in bytes:

```
vbroker.se.<se-name>.scm.<scm-name>.listener.shmSize=<bytes>
```

If the shared memory-mapped file needs to be hidden in a directory accessible only by the user, the following boolean property needs to be set:

```
vbroker.se.<se-name>.scm.<scm-name>.listener.userConstrained=true|false
```

### IIOP listener properties

IIOP listners need to define a port and (if desired) a proxy port in conjunction with their hosts. These are set using the `port` and `proxyPort` properties, as follows:

```
vbroker.se.<se-name>.scm.<scm-name>.listener.port=<port>
vbroker.se.<se-name>.scm.<scm-name>.listener.proxyPort=<proxy-port>
```

**Note**

If you do not set the port property (or set it to `0` [zero]), a random port will be selected. A `0` value for the `proxyPort` property means that the IOR will contain the actual port (defined by the `listener.port` property or selected by the system randomly). If it is not required to advertise the actual port, set the proxy port to a non-zero (positive) value.

Setting properties to define standard TCP socket options is also supported for send|receive buffer sizes, socket lingering time, and whether or not to keep inactive sockets alive. The following properties are provided for these mechanisms:

```
vbroker.se.<se-name>.scm.<scm-name>.listener.rcvBuffSize=<bytes>
vbroker.se.<se-name>.scm.<scm-name>.listener.sendBuffSize=<bytes>
vbroker.se.<se-name>.scm.<scm-name>.listener.socketLinger=<seconds>
vbroker.se.<se-name>.scm.<scm-name>.connection.keepAlive=true|false
```

If for any reason you wish to simply use your system's defaults for the TCP socket properties, simply set the appropriate property to a value of 0 (zero).

### Dispatcher

The Dispatcher defines a set of properties that determine how the SCM dispatches requests to threads. Three types of dispatchers are provided: `ThreadPool`, `ThreadSession`, and `MainThread`. You set the dispatcher type with the `type` property:

```
vbroker.se.<se-name>.scm.<scm-name>.dispatcher.type=ThreadPool|ThreadSession|
MainThread
```

Further control is provided through the SCM for the ThreadPool dispatcher type. The ThreadPool defines the minimum and maximum number of threads that can be created in the thread pool, as well as the maximum time in seconds after which an idled thread is destroyed. These values are controlled with the following properties:

```
vbroker.se.<se-name>.scm.<scm-name>.dispatcher.threadMin=<integer>
vbroker.se.<se-name>.scm.<scm-name>.dispatcher.threadMax=<integer>
vbroker.se.<se-name>.scm.<scm-name>.dispatcher.threadMaxIdle=<seconds>
```

The ThreadPool dispatcher allows a “cooling time” to be set. A thread is said to be “hot” when the GIOP connection being served is potentially readable, either upon creation of the connection or upon the arrival of a request. After the cooling time (in seconds), the thread can be returned to the thread pool.

The following property is used to set the cooling time:

```
vbroker.se.<se-name>.scm.<scm-name>.dispatcher.coolingTime=<seconds>
```

## When to use these properties

There are many times where you need to change some of the server engine properties. The method for changing these properties depends on what you need. For example, suppose you want to change the port number. You could accomplish this by:

- Changing the default `listener.port` property
- Creating a new server engine

Changing the default `listener.port` property is the simplest method, but this affects all POAs that use the default server engine. This may or may not be what you want.

If you want to change the port number on a specific POA, then you'll have to create a new server engine, define the properties for this new server engine, and then reference the new server engine when creating the POA. The previous sections show how to update the server engine properties. The following code snippet shows how to define properties of a server engine and create a POA with a user-defined server engine policy:

```
// static initialization
AccountRegistry AccountManagerImpl::_accounts;
int main(int argc, char* const* argv)
{
   try {
   // Initialize the orb
   CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
   // Get the property manager; notice the value returned is not placed into a
   // 'var' type.
   VISPropertyManager_ptr pm = orb->getPropertyManager();
   pm->addProperty("vbroker.se.mySe.host", "");
   pm->addProperty("vbroker.se.mySe.proxyHost", "");
   pm->addProperty("vbroker.se.mySe.scms", "scmlist");
   pm->addProperty("vbroker.se.mySe.scm.scmlist.manager.type", "Socket");
   pm->addProperty("vbroker.se.mySe.scm.scmlist.manager.connectionMax", 100UL);
   pm->addProperty("vbroker.se.mySe.scm.scmlist.manager.connectionMaxIdle",
      300UL);
   pm->addProperty("vbroker.se.mySe.scm.scmlist.listener.type", "IIOP");
   pm->addProperty("vbroker.se.mySe.scm.scmlist.listener.port", 55000UL);
   pm->addProperty("vbroker.se.mySe.scm.scmlist.listener.proxyPort", 0UL);
   pm->addProperty("vbroker.se.mySe.scm.scmlist.dispatcher.type", "ThreadPool");
```

```
      pm->addProperty("vbroker.se.mySe.scm.scmlist.dispatcher.threadMax", 100UL);
      pm->addProperty("vbroker.se.mySe.scm.scmlist.dispatcher.threadMin", 5UL);
      pm->addProperty("vbroker.se.mySe.scm.scmlist.dispatcher.threadMaxIdle",
         300UL);
      // Get a reference to the root POA
      CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
      PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);
      // Create the policies
      CORBA::Any_var seAny(new CORBA::Any);
      // The SERVER_ENGINE_POLICY_TYPE requires a sequence, even if
      // only one engine is being specified.
      CORBA::StringSequence_var engines = new CORBA::StringSequence(1UL);
      engines->length(1UL);
      engines[0UL] = CORBA::string_dup("mySe");
      seAny <<= engines;
      CORBA::PolicyList_var policies = new CORBA::PolicyList(2UL);
      policies->length(2UL);
      policies[0UL] = orb->create_policy(
         PortableServerExt::SERVER_ENGINE_POLICY_TYPE, seAny);
      policies[1Ul] = rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);
      // Create our POA with our policies
      PortableServer::POAManager_var manager = rootPOA->the_POAManager();
      PortableServer::POA_var myPOA = rootPOA->create_POA(
         "bank_se_policy_poa", manager, policies);

      // Create the servant
      AccountManagerImpl* managerServant = new AccountManagerImpl();
      // Activate the servant
      PortableServer::ObjectId_var oid =
         PortableServer::string_to_ObjectId("BankManager");
      myPOA->activate_object_with_id(oid ,managerServant);
      // Obtain the reference
      CORBA::Object_var ref = myPOA->servant_to_reference(managerServant);
      CORBA::String_var string_ref = orb->object_to_string(ref.in());
      ofstream refFile("ref.dat");
      refFile << string_ref << endl;
      refFile.close();
      // Activate the POA manager
      manager->activate();
      // Wait for Incoming Requests
      cout << "AccountManager Server ready" << endl;
      orb->run();
      }
      catch(const CORBA::Exception& e) {
      cerr << e << endl;
   return (1);
   }
   return (0);
   }
```

## 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 included with the product.

## 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 VisiBroker ORB first locates the appropriate server, or starts the server if needed. It then locates the appropriate POA within that server.

Once the VisiBroker 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.

9

# Managing threads and connections

This section discusses the use of multiple threads in client programs and object implementations, and will help you understand the VisiBroker thread and connection model.

## Using threads

A *thread*, or a single sequential flow of control within a process, is also called a lightweight process that reduces overhead by sharing fundamental parts with other threads. Threads are lightweight so that there can be many of them present within a process.

Using multiple threads provides concurrency within an application and improves performance. Applications can be structured efficiently with threads servicing several independent computations simultaneously. For example, a database system may have many user interactions in progress while at the same time performing several file and network operations.

Although it is possible to write the software as one thread of control moving asynchronously from request to request, the code may be simplified by writing each request as a separate sequence, and letting the underlying system handle the synchronous interleaving of the different operations.

Multiple threads are useful when:

- There are groups of lengthy operations that do not necessarily depend on other processing (like painting a window, printing a document, responding to a mouse-click, calculating a spreadsheet column, signal handling).
- There will be few locks on data (the amount of shared data is identifiable and small).
- The task can be broken into various responsibilities. For example, one thread can handle the signals and another thread can handle the user interface.

Thread and connection management occurs within the scope of an entity known as a server engine. Several default server engines are created automatically by VisiBroker, which include thread pool engines for IIOP, for LIOP, and so forth. Additional server engines can be used and created in a VisiBroker server by applications. See the example in:

```
<install_dir>/examples/Vbroker/poa/server_engine_policy/Server.C
```

Server engines are created, configured, and used independently. The creation and configuration of one server engine does not affect other server engines in the same server. Usually, each server engine has one transport end point, called the *listen point/ socket*.

The relationship between server engines and POAs is many-to-many. Each server engine can be used by multiple POAs, and each POA may also use multiple server engines.

Server engines can consist of multiple Server Connection Managers (SCMs). An SCM is composed of managers, listeners, and dispatchers. The properties of managers, listeners and dispatchers can be configured to determine how the SCM functions. These properties are discussed in "Setting connection management properties".

## Listener thread, dispatcher thread, and worker threads

Each server engine has a listener and a dispatcher thread. The listener thread is responsible for:

- Accepting new connections. Therefore, it listens on the listen end-point.
- Monitoring readability on idle GIOP connections.
- Updating the monitoring list.
- Idle connection garbage collection based on property settings.

The dispatcher determines which threads to send requests.

Each server engine uses a certain number of worker threads to receive and process requests. Different requests may handled by different worker threads. For a given request, the request reading, processing (include server side interceptor intercepting), and replying are all handled by the same thread. The number of worker threads used by a server engine depends on:

- The thread model.
- The number of concurrent requests or connections.
- The property settings.

## Thread policies

The two major thread models supported by VisiBroker are the thread pool (also known as thread-per-request, or `TPool`) and thread-per-session (also known as thread-per-connection, or `TSession`). Single-thread and main-thread models are not discussed in this document. The thread pool and thread-per-session models differ in these fundamental ways:

- Situation in which they are created
- How simultaneous requests from the same client are handled
- When and how threads are released

The default thread policy is the thread pool. For information about setting thread-per-session or changing properties for the thread pool model, see "Setting dispatch policies and properties".

## Thread pool policy

When your server uses the thread pool policy, it defines the maximum number of threads that can be allocated to handle client requests. A worker thread is assigned for each client request, but only for the duration of that particular request. When a request is completed, the worker thread that was assigned to that request is placed into a pool of available threads so that it may be reassigned to process future requests from any of the clients.

Using this model, threads are allocated based on the amount of request traffic to the server object. This means that a highly active client that makes many requests to the server at the same time 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. Additionally, the overhead associated with the creation and destruction of worker threads is reduced, because threads are reused rather than destroyed, and can be assigned to multiple new connections.

VisiBroker conserves system resources by dynamically allocating the number of threads in the thread pool based on the number of concurrent client requests by default. If the client becomes very active, new threads are allocated to meet its needs. If threads remain inactive, VisiBroker releases them, only keeping enough threads to meet current client demand. This enables the optimal number of threads to be active in the server at all times.

The size of the thread pool grows based upon server activity and is fully configurable, either before or during execution, to meet the needs of specific distributed systems. With the thread pool model, you can configure the following:

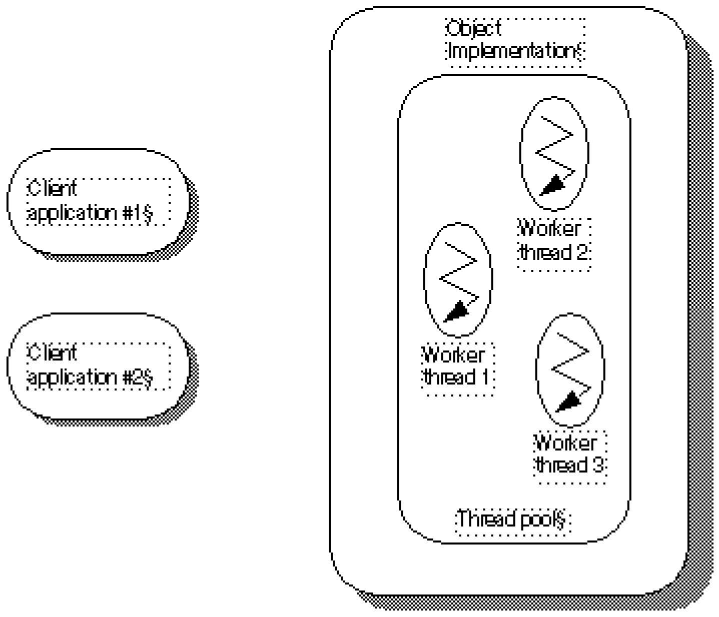
- Maximum and minimum number of threads
- Maximum idle time

Each time a client request is received, an attempt is made to assign a thread from the thread pool to process the request. If this is the first client request and the pool is empty, a thread will be created. Likewise, if all threads are busy, a new thread will be created to service the request.

A server can define a maximum number of threads that can be allocated to handle client requests. If there are no threads available in the pool and the maximum number of threads have already been created, the request will block until a thread currently in use has been released back into the pool.

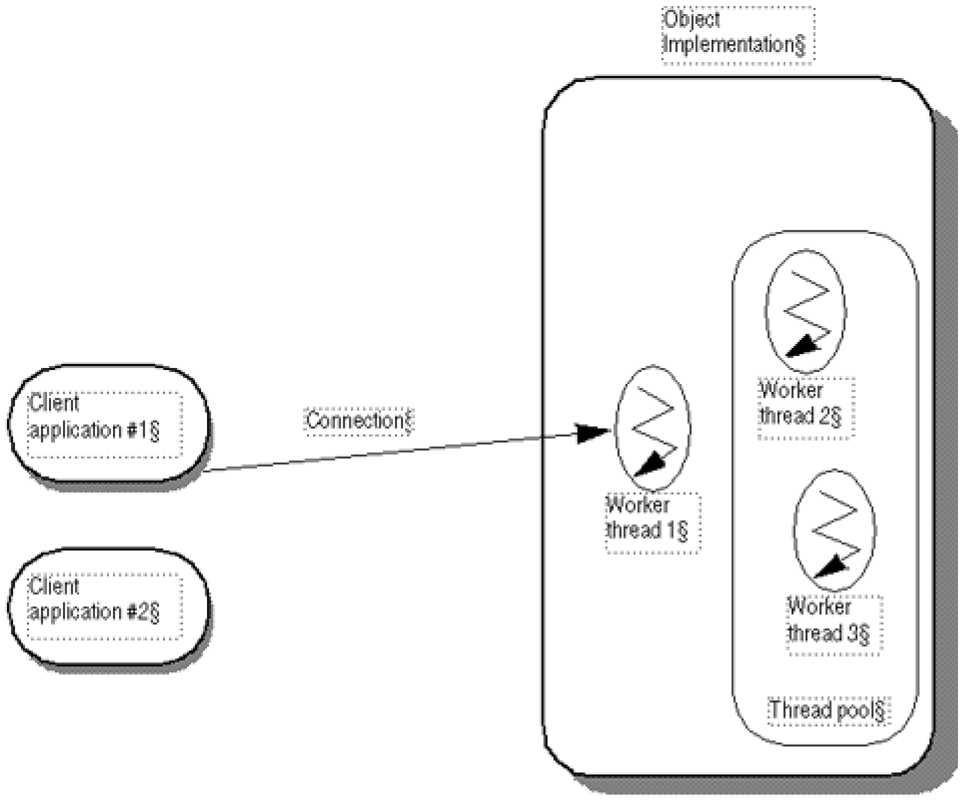
Thread pool is the default thread policy. You do not have to set up anything to define this environment. If you want to set properties for the thread pool, see “Setting dispatch policies and properties”.

**Figure 9.1** Pool of threads is available



The figure above shows the object implementation using the thread pool policy. As the name implies, there is an available pool of worker threads in this policy.

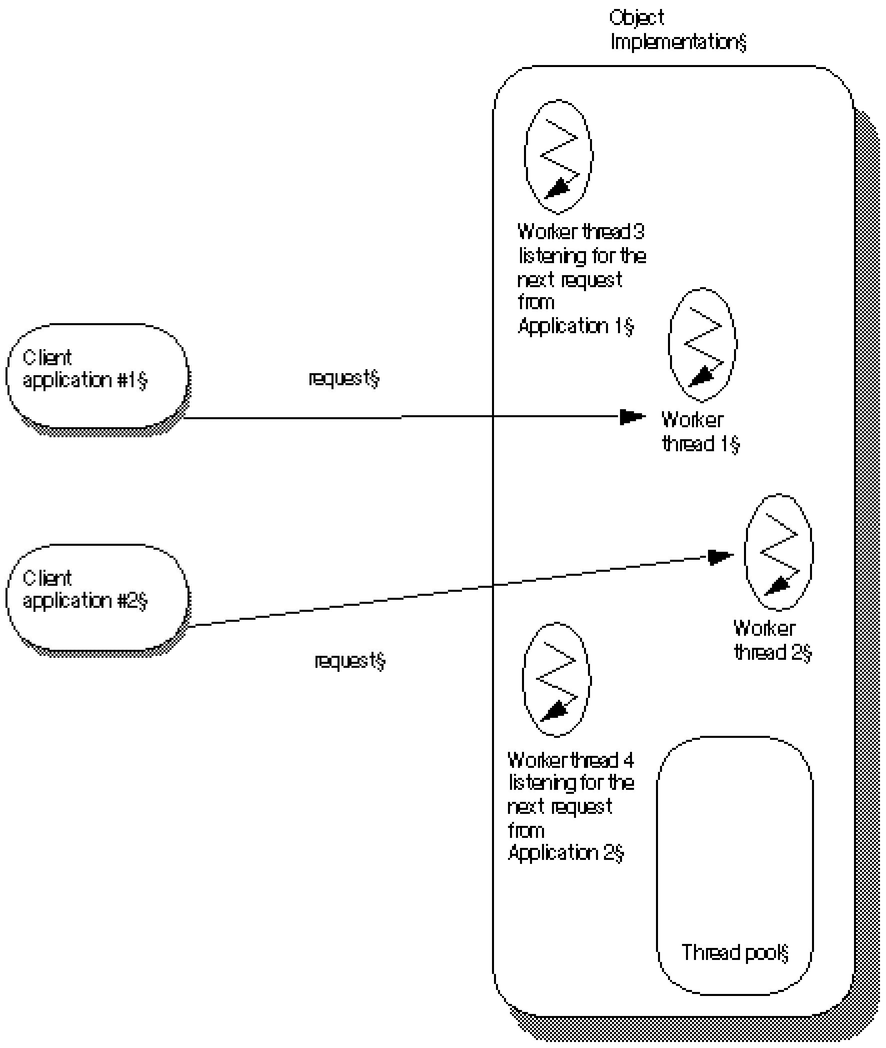
**Figure 9.2** Client application #1 sends a request



In the above figure, Client application #1 establishes a connection to the Object Implementation and a thread is created to handle requests. In the thread pool, there is one connection per client and one thread per connection. When a request comes in, a worker thread receives the request; that worker thread is no longer in the pool.

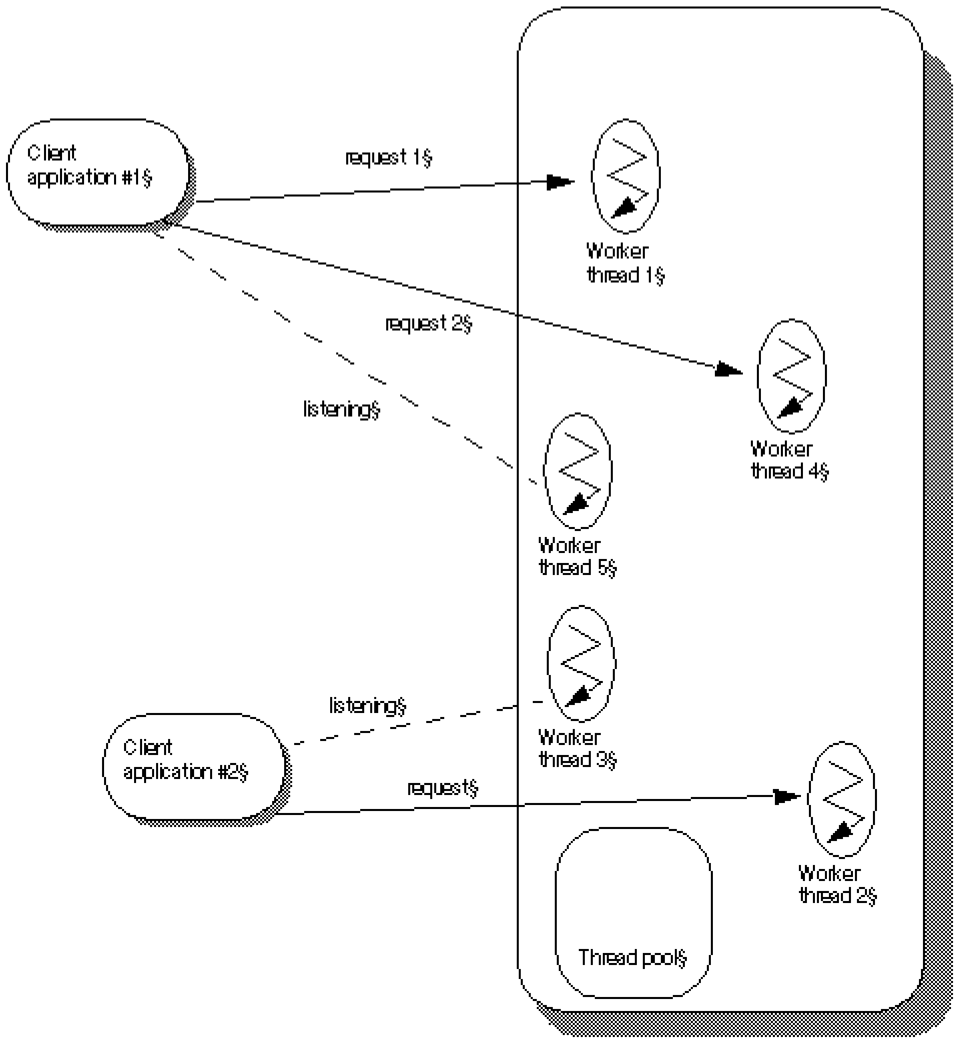
A worker thread is removed from the thread pool and is always listening for requests. When a request comes in, that worker thread reads in the request and dispatches the request to the appropriate object implementation. Prior to dispatching the request, the worker thread wakes up one other worker thread which then listens for the next request.

**Figure 9.3** Client application #2 sends a request


Object Implementation§
Worker thread 3 listening for the next request from Application 1§
Client application #1§
request§
Worker thread 1§
Client application #2§
request§
Worker thread 2§
Worker thread 4 listening for the next request from Application 2§
Thread pool§

As the above figure shows, when Client application #2 establishes its own connection and sends a request, a second worker thread is created. Worker thread #3 is now listening for incoming requests.

**Figure 9.4** Client application #1 sends a second request

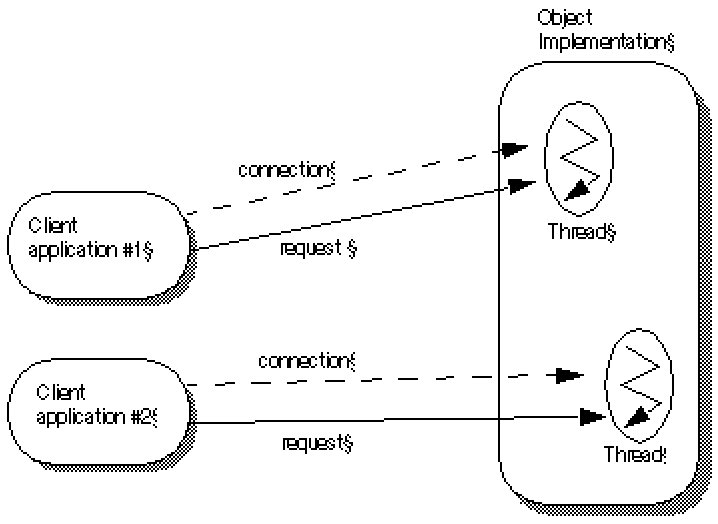


The above figure shows that when a second request comes in from Client application #1, it uses worker thread #4. Worker thread #5 is spawned to listen for new requests. If more requests came in from Client application #1, more threads would be assigned to handle them, each spawned after the listening thread receives a request. As worker threads complete their tasks, they are returned to the pool and become available to handle requests from any client.

## Thread-per-session policy

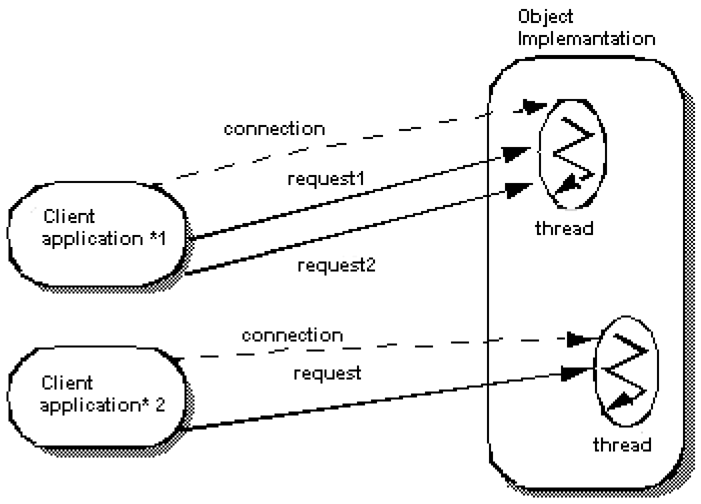
With the thread-per-session (`TSession`) policy, threading is driven by connections between the client and server processes. When your server selects the thread-per-session policy, a new thread is allocated each time a new client connects to a server. A thread is assigned to handle all the requests received from a particular client. Because of this, thread-per-session is also referred to as thread-per-connection. When the client disconnects from the server, the thread is destroyed. You may limit the maximum number of threads that can be allocated for client connections by setting the `vbroker.se.iiop_ts.scm.iiop_ts.manager.connectionMax` property.

**Figure 9.5** Object implementation using the thread-per-session policy



The above figure shows the use of the thread-per-session policy. The Client application #1 establishes a connection with the object implementation. A separate connection exists between Client application #2 and the object implementation. When a request comes in to the object implementation from Client application #1, a worker thread handles the request. When a request from Client application #2 comes in, a different worker thread is assigned to handle this request.

**Figure 9.6** Second request comes in from the same client



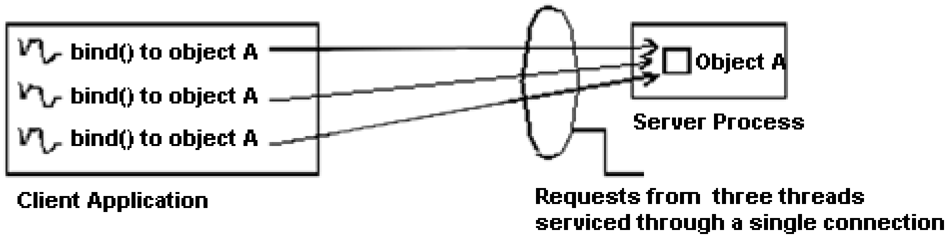
In the above figure, a second request has come in to the object implementation from Client application #1. The same thread that handles request 1 will handle request 2. The thread blocks request 2 until it completes request 1. (With thread-per-session, requests from the same Client are not handled in parallel.) When request 1 has completed, the thread can handle request 2 from Client application #1. Multiple requests may come in from Client application #1. They are handled in the order that they come in; no additional threads are assigned to Client application #1.

## Connection management

Overall, VisiBroker's connection management minimizes the number of client connections to the server. In other words there is only one connection per server process which is shared. All requests from a single client application 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 server.

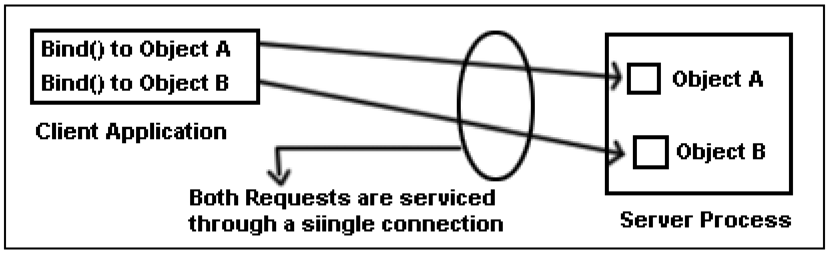
In the following scenario, a client application is bound to two objects in the server process. Each `bind()` shares a common connection to the server process, even though the `bind()` is for a different object in the server process.

**Figure 9.7** Binding to two objects in the same server process



The following figure shows the connections for a client using multiple threads that has several threads bound to an object on the server.

**Figure 9.8** Binding to an object in a server process



As the above figure shows, all invocations from all threads are serviced by the same connection. For that scenario, the most efficient multi threading model to use is the thread pool model. If the thread-per-session model is used in this scenario, only one thread on the server will be allocated to service all requests from all threads in the client application, which could easily result in poor performance.

The maximum number of connections to a server, or from a client, can be configured. Inactive connections will be recycled when the maximum is reached, ensuring resource conservation.

## ServerEngines

Thread and connection management on the server side is performed by ServerEngines, which can consist of one or more Server Connection Managers (SCMs). An SCM is a collection of properties of the manager, listener, and dispatcher.

Defining a ServerEngine consists of specifying a set of properties in a properties file. For example, if on UNIX the property file called `myprops.properties` is in home directory, the command line is

```
prompt> vbj -DORBpropStorage=~/myprops.properties myServer
```

### ServerEngine properties

```
vbroker.se.<srvr_eng_name>.scms=<srvr_connection_mngr_name1>,<srvr_connection_m
ngr_name2>
```

The set of Server Connection Managers associated with a ServerEngine is defined by this property. The name specified in the above property as the `<svr_eng_name>` is the name of the ServerEngine. The SCMs listed here will be the list of SCMs for the associated server engine. SCMs cannot be shared between ServerEngines. However, ServerEngines can be shared by multiple POAs.

The other properties are

```
vbroker.se.<se>.host
```

The `host` property is the IP address for the server engine to listen for messages.

```
vbroker.se.<se>.proxyHost
```

The `proxyHost` property specifies the proxy IP address to send to the client in the case where the server does not want to publish its real hostname.

## Setting dispatch policies and properties

Each POA in a multi-threaded object server can choose between two dispatch models: *thread-per-session* or *thread pool*. You choose a dispatch policy by setting the `dispatcher.type` property of the `ServerEngine`.

```
vbroker.se.<srvr_eng_name>.scm.<srvr_connection_mngr_name>.dispatcher.type=
    ThreadPool
vbroker.se.<srvr_eng_name>.scm.<srvr_connection_mngr_name>.dispatcher.type=
    ThreadSession
```

For more information about these properties see "Using POAs" and the *VisiBroker Programmer's Reference*.

### Thread pool dispatch policy

`ThreadPool` (thread pooling) is the default dispatch policy when you create a POA without specifying the `ServerEnginePolicy`.

For `ThreadPool`, you can set the following properties:

– `vbroker.se.default.dispatcher.tp.threadMax`

  This property sets a `TPool` server engine's maximum number of worker threads in the thread pool. The property can be set statically on server startup or dynamically reconfigured using the property API. For instance, the start up property

  ```
  vbroker.se.default.dispatcher.tp.threadMax=32
  ```

  or

  ```
  vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMax=32
  ```

  sets the initial maximum worker thread limitation to 32 for the default `TPool` server engine. The default value of this property is unlimited (0). If there are no threads available in the pool and the maximum number of threads have already been created, the request is blocked until a thread currently in use has been released back into the pool.

– `vbroker.se.default.dispatcher.tp.threadMin`

  This property sets a `TPool` server engine's minimum number of worker threads in the thread pool. The property can be set statically on server startup or dynamically reconfigured using the property API. For instance, the start up property

  ```
  vbroker.se.default.dispatcher.tp.threadMin=8
  ```

  or

  ```
  vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMin=8
  ```

  sets the initial worker thread minimum number to 8 for the default `TPool` server engine. The default value of this property is 0 (no worker threads).

– `vbroker.se.default.dispatcher.tp.threadMaxIdle`

  This property sets a TPool server engine's idle thread check interval. The property can be set statically on server startup or dynamically reconfigured using the property API. For instance, the start up property

```
vbroker.se.default.dispatcher.tp.threadMaxIdle=120
```

or

```
vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMaxIdle=120
```

sets the initial idle worker thread check interval to 120 seconds for the default `TPool` server engine. The default value of this property is 300 seconds. With this setting, the server engine will check the idle state of each worker thread every 120 seconds. If a worker thread has been idle across two consecutive checks, it will be recycled (terminated) at the second check. Therefore, the actual idle thread garbage collection time is between 120 to 240 seconds under the above setting, instead of exactly 120 seconds.

– `vbroker.se.default.dispatcher.tp.coolingTime`

The `ThreadPool` dispatcher allows a "cooling time" to be set. A thread is said to be "hot" when the GIOP connection being served is potentially readable, either upon creation of the connection or upon the arrival of a request. After the cooling time (in seconds), the thread can be returned to the thread pool. The property can be set statically on server startup or dynamically reconfigured using the property API. For instance, the startup property

```
vbroker.se.default.dispatcher.tp.coolingTime=6
```

or

```
vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.coolingTime=6
```

sets the initial cooling time to 6 seconds for the default engine (or the IIOP `TPool` server engine).

The default value of this property is 3 seconds. The maximum value is 10 seconds.

**Note**

The `vbroker.se.default.xxx.tp.xxx` property is recommended when `vbroker.se.default=iiop_tp`. When using with ThreadSession, it is recommended that you use the `vbroker.se.iiop_ts.scm.iiop_ts.xxx` property.

## Thread-per-session dispatch policy

When using the `ThreadSession` as the dispatcher type, you must set the `se.default` property to `iiop_ts`.

```
vbroker.se.default=iiop_ts
```

**Note**

In thread-per-session, there are no `threadMin`, `threadMax`, `threadMaxIdle`, and `coolingTime` dispatcher properties. Only the Connection and Manager properties are valid properties for ThreadSession.

## Coding considerations

All code within a server that implements the VisiBroker ORB object must be thread-safe. You must take special care when accessing a system-wide resource within an object implementation. For example, many database access methods are not thread-safe. Before your object implementation attempts to access such a resource, it must first lock access to the resource using a synchronized block.

If serialized access to an object is required, you need to create the POA on which this object is activated with the `SINGLE_THREAD_MODEL` value for the `ThreadPolicy`.

## Setting connection management properties

The following properties are used to configure connection management. Properties whose names start with `vbroker.se` are server-side properties. The client side properties have their names starting with `vbroker.ce`.

**Note**

The command line options for VisiBroker 3.x backward-compatibility are less obvious in terms of whether they are client-side or server-side. However, the connection and thread management options that start with the `-ORB` prefix set the client-side options whereas the options with the `-OA` prefix are used for the server-side options. There are no common properties which are used for both client-side and server-side thread and connection management.

The distinction between client and server vanishes if callback or bidirectional GIOP is used.

- `vbroker.se.default.socket.manager.connectionMax`

  This property sets the maximum allowable client connections to a server engine. The property can be set statically on server startup or dynamically reconfigured using the property API. For instance, the start up property

  ```
  -Dvbroker.se.default.socket.manager.connectionMax=128
  ```

  or

  ```
  -Dvbroker.se.iiop_tp.scm.iiop_tp.manager.connectionMax=128
  ```

  sets the initial maximum connection limitation on this server engine to `128`. The default value of this property is unlimited (`0` [zero]). When the server engine reaches this limitation, before accepting a new client connection, the server engine needs to reuse an idle connection. This is called connection swapping. When a new connection arrives at the server, it will try to detach the oldest unused connection. If all the connections are busy, the new connection will be dropped. The client may retry again until some timeout expires.

- `vbroker.se.default.socket.manager.connectionMaxIdle`

  This property sets the maximum length of time an idle connection will remain open on a server engine. The property can be set statically on server startup or dynamically reconfigured using property API. For instance, the start up property

  ```
  -Dvbroker.se.default.socket.manager.connectionMaxIdle=300
  ```

  or

  ```
  -Dvbroker.se.iiop_tp.scm.iiop_tp.manager.connectionMaxIdle=300
  ```

  s

  ets the initial idle connection maximum lifetime to 300 seconds. The default value of this property is 0 (unlimited). When a client connection has been idle longer than this value, it becomes a candidate for garbage collection.

- `vbroker.ce.iiop.ccm.connectionMax`

  Specifies the maximum number of the total connections within a client. This is equal to active connections plus the ones that are cached. The default value of zero means that the client does not try to close any of the old active or cached connections. If a new client connection will result in exceeding the limit set by this property, the VisiBroker for C++ will try to release one of the cached connections. If there are no cached connections, it will try to close the oldest idle connection. If both of them fail, the `CORBA::NO_RESOURCE` exception will result.

## Valid values for applicable properties

The following properties have a fixed set or range of valid values:

– `vbroker.ce.iiop.ccm.type=Pool`

  Currently, `Pool` is the only supported type.

In the following properties, `xxx` is the server engine name and `yyy` is the server connection manager name:

– `vbroker.se.xxx.scm.yyy.manager.type=Socket`

  Other possible values are `Local` for LIOP and `BIDIR` for bidir (bidirectional) SCMs.

– `vbroker.se.xxx.scm.yyy.listener.type=IIOP`

  You can also use the value `LIOP` for local IPC and `SSL` for security.

– `vbroker.se.xxx.scm.yyy.disptacher.type=ThreadPool`

  The other possible values are `ThreadSession` and `MainThread`.

– `vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.coolingTime`

  The default value is 3 , and the maximum value is 10, so a value greater than 10 will be clamped to 10.

## Effects of property changes

The effect of a change in a property value depends on the actions associated with the properties. Most of the actions are directly or indirectly related to the utilization of system resources. The availability and restrictions of the system resources to the CORBA application vary depending on the system and the nature of the application.

For instance, increasing the garbage collector timer may increase the system activities, as the garbage collector will run more frequently. On the other hand, increasing its value means the idle threads will remain in system unclaimed for longer periods of time.

## Dynamically alterable properties

The following properties can be changed dynamically and the effect will be immediate unless stated otherwise:

```
vbroker.ce.iiop.ccm.connectionCacheMax=5
vbroker.ce.iiop.ccm.connectionMax=0
vbroker.ce.iiop.ccm.connectionMaxIdle=360
vbroker.ce.iiop.connection.rcvBufSize=0
vbroker.ce.iiop.connection.sendBufSize=0
vbroker.ce.iiop.connection.tcpNoDelay=false
vbroker.ce.iiop.connection.socketLinger=0
vbroker.ce.iiop.connection.keepAlive=true
vbroker.ce.liop.ccm.connectionMax=0
vbroker.ce.liop.ccm.connectionMaxIdle=360
vbroker.ce.liop.connection.rcvBufSize=0
vbroker.ce.liop.connection.sendBufSize=0
vbroker.se.iiop_tp.scm.iiop_tp.manager.connectionMax=0
vbroker.se.iiop_tp.scm.iiop_tp.manager.connectionMaxIdle=0
vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMin=0
vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMax=100
```

The new dispatcher threadMax properties will be reflected after the next garbage collector run.

```
vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMaxIdle=300
vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.coolingTime=3
vbroker.se.iiop_tp.scm.iiop_tp.manager.garbageCollectTimer=30
vbroker.se.liop_tp.scm.liop_tp.listener.userConstrained=false
```

### Determining whether property value changes take effect

For this purpose, the server manager needs to be enabled, using the property `vbroker.orb.enableServerManager=true`, and the properties can be obtained through the server manager query either through the Console or through a command-line utility.

### Impact of changing property values

It is very difficult to determine the impact of changing the value of a property to something other than the default. For thread and connection limits, the available system resources vary depending on the machine configuration and the number of other processes running. The setting of properties allows performance tuning for a given system.

## Garbage collection

A dispatcher's thread pool in VisiBroker has an idle timeout `vbroker.se.xxx.scm.xxx.dispatcher.threadMaxIdle`. The default value is `300` seconds, and after the idle timeout expires the dispatcher will remove any idle worker threads in the thread pool.

A Server Connection Manager (SCM) has its own garbage collection timeout `vbroker.se.xxx.scm.xxx.manager.garbageCollectTimer`. The default value is `30` seconds, and after the timeout expires any idle connections are garbage collected.

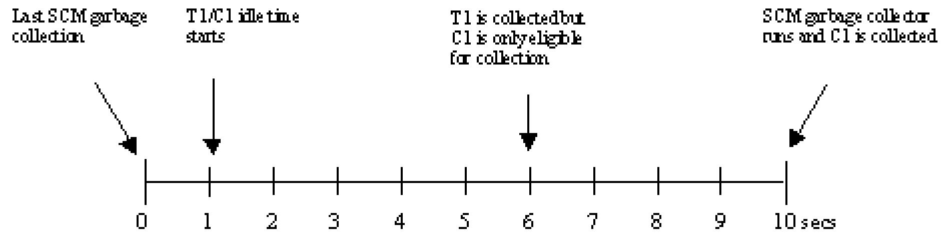
Since the SCM only garbage collects idle connections, the property `vbroker.se.xxx.scm.xxx.manager.connectionMaxIdle` needs to be set greater than `0` (zero) in order for connections to go to an idle state. The default value is `0` (zero), which means that a connection is never considered idle and nothing is collected, even if the SCM's garbage collection timeout expires.

The dispatcher and the SCM perform garbage collection independently and there is no garbage collection performed by the ORB itself. Hence given the values below.

```
vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMaxIdle=5
vbroker.se.iiop_tp.scm.iiop_tp.manager.connectionMaxIdle=5
vbroker.se.iiop_tp.scm.iiop_tp.manager.garbageCollectTimer=10
```

When the thread pool worker thread, T1, has been idle for `5` seconds it is immediately removed from the dispatcher's thread pool. The connection, `C1`, which has been idle for `5` seconds is only garbage collected by the SCM after `10` seconds.

**Figure 9.9** Collection of resources by the SCM GC


Last SCM garbage collection
T1/C1 idle time starts
T1 is collected but C1 is only eligible for collection
SCM garbage collector runs and C1 is collected
0
1
2
3
4
5
6
7
8
9
10 secs

On the Client side the Client Connection Manager's (CCM) cached connections can be given an idle timeout by setting the property `vbroker.ce.xxx.ccm.connectionMaxIdle`. The default value is `0` (zero), meaning that the cached connections do not have an idle timeout. Given an idle timeout, the idle cached connections in the connection pool/ cache are marked eligible for garbage collection. Unlike the SCM, the CCM has no garbage collection timer, however whenever any connection is being cached it will attempt to garbage collect any cached connections that are marked eligible for collection.

# 10

# Using the tie mechanism

This section describes how the tie mechanism may be used to integrate existing C++ code into a distributed object system. This section 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::Servant` . When it is not convenient or possible to alter existing classes to inherit from the VisiBroker servant class, the *tie* mechanism offers an attractive 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. The delegator implementation 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 files are generated from the IDL compiler:

– `<interface_name>POATie` defers implementation of all IDL defined methods to a delegate. The delegate implements the interface `<interface_name>Operations`. Legacy implementations can be trivially extended to implement the operations interface and in turn delegate to the real implementation.

– `<interface_name>Operations` defines all of the methods that must be implemented by the object implementation. This interface acts as the delegate object for the associated `<interface_name>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:

```
<install_dir>\vbroker\examples\basic\bank_tie
```

### Looking at the tie template

The `idl2cpp` compiler will automatically generate a `_tie_Account` template class, as shown in the code sample below. 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.

```
...
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

The code sample below shows the modifications to the `Server.C` file required to use the `_tie_account` class.

```
#include "Bank_s.hh"
#include <math.h>
...
int main(int argc, char* const* argv) {
  try {
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    // get a reference to the root POA
    PortableServer::POA_var rootPOA =
      PortableServer::POA::_narrow(
        orb->resolve_initial_references("RootPOA"));
    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
    AccountManagerImpl managerServant(rootPOA);
    // Create the delegator
    POA_Bank_AccountManager_tie<AccountManagerImpl>
      tieServer(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
    myPOA->activate_object_with_id(managerId, &tieServer);
    // Activate the POA Manager
    poa_manager->activate();
    cout << myPOA->servant_to_reference(&tieServer) <<
      " is ready" << endl;
    // Wait for incoming requests
    orb->run();


  } catch(const CORBA::Exception& e) {
    cerr << e << endl;
```

```
      return 1;
    }
    return 0;
  }
```

## Building the tie example

The instructions described in “Developing an example application with VisiBroker” are also valid for building the tie example.

# 11

# Client basics

This section describes how client programs access and use distributed objects.

## Initializing the VisiBroker ORB

The Object Request Broker (ORB) provides a communication link between the client and the server. When a client makes a request, the VisiBroker ORB locates the object implementation, activates the object if necessary, delivers the request to the object, and returns the response to the client. The client is unaware whether the object is on the same machine or across a network.

Though much of the work done by the VisiBroker ORB is transparent to you, your client program must explicitly initialize the VisiBroker ORB. VisiBroker ORB options, described in the *VisiBroker Programmer's Reference*, Programmer tools for C++ can be specified as command-line arguments. To ensure these options take effect you will need to pass the supplied `argc` and `argv` arguments to `ORB_init`. The code samples below illustrate the VisiBroker ORB initialization.

```
#include <fstream.h>
#include "Bank_c.hh"

int main(int argc, char* const* argv) {
  CORBA::ORB_var orb;
  CORBA::Float balance;
  try {
    // Initialize the ORB.
    orb = CORBA::ORB_init(argc, argv);
    ...
}
```

## 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()` method. The VisiBroker 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.

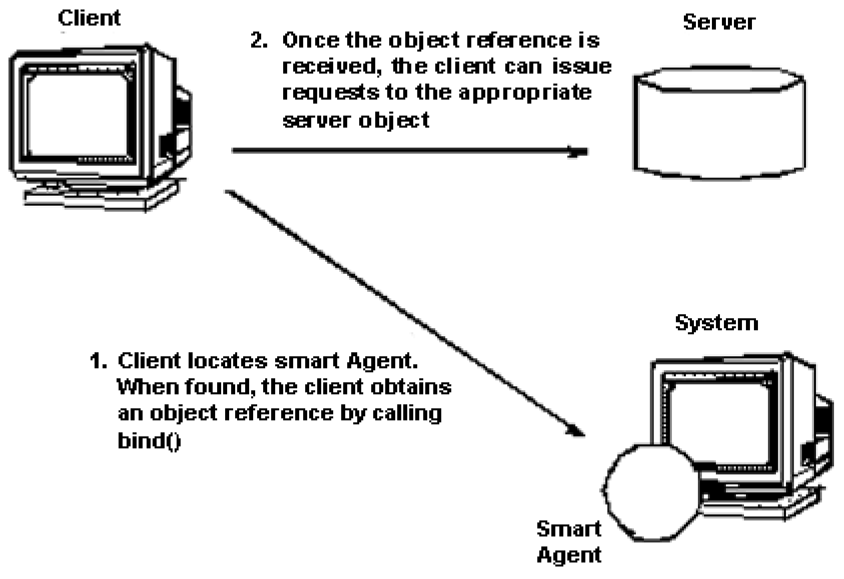
## Action performed during the bind process

When the server process starts, it performs `CORBA::ORB.init()` and announces itself to Smart Agents on the network.

When your client program invokes the `_bind()` method, the VisiBroker ORB performs several functions on behalf of your program.

- The VisiBroker ORB contacts the Smart Agent to locate an object implementation that offers the requested interface. If an object name is specified when `_bind()` is invoked, that name is used to further qualify the directory service search. The Object Activation Daemon (OAD), described in "Using the Object Activation Daemon (OAD)," may be involved in this process if the server object has been registered with the OAD.
- When an object implementation is located, the VisiBroker ORB attempts to establish a connection between the object implementation that was located and your client program.
- Once the connection is successfully established, the VisiBroker 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 11.1** Client interaction with the Smart Agent


Client
2. Once the object reference is received, the client can issue requests to the appropriate server object
Server
System
1. Client locates smart Agent. When found, the client obtains an object reference by calling bind()
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()` method.

```
PortableServer::ObjectId_var manager_id =
PortableServer::string_to_ObjectId("BankManager");
Bank::AccountManager_var =
  Bank::AccountManager::_bind("/bank_agent_poa", manager_id);
```

## 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.

The following example shows how to invoke an operation using an object reference:

```
// Invoke the balance operation.
balance = account->balance();
cout << "Balance is $" << balance << endl;
```

## Manipulating object references

The `_bind()` method returns a reference to a CORBA object to your client program. 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 methods that all VisiBroker 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` (zero) if the object reference is not `nil`.

```
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`.

```
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`.

```
class Object {
  ...
  static CORBA::Object_ptr _duplicate(CORBA::Object_ptr obj);
  ...
};
```

**Note**

The `_duplicate` member function has no meaning for the POA or VisiBroker 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.

```
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.

```
class CORBA {
  ...
  static void release();
  ...
};
```

## 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` (zero), VisiBroker automatically deletes the object reference. The code sample below 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.

```
class Object {
  ...
  CORBA::Long _ref_count() const;
  ...
};
```

## Converting a reference to a string

VisiBroker provides a VisiBroker ORB class with methods that allow 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.

| Method | Description |
|---|---|
| `object_to_string` | Converts an object reference to a string. |
| `string_to_object` | Converts a string to an object reference. |

A client program can use the `object_to_string` method 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` method, and use the object reference without having to explicitly bind to the object.

The caller of `object_to_string` is responsible for calling `CORBA::string_free()` on the returned string.

**Note**

Locally-scoped object references like the VisiBroker ORB or the POA cannot be stringified. If an attempt is made to do so, a `MARSHAL` exception is raised with the minor code `4`.

## Obtaining object and interface names

The table below shows the methods 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**

When you invoke `_bind()` without specifying an object name, invoking the `_object_name()` method with the resulting object reference will return `NULL` .

| Method | Description |
|---|---|
| `_interface_name` | Returns the interface name of this object. |
| `_object_name` | Returns this object's name. |
| `_repository_id` | Returns the repository's type identifier. |

## Determining the type of an object reference

You can use the `_hash()` member function 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.

You can check whether an object reference is of a particular type by using the `_is_a()` method. You must first obtain the repository id of the type you wish to check using the `_repository_id()` method. 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` (zero) if the object is not of the type specified. Note that this may require remote invocation to determine the type.

You can use `_is_equivalent()` to check if two object references refer to the same object implementation. This method returns 1 if the object references are equivalent. It returns `0` (zero) if the object references are distinct, but it does not necessarily indicate that the object references are two distinct objects. This is a lightweight method and does not involve actual communication with the server object.

| Method | Description |
|---|---|
| `_hash` | Returns a hash value for the object reference. |

| Method | Description |
|---|---|
| `_is_a` | Determines if an object implements a specified interface. |
| `_is_equivalent` | Returns `true` 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 `_is_bound()` to determine if the object bound. The method returns 1 if the object is bound and returns `0` (zero) if the object is not bound.

The `_is_local()` method returns 1 if the client program and the object implementation reside within the same process or address space where the method is invoked.

The `_is_remote()` method returns 1 if the client program and the object implementation reside in different processes, which may or may not be located on the same host.

| Method | Description |
|---|---|
| `_is_bound` | Determines if a connection is currently active for this object. |
| `_is_local` | Determines if this object is implemented in the local address space. |
| `_is_remote` | Determines if this object's implementation does not reside in the local address space. |

## 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 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*.

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 maintains a type graph 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`.

```
Account *acct;
Account *acct2;
Object *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*. The code sample below 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.

```
...
Account *acct;
CORBA::Object *obj;
acct = Account::_bind();
obj  = (CORBA::Object *)acct;...
```

## Using Quality of Service (QoS)

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 (QoS)

QoS policy management is performed through operations accessible in the following contexts:

– The VisiBroker 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 VisiBroker 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 VisiBroker ORB level.

**Note**

– 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 in at the VisiBroker ORB or thread level.

**Note**

The QoS policies installed at the ORB level will only affect those objects on which no method is called before installing the policies, for example a `non_existent` call internally makes a call on a server object. If ORB level QoS policies are installed after the `non_existent` call, then the policies do not apply.

#### 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 `org.omg.CORBA.INV_POLICY` exception is raised.

## QoS interfaces

The following interfaces are used to get and set QoS policies.

### 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.

### CORBA::Object

- `_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 VisiBroker 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 VisiBroker 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, `NO_PERMISSION` 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 `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'sPolicyCurrent is obtained by invoking ORB::resolve_initial_references() and specifying an identifier of PolicyCurrent.

## QoSExt::DeferBindPolicy

The DeferBindPolicy determines if the VisiBroker 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 values of DeferBindPolicy are true and false. If DeferBindPolicy is set to true all binds will be deferred until the first invocation of a binding instance. 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 before as an option to the Bind method on the generated helper classes.

The code sample below illustrates an example for creating a DeferBindPolicy and setting the policy on the VisiBroker ORB.

```
//Initialize the flag and references
CORBA::Boolean deferMode = (CORBA::Boolean) 1;
CORBA::Any policy_value;
policy_value <<= CORBA::Any::from_boolean(deferMode);

CORBA::Policy_var policy =
  orb->create_policy(QoSExt::DEFER_BIND_POLICY_TYPE, policy_value);

CORBA::PolicyList policies;
policies.length(1);
policies[0] = CORBA::Policy::_duplicate(policy);

// Get a reference to the thread manager
CORBA::Object_var obj = orb->resolve_initial_references("ORBPolicyManager");
CORBA::PolicyManager_var orb_mgr = CORBA::PolicyManager::_narrow(obj);

// Set the policy on the ORB level
orb_mgr->set_policy_overrides(policies, CORBA::SET_OVERRIDE);
```

## QoSExt::RelativeConnectionTimeoutPolicy

The RelativeConnectionTimeoutPolicy indicates a timeout after which attempts to connect to an object using one of the available endpoints is aborted. The timeout situation is likely to happen with objects protected by firewalls, where HTTP tunneling is the only way to connect to the object.

## Messaging::RebindPolicy

RebindPolicy is used to indicate whether the ORB may transparently rebind once successfully bound to a target. An object reference is considered bound once it is in a state where a LocateRequest message would result in a LocateReply message with status OBJECT_HERE. RebindPolicy accepts values of type Messaging::RebindMode and are set only on the client side. It can have one of six values that determine the behavior in the case of a disconnection, an object forwarding request, or an object failure after an object reference is bound. The supported values are:

– Messaging::TRANSPARENT allows the VisiBroker ORB to silently handle object-forwarding and necessary reconnections during the course of making a remote request.

– Messaging::NO_REBIND allows the VisiBroker 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 VisiBroker 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 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. If a client has received a `CloseConnection` message before, it will also reestablish the closed connection.
- `QoSExt::VB_NO_REBIND` does not enable failover. It only allows the client VisiBroker ORB to reopen a closed connection to the same server; it does not allow object forwarding of any kind.

**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**

If the Client has set `RebindPolicy` and the `RebindMode` is anything other that the default(`VB_TRANSPARENT`), then the `RebindPolicy` is propagated in a special `ServiceContext` as per the CORBA specification. The propagation of the `ServiceContext` occurs only when the client invokes the server through a `GateKeeper` or a `RequestAgent`. This propagation does not occur in a normal Client/Server scenario.

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 describes the behavior of the different `RebindMode` types.

| RebindMode type | Reestablish closed connection to the same object? | Allow object forwarding? | Object failover? |
|---|---|---|---|
| `NO_RECONNECT` | No, throws REBIND exception. | No, throws REBIND exception. | No |
| `NO_REBIND` | Yes | Yes, if policies match. No, throws REBIND exception. | No |
| `TRANSPARENT` | Yes | Yes | No |
| `VB_NO_REBIND` | Yes | No, throws REBIND exception. | No |
| `VB_NOTIFY_REBIND` | No, throws exception. | Yes | Yes. VB_NOTIFY_REBIND throws an exception after failure detection, and then tries a failover on subsequent requests. |
| `VB_TRANSPARENT` | Yes | Yes | Yes, transparently. |

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 Messaging section of the CORBA specification.

### Messaging::RelativeRequestTimeoutPolicy

The `RelativeRequestTimeoutPolicy` indicates the relative amount of time which a Request or its responding Reply may be delivered. After this amount of time, the Request is canceled. This policy applies to both synchronous and asynchronous invocations. Assuming the request completes within the specified timeout, the Reply will never be discarded due to timeout. The timeout value is specified in 100s of nanoseconds. This policy is only effective on established connections, and is not applicable to establishing a connection.

### Messaging::RelativeRoundTripTimeoutPolicy

The `RelativeRoundTripTimeoutPolicy` specifies the relative amount of time for which a Request or its corresponding Reply may be delivered. If a response has not yet been delivered after this amount of time, the Request is canceled. Also, if a Request had already been delivered and a Reply is returned from the target, the Reply is discarded after this amount of time. This policy applies to both synchronous and asynchronous invocations. Assuming the request completes within the specified timeout, the Reply will never be discarded due to timeout. The timeout value is specified in 100s of nanoseconds. This policy is only effective on established connections, and is not applicable to establishing a connection.

### Messaging::SyncScopePolicy

The `SyncScopePolicy` defines the level of synchronization for a request with respect to the target. Values of type `SyncScope` are used in conjunction with a `SyncScopePolicy` to control the behavior of one-way operations.

The default `SyncScopePolicy` is `SYNC_WITH_TRANSPORT`. To perform one-way operations via the OAD, you must use `SyncScopePolicy=SYNC_WITH_SERVER`. Valid values for `SyncScopePolicy` are defined by the OMG.

**Note**

Applications must explicitly set an VisiBroker ORB-level `SyncScopePolicy` to ensure portability across VisiBroker ORB implementations. When instances of `SyncScopePolicy` are created, a value of type `Messaging::SyncScope` is passed to `CORBA::ORB::create_policy`. This policy is only applicable as a client-side override.

## Exceptions

| Exception | Description |
|---|---|
| CORBA::INV_POLICY | Raised when there is an incompatibility between `Policy` overrides. |
| CORBA::REBIND | Raised when the `RebindPolicy` has a value of `NO_REBIND`, `NO_RECONNECT`, or `VB_NO_REBIND` and an invocation on a bound object references results in an object-forward or location-forward message. |
| CORBA::PolicyError | Raised when the requested `Policy` is not supported. |
| CORBA::InvalidPolicies | 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. |
| CORBA::COMM_FAILURE | Raised by the client side ORB if communication is lost while an operation is in progress. Potentially, the operation was completed. |
| CORBA::TRANSIENT | Raised by the client side ORB if it attempted to reach the object and failed. The operation was not successful. |

# 12

# Using IDL

This section describes how to use the CORBA interface description language (IDL).

## 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.

For more information see the *VisiBroker Programmer's Reference* Programmer's tools for C++ .

The OMG has defined specifications for such language mapping. Information about the language mapping is not covered in this manual since VisiBroker adheres to the specification set forth by OMG. If you need more information about language mapping, see the OMG web site at `http://www.omg.org`.

**Note**

The CORBA 3.0 formal specification can be found at: `http://www.omg.org/technology/documents/vault.htm#CORBA_IIOP`.

Discussions on the IDL can be quite extensive. Because VisiBroker 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.

### 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. The IDL sample below shows an IDL specification for an object named `example`. The `example` object has only one method, `op1`.

```
// IDL specification for the example object
interface example {
  long op1(in char x, out short y);
};
```

## Looking at generated code for clients

The code sample below shows how the IDL compiler generates two client files, `example_c.hh` and `example_c.cc` from the "Using IDL." 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 an `.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.

**Important**

Do not modify the contents of the files generated by the IDL compiler.

```
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

The code sample above shows the op1 method generated by the IDL complier, 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. The code sample below shows the type definition for the example class.

```
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.

```
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.

| Method | Description |
|---|---|
| example_var() | Constructor that initializes the_ptr to NULL. |
| example_var(example_ptr ptr) | Constructor that creates an object with the_ptr initialized to the argument passed. The var invokes release() on _ptr at the time of destruction. When the _ptr's reference count reaches 0, that object will be deleted. |
| example_var(const example_var& var) | Constructor that makes a copy of the object passed as a parameter var and points _ptr to the newly copied object. |
| ~example() | Destructor that invokes _release() once on the object to which _ptr points. |
| operator=(example_ptr p) | Assignment operator invokes _release() on the object to which _ptr points and then stores p in _ptr. |

| Method | Description |
|---|---|
| operator=(const example_ptr p) | Assignment operator invokes _release() on the object to which _ptr points and then stores a _duplicate() of p in _ptr. |
| example_ptr operator->() | Returns the _ptr stored in this object. This operator should not be called until this object has been properly initialized. |

## Looking at generated code for servers

The code sample below 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. The POA_example class is derived from the PortableServer_ServantBase class.

**Important**

You should not modify the contents of the files generated by the IDL compiler.

```
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 below 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. The sample below shows the template class generated by the IDL compiler 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 "Using the tie mechanism."

You may also generate a `_ptie` template for integrating an object database with your servers.

## Defining interface attributes in 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.

The IDL sample below shows an IDL specification that defines two attributes, one read-write and one read-only.

```
interface Test {
  attribute long count;
  readonly attribute string name;
};
```

The following code sample shows the operations class generated for the interface declared in the IDL.

```
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 one-way methods with no return value

IDL allows you to specify operations that have no return value, called *one-way* 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 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**

One-way operations cannot raise exceptions or return values.

```
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.

```
interface parent {
  void operation1();
};
interface child : parent {
   ...
  long operation2(in short s);
};
```

The code sample below shows the code that is generated from the interface specification shown above.

```
class parent : public virtual CORBA::Object {
   ...
  void operation1( );
   ...
};
class child : public virtual parent {
   ...
  CORBA::Long operation2(CORBA::Short s);
   ...
};
```

# 13

# Using the Smart Agent

This section describes the Smart Agent (`osagent`), which client programs register with in order to find object implementations. It explains how to configure your own VisiBroker ORB domain, connect Smart Agents on different local networks, and migrate objects from one host to another.

## What is the Smart Agent?

VisiBroker'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. 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. Like client programs, the communication with the Smart Agent is completely transparent to the object implementation. For more information about POAs, see "Using POAs."

### Best practices for Smart Agent configuration and synchronization

While the Smart Agent imposes no hard limits on the numbers and types of objects that it can support, there are reasonable best practices that can be followed when incorporating the it into a larger architecture.

The Smart Agent is designed to be a lightweight directory service with a flat, simple namespace, which can support a small number of well known objects within a local network.

Since all objects' registered services are stored in memory, scalability cannot be optimized and be fault tolerant at the same time. Applications should use well known objects to bootstrap to other distributed services so as not to rely on the Smart Agent for all directory needs. If a heavy services lookup load is necessary, it is advisable to use the VisiBroker Naming Service (VisiNaming). VisiNaming provides persistent storage capability and cluster load balancing whereas the Smart Agent only provides a simple round robin on a per `osagent` basis. Due to the in-memory design of the Smart Agent, if it is terminated by a proper shutdown or an abnormal termination, it does not failover to another Smart Agent in the same ORB domain, that is to the same `OSAGENT_PORT` number, whereas the VisiNaming Service provides such failover functionality. For more information on the VisiBroker naming service, see "Using the VisiNaming Service."

## General guidelines

The following are some general guidelines for best practice Smart Agent usage.

- Server registrations should be limited to less than 100 object instances or POAs per ORB domain.
- The Smart Agent keeps track of all clients (not just CORBA servers), so every client creates a small load on the Smart Agent. Within any 10 minute period, the client population should generally not exceed 100 clients.

Note

The GateKeeper counts as one client even though it is acting on behalf of many real clients.

- Applications should use the Smart Agent sparsely by binding to small sets of well known objects at startup and then using those objects for further discovery. The Smart Agent communications are based on UDP. Although the message protocol built on top of UDP is reliable, UDP is often not reliable or allowed in wide area networks. Since the Smart Agent is designed for intranet use, it is not recommended over wide area networks that involve firewall configurations.
- The real default IP of the Smart Agent must be accessible to clients on a subnet that is not directly connected to the Smart Agent host. The Smart Agent cannot be configured for client access behind a Network Address Translation (NAT) firewall.
- The Smart Agent configures itself at startup using the network information available at that time. It is not able to detect new network interfaces that are added later, such as interfaces associated with a dial up connection. Therefore, the Smart Agent is meant for use in static network configurations.

## Load balancing/ fault tolerance guidelines

- The Smart Agent implements load balancing using a simple round-robin algorithm on a per agent basis, not on an ORB domain basis. For load balancing between server replicas, when you have more than one Smart Agent in the ORB domain, make sure all servers are registered with the same Smart Agent.
- The ORB runtime caches access to the Smart Agent, so multiple binds to the same server object from the same ORB process do not result in round-robin behavior because all subsequent attempts to bind to the object us the cache rather than sending a new request to the Smart Agent. This behavior can be changed using ORB properties. For more information, refer to the *VisiBroker Programmer's Reference*, "Using VisiBroker properties."
- When a Smart Agent is terminated, all servers that were registered with that agent attempt to locate another agent with which to register. This process is automatic, but may take up to two minutes for the server to perform this function. During that two minute window, the server is not registered in the ORB domain and therefore is not available to new clients. However, this does not affect ongoing IIOP communications between the server and clients that were previously bound.

## Location service guidelines

The location service is built upon the Smart Agent technology. Therefore, the location service is subject to the same guidelines described above.

- The location service triggers generate UDP traffic between the Smart Agent and the trigger handlers registered by applications. Use of this feature should be limited to less than 10 objects, monitored by less than 10 processes.

- The location service triggers fire when the Smart Agent determines that an object is available or down. There may be a delay of up to four minutes for a “down” trigger to fire. For this reason, you may not want to use this feature for time critical applications.

For more information about the Location Service, refer to “Using the Location Service.”

### When not to use a Smart Agent

- When the ORB domain spans a large number (greater than 5) of subnets. Maintaining the `agentaddr` files for a large ORB domain spread over a large number of subnets is difficult to manage.
- When the name space requires a large number (greater than 100) of well known objects.
- When the number of applications (clients) that require the Smart Agent consistently exceeds 100 in a 10 minute period.

**Note**

In the above situations an alternative directory, such as the Naming Service, may be more appropriate. Refer to “Using the VisiNaming Service” for more information.

## Locating Smart Agents

VisiBroker locates a Smart Agent for use by a client program or object implementation using a 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. For information on how to override the use of broadcast messages, see “Using point-to-point communications”.

## Locating objects through Smart Agent cooperation

When a Smart Agent is started on more than one host 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 processes should terminate unexpectedly, all implementations registered with that Smart Agent discover this event and they will automatically re register with another available Smart Agent.

## Cooperating with the OAD to connect with objects

Object implementations may be registered with the Object Activation Daemon (OAD) so they can be started on demand. Such objects are registered with the Smart Agent as if they are actually active and located within the OAD. When a client requests one of these objects, it is directed to the OAD. The OAD then forwards the client request to the *actual* server. The Smart Agent does not know that the object implementation is not truly active within the OAD. For more information about the OAD, see “Using the Object Activation Daemon (OAD).”

## Starting a Smart Agent (osagent)

At least one instance of the Smart Agent should be running on a host in your local network. Local network refers to a subnetwork in which broadcast messages can be sent.

### Windows

To start the Smart Agent:

– Double-click the osagent executible `osagent.exe` located in:

```
<install_dir\bin\
```

or

– At the Command Prompt, enter: `osagent` [options]. For example:

```
prompt> osagent [options]
```

### UNIX

To start the Smart Agent, enter: `osagent &`. For example:

```
prompt> osagent &
```

### Note

Due to signal handling changes, bourne and korn shell users need to use the `ignoreSignal hup` parameter when starting `osagent` in order to prevent the hangup (`hup`) signal from terminating the process when the user logs out. For example:
`nohup $VBROKERDIR/bin/osagent ignoreSignal hup &`

The `osagent` command accepts the following command line arguments:

| Option | Description |
|---|---|
| `-p` <UDP_port> | 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. |
| `-l` | Turns off logging if `OSAGENT_LOGGING_ON` is set. |
| `-ls` <size> | Specifies trimming log size of 1024KB block. Max value is 300, therefore largest log size is 300MB |
| `+l` <options> | Show/enable logging level. Options supported are: ■ Turn logging on and enable level `"ief" (== +l oief)`, equivalent to `OSAGENT_LOGGING_ON` set. Logs are auto-trim and written to `OSAGENT_LOG_DIR` or `VBROKER_ADM` directory if set. Otherwise default is to `/tmp` on UNIX and `%TEMP%` on Windows. ■ `i` - Informational ■ `e` - Error ■ `w` - Warning ■ `f` - Fatal ■ `d` - Debugging ■ `a` - All |
| `-n, -N` | Disables system tray icon on Windows. |

**Example:**

The following example of the `osagent` command specifies a particular UDP port:

```
osagent -p 17000
```

### Verbose output

#### UNIX

On UNIX, the verbose output is sent to `stdout`.

#### Windows

On Windows, the verbose output is written to a log file stored in either of the following locations:

- `C:\TEMP\vbroker\log\osagent.log`.
- the directory specified by the `VBROKER_ADM` environment variable.

#### Note

To specify a different directory in which to write the log file, use `OSAGENT_LOG_DIR`. To configure logging options you can right-click the Smart Agent icon and select Log Options.

### Disabling the agent

Communication with the Smart Agent can be disabled by passing the VisiBroker ORB the property at runtime:

```
prompt> Server -Dvbroker.agent.enableLocator=false
```

If using string-to-object references, a naming service, or passing in a URL reference, the Smart Agent is not required and can be disabled. If you pass an object name to the `bind()` method, you must use the Smart Agent.

## Ensuring Smart Agent availability

Starting a Smart Agent on more than one host within the local network allows clients to continually bind to objects, even if one Smart Agent 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 is contacted.

If a Smart Agent terminates, any connections between a client and an object implementation established before the Smart Agent terminated will continue without interruption. However, any new `bind()` requests issued by a client causes 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 be sure a Smart Agent is started on one or more host on the local network.

### Checking client existence

A Smart Agent sends an “are you alive” message (often called a *heartbeat* message) to its clients every two minutes to verify the client is still connected. If the client does not respond, the Smart Agent assumes the client has terminated the connection.

You can not change the interval for polling the client.

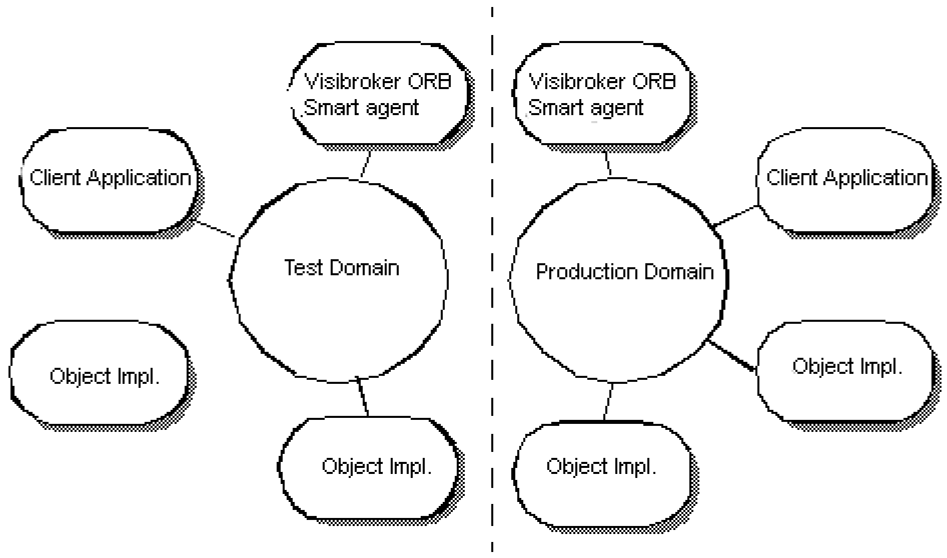
#### Note

The use of the term “client” does not necessarily describe the function of the object or process. Any program that connects to the Smart Agent for object references is a client.

## Working within VisiBroker ORB domains

It is often useful to have two or more VisiBroker ORB domains running at the same time. One domain might consist of production versions of client programs and object implementations, while another domain might consist 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 VisiBroker ORB domain so that their tests do not conflict with one another.

**Figure 13.1** Running separate VisiBroker ORB domains simultaneously



VisiBroker allows you to distinguish between multiple VisiBroker ORB domains on the same network by using unique UDP port numbers for the Smart Agents of 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 VisiBroker ORB domain. For example,

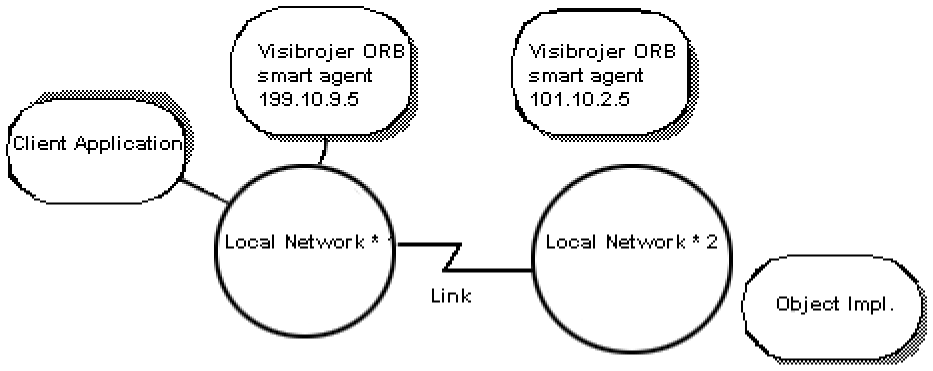
```
prompt> setenv OSAGENT_PORT 5678
prompt> osagent &
prompt> oad &
```

The Smart Agent uses an additional internal port number for both TCP and UDP protocols, the port number is the same for both. This port number is set by using the `OSAGENT_CLIENT_HANDLER_PORT` environment variable.

## 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. The following figure shows two local networks connected by a network link.

**Figure 13.2** Two Smart Agents on separate local networks


Visibrojer ORB smart agent 199.10.9.5
Visibrojer ORB smart agent 101.10.2.5
Client Application
Local Network * 
Local Network * 2
Link
Object Impl.

To allow the Smart Agent on one network to contact a Smart Agent on another local network, use the `OSAGENT_ADDR_FILE` environment variable, as shown in the following example:

```
setenv OSAGENT_ADDR_FILE=<path to agent addr file>
```

Alternatively, use the `vbroker.agent.addrFile` property, as shown in the following example:

```
vbj -Dvbroker.agent.addrFile=<path to agent addr file> ....
```

The following example shows what the `agentaddr` file would contain to allow a Smart Agent on Local Network #1 to connect to a Smart Agent on another local network.

```
101.10.2.6
```

With the appropriate `agentaddr` file, a client program on Network #1 locates and uses object implementations on Network #2. For more information on environment variables, see the Borland VisiBroker *Installation Guide*.

**Note**

If a remote network has multiple Smart Agents running, you should list all the IP addresses 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:

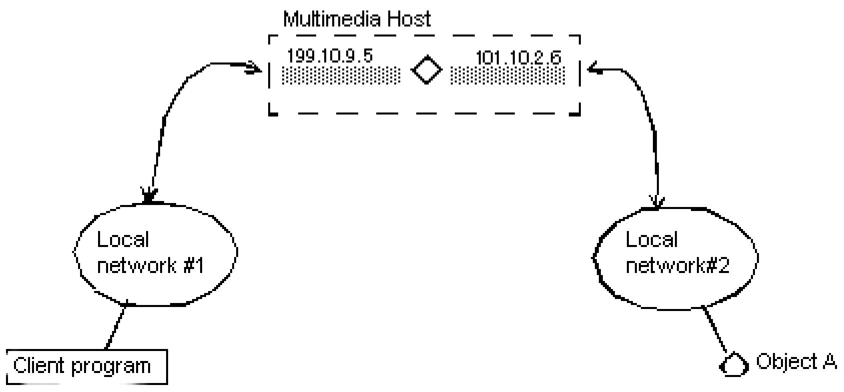
- When Agent 2 starts, it UDP broadcasts its existence and sends a request message to locate any other Smart Agents.
- Agent 1 makes note that Agent 2 is available on the network and responds to the request message.
- Agent 2 makes note that another agent, Agent 1, is available on the network.

  If Agent 2 is terminated gracefully (such as killing with *Ctrl+C*), Agent 1 is notified that Agent 2 is no longer available.

## 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, therefore bridging the local networks.

**Figure 13.3** Smart Agent on a multihomed host



#### UNIX

On a multihomed UNIX host, the Smart Agent dynamically configures itself to listen and broadcast on all of the host's interfaces which support point-to-point connections or broadcast connections. You can explicitly specify interface settings using the `localaddr` file as described in "Specifying interface usage for Smart Agents".

#### Windows

On a multihomed Windows 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 `-v` (verbose) option, each interface that the Smart Agent uses will be listed at the beginning of the messages produced. The example below shows the sample output from a Smart Agent started with the verbose option 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
...
```

The above output shows the address, subnet mask, and broadcast address for each interface in the machine.

#### UNIX

The above output should match the results from the UNIX command `ifconfig -a`.

If want to override these settings, configure the interface information in the `localaddr` file. See "Specifying interface usage for Smart Agents" for details.

## Specifying interface usage for Smart Agents

#### 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 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, and are treated as comments and ignored. The code sample below shows the contents of the `localaddr` file for the multihomed host listed above.

```
#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 on UNIX, you can use the `localaddr` file to explicitly specify the interfaces that your host contains. You can display all available interface values for the UNIX 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 ffffff00 broadcast 199.10.9.255
le1: flags=863<UP,BROADCAST,NOTRAILERS,RUNNING,MULTICAST> mtu 1500
      inet 101.10.2.6 netmask ffffff00 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, the `ipconfig` command will provide the needed values. This command is as follows:

```
prompt> ipconfig
```

Output from this command appears similar to the following:

```
Ethernet adapter El90x1:

      IP Address. . . . . . . . . : 172.20.30.56
      Subnet Mask . . . . . . . . : 255.255.255.0
      Default Gateway . . . . . . : 172.20.0.2
Ethernet adapter Elnk32:
      IP Address. . . . . . . . . : 101.10.2.6
      Subnet Mask . . . . . . . . : 255.255.255.0
      Default Gateway . . . . . . : 101.10.2.1
```

## Using point-to-point communications

VisiBroker provides three different mechanisms for circumventing the use of UDP broadcast messages for locating Smart Agent processes. 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, VisiBroker will revert to using the broadcast message scheme to locate a Smart Agent.

### Specifying a host as a runtime parameter

The code sample below shows how to 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 host located outside your local network. This mechanism takes precedence over any other host specification.

```
prompt> Server -Dvbroker.agent.addr=<ip_address>
```

You can also specify the IP address through the properties file. Look for the `vbroker.agent.addr` entry.

```
vbroker.agent.addr=<ip_address>
```

By default, `vbroker.agent.addr` in the properties file is set to `NULL`.

You can also list the host names where the agent might reside and then point to that file with the `vbroker.agent.addrFile` option in the properties file.

### 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 host is not specified as a runtime parameter.

#### UNIX

```
prompt> setenv OSAGENT_ADDR 199.10.9.5
prompt> client
```

#### Windows

To set the `OSAGENT_ADDR` environment variable on a Windows system, you can use the System control panel and edit the environment variables:

1. Under System Variables, select any current variable.
2. Type `OSAGENT_ADDR` in the Variable edit box.
3. Type the IP address in the Value edit box. For example, `199.10.9.5`.

### Specifying hosts with the agentaddr file

Your client program or object implementation can use the `agentaddr` file to circumvent the use of a UDP broadcast message to locate a Smart Agent. Simply create a file containing the IP addresses or fully qualified hostnames of each host where a Smart Agent is running and then set the `OSAGENT_ADDR_FILE` environment variable to point to the path of the file. When a client program or object implementation has this environment variable set, VisiBroker 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/agentaddr` file is used.

## Ensuring object availability

You can provide fault tolerance for objects by starting instances of those objects on multiple hosts. If an implementation becomes unavailable, the VisiBroker 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. For more information on establishing client policies, see “Using Quality of Service (QoS)”.

#### Note

The Smart Agent implements load balancing using a simple round-robin algorithm on a per agent basis, not on an ORB domain basis. For load balancing between server replicas, when you have more than one Smart Agent in the ORB domain, make sure all servers are registered with the same Smart Agent.

#### Important

The rebind option must be enabled if VisiBroker is to attempt reconnecting the client with an instance 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 VisiBroker ORB object. For information on using QoS, see "Using Quality of Service (QoS)".

When the connection to an object implementation fails and VisiBroker reconnects the client to a replica object implementation, the `bind` method of the bind interceptor will be invoked by VisiBroker. The client must provide an implementation of this bind method to bring the state of the replica up to date. Client interceptors are described in "Client Interceptors".

### Replicating objects registered with the OAD

The OAD ensures greater object availability because if the object goes down, the OAD will restart it. If you want fault tolerance for hosts that may become unavailable, the OAD must be started on multiple hosts and the objects must be registered with each OAD instance.

**Note**

The type of object replication provided by VisiBroker does not provide a multicast or mirroring facility. At any given time there is always a one-to-one correspondence between a client program and a particular object implementation.

## Migrating objects between hosts

Object migration is the process of terminating an object implementation on one host, and then starting it on another host. Object migration can be used to provide load balancing by moving objects from overloaded hosts to hosts that have more resources or processing power (there is no load balancing between servers registered with different Samrt Agents.) Object migration can also be used to keep objects available when a host is 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 host.

### 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 VisiBroker reconnects the client to the object, the interceptor's `rebind_succeeded()` method will be invoked by VisiBroker. The client can implement this method to bring the state of the object up to date.

Refer to "Using Portable Interceptors" for more information about how to use the interceptors.

### Migrating instantiated objects

If the objects that you wish to migrate were created by a server process instantiating the implementation's class, you need only start it on a new host and terminate the server process. When the original instance is terminated, it will be unregistered with the Smart Agent. When the new instance is started on the new host, it will register with the Smart Agent. From that point on, client invocations are routed to the object implementation on the new host.

### Migrating objects registered with the OAD

If VisiBroker objects that you wish to migrate are registered with the OAD, you must first unregister them with the OAD on the old host. Then, reregister them with the OAD on the new host.

Use the following procedure to migrate objects already registered with the OAD:

1. Unregister the object implementation from the OAD on the old host.
2. Register the object implementation with the OAD on the new host.
3. Terminate the object implementation on the old host.

See "Using the Object Activation Daemon (OAD)" for detailed information on registering and unregistering object implementations.

## Reporting all objects and services

The Smart Finder (`osfind`) command reports on all VisiBroker 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 host on which they are executing. The `osfind` command also reports on all VisiBroker objects that are active on the network if these objects are registered with the Smart Agent. 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:

```
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. |
| `-b` | Uses the VisiBroker 2.0 backward compatible osfind mechanism. |
| `-d` | Prints hostnames as quad addresses. |
| `-f` <agent_address_file_name> | Queries Smart Agents running on the hosts specified in the file. This file contains one IP address or fully qualified host name per line. Note that this file is not used when reporting all Smart Agents; it is only used when reporting objects implementations and services. |
| `-g` | Verifies object existence. This can cause considerable delay on loaded systems. Only objects registered `BY_INSTANCE` are verified for existence. Objects that are either registered with the OAD, or those registered `BY_POA` policy are not verified for existence. |
| `-h`, `-help`, `-usage`, `-?` | Prints help information for this option. |
| `-o` | Lists all OADs in your domain. |
| `-p` | Lists all POA instances activated on the same host. Without this option only unique POA names are listed. |

**Windows**

`osfind` is a console application. If you start `osfind` from the Start menu, it runs until completion and exits before you can view the results.

## Binding to Objects

Before your client application invokes a method on an interface it must first obtain an object reference using the `bind()` method.

When your client application invokes the `bind()` method, VisiBroker performs several functions on behalf of your application. These are shown below.

- VisiBroker contacts the `osagent` to locate an object server that is offering the requested interface. If an object name and a host name (or IP address) are specified, they will be used to further qualify the directory service search.
- When an object implementation is located, VisiBroker attempts to establish a connection between the object implementation that was located and your client application.
- If the connection is successfully established, VisiBroker will create a proxy object if necessary, and return a reference to that object.

**Note**

VisiBroker is not a separate process. It is a collection of classes and other resources that allow communication between clients and servers.

14

# Using the Location Service

The VisiBroker Location Service provides enhanced object discovery that enables you to find object instances based on particular attributes. Working with VisiBroker 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 extension to the CORBA specification and is only useful for finding objects implemented with VisiBroker. For more information on the Smart Agent (`osagent`), see "Using the Smart Agent."

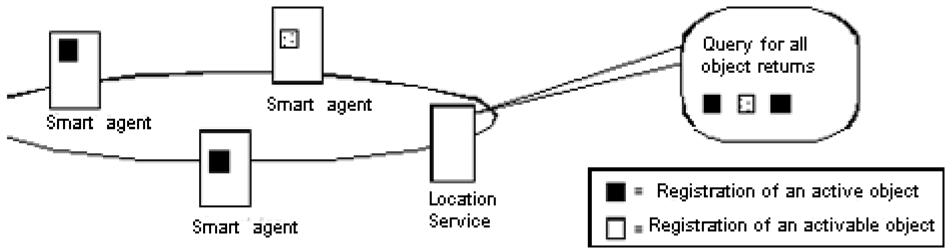
## What is the Location Service?

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. 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 server containing these objects may be started manually or automatically by the OAD. For more information, see "Using POAs", "Using the BOA with VisiBroker", and "Using the Object Activation Daemon (OAD)."

The following diagram illustrates this concept.

**Figure 14.1** Using the Smart Agent to find instances of objects


Smart agent
Smart agent
Smart agent
Location Service
Query for all object returns
■ = Registration of an active object
□ = Registration of an activable object

**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 below.

```
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:

- The object reference, `ref`, is a handle for invoking the object.
- The `iiop_locator` interface provides 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.
- The `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.
- The `instance_name`, which is the name given to the object by its server.
- The `activable` flag, which differentiates between instances that can be activated by an OAD and instances that are started manually.
- The `agent_hostname`, the 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 hosts. You could deploy a bind interceptor that maintains a cache of the host names that offer a replica and each host's recent load average. The interceptor updates its cache by asking the Location Service for the hosts currently offering instances of the object, and then queries the hosts to obtain their load averages. The interceptor then returns an object reference for the replica on the host with the lightest load. For more information about writing interceptors, see "Using Portable Interceptors" and "Using VisiBroker 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).

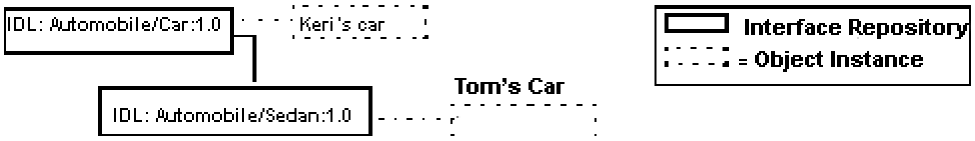
**Note**

The `locserv` executable no longer exists since the service is now part of the core VisiBroker ORB.

The figure below illustrates the use of interface repository IDs and instance names given the following example IDL:

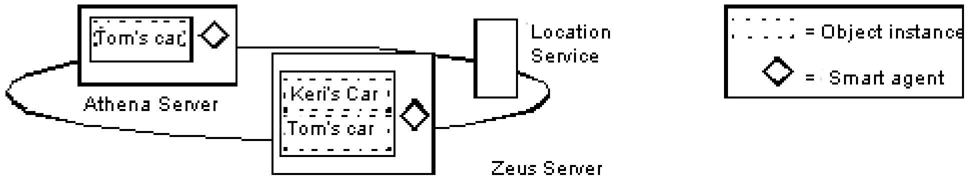
```
module Automobile {
  interface Car{...};
  interface Sedan:Car {...};
}
```

**Figure 14.2** Use of interface repository IDs and instance names



Given the previous example, 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 Keri's Car and two replicas of Tom's Car.

**Figure 14.3** 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 Smart Agents for information. Each of the query methods can raise the `Fail` exception, which provides a reason for the failure.

## Obtaining addresses of all hosts running Smart Agents

Using the `HostnameSeq` method, you can find out which servers are hosting VisiBroker Smart Agents. In the example shown in the figure below, this method would return the addresses (such as, IP address string) of two servers: Athena and Zeus.

## Finding all accessible interfaces

You can query the VisiBroker Smart Agents on a network to find out about all accessible interfaces. To do so, you can use the `RepositoryIDSeq` method. In the example shown in the following figure, this method would return the repository IDs of two interfaces: Car and Sedan.

**Note**

Earlier versions of the VisiBroker 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 the figure above, 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 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:

| Method | Description |
|---|---|
| `CORBA::ObjectSeq* all_instances(const char* _repository_id)` | Use this method to return object references to instances of the interface. |
| `DescSeq* all_instances_descs(const char* _repository_id)` | Use this method to return an instance description for instances of the interface. |

In the example shown in the figure above, 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 Keri'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 Smart Agents on a network to return all occurrences of a particular instance name.

| Method | Description |
|---|---|
| `CORBA::ObjectSeq* all_replica(const char* _repository_id, const char*_instance_name)` | Use this method to return object references to like-named instances of the interface. |
| `DescSeq all_replica_descs(const char*_repository_id, const char* _instance_name)` | Use this method to return an instance description for like-named instances of the interface. |

In the example shown in the previous figure, 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 tables:

| Methods | Description |
|---|---|
| `void reg_trigger(const TriggerDesc& _desc,TriggerHandler_ptr _handler)` | Use this method to register a trigger handler. |
| `void unreg_trigger(const TriggerDesc& _desc,TriggerHandler_ptr _handler)` | 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 tables:

| Method | Description |
|---|---|
| `void impl_is_ready(const Desc& _desc)` | This method is called by the Location Service when an instance matching the `desc` becomes accessible. |
| `void impl_is_down(const Desc& _desc)` | 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 with 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.

```
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 the figure above, 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 Keri'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 Tom's Car on the Athena server 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:

```
// 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`.

Finding all instances satisfying the AccountManager interface:

```
#include "corba.h"
#include "locate_c.hh"

// USE_STD_NS is a define setup by VisiBroker to use the std namespace
USE_STD_NS
int main(int argc, char** argv) {
  try {
    // ORB initialization
    CORBA::ORB_var the_orb = CORBA::ORB_init(argc, argv);

    // Obtain a reference to the Location Service
    CORBA::Object_var obj = the_orb->
      resolve_initial_references("LocationService");
    if ( CORBA::is_nil(obj) ) {
      cout << "Unable to locate initial LocationService" << endl;
      return 0;
    }
    ObjLocation::Agent_var the_agent = ObjLocation::Agent::_narrow(obj);

    // Query the Location Service for all implementations of
    // the Account interface
    ObjLocation::ObjSeq_var accountRefs =
      the_agent->all_instances("IDL:Bank/AccountManager:1.0");
    cout << "Obtained " << accountRefs->length()
      << " Account objects" << endl;
    for (CORBA::ULong i=0; i < accountRefs->length(); i++) {
      cout << "Stringified IOR for account #" << i <<
        ":" << endl;
      CORBA::String_var stringified_ior(the_orb
        ->object_to_string(accountRefs[i]));
      cout << stringified_ior << endl;
      cout << endl;
    }
  }
  catch (const CORBA::Exception& e) {
    cout << "Caught exception: " << e << endl;
    return 0;
  }
  return 1;
}
```

## Finding interfaces and instances 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.

Finding everything known to a Smart Agent:

```
#include "corba.h"
#include "locate_c.hh"

// USE_STD_NS is a define setup by VisiBroker to use the std namespace
// if it exists
USE_STD_NS
```

```
int DisplaybyRepID(CORBA::ORB_ptr the_orb,
  ObjLocation::Agent_var the_agent,
  char * myRepId) {

  ObjLocation::ObjSeq_var accountRefs;
  accountRefs = the_agent->all_instances(myRepId);
  cout << "Obtained " << accountRefs->length()
    << " Account objects" << endl;
  for (CORBA::ULong i=0; i < accountRefs->length(); i++) {
    cout << "Stringified IOR for account #" << i << ":"
      << endl;
    CORBA::String_var stringified_ior(
      the_orb->object_to_string(accountRefs[i]));
    cout << stringified_ior << endl;
    cout << endl;
  }
  return(1);
}
void PrintUsage(char * name) {
  cout << "\nUsage: \n" << endl;
  cout << "\t" << name << " [Rep ID]" << endl;
  cout << "\n\tWith no argument, finds and prints all objects" << endl;
  cout << "\tOptional rep ID searches for specific rep ID\n" << endl;
}
int main(int argc, char** argv) {
  char myRepId[255] = "";
  if (argc == 2) {
    if (!strcmp(argv[1], "-h") || !strcmp(argv[1], "/?") ||
      !strcmp(argv[1], "-?") ) {
      PrintUsage(argv[0]);
      exit(0);
    } else {
      strcpy(myRepId, argv[1]);
    }
  }


  else if (argc > 2) {
    PrintUsage(argv[0]);
    exit(0);
  }
  try {
    CORBA::ORB_ptr the_orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_ptr obj = the_orb->
    resolve_initial_references("LocationService");
    if ( CORBA::is_nil(obj) ) {
      cout << "Unable to locate initial LocationService" << endl;
      return 0;
  }
  ObjLocation::Agent_var the_agent = ObjLocation::Agent::_narrow(obj);
  ObjLocation::DescSeq_var descriptors;
  //Display stringified IOR for RepID requested and exit
  if (argc == 2) {
    DisplaybyRepID(the_orb, the_agent, myRepId);
    exit(0);
  }
  //Report all hosts running osagents
  ObjLocation::HostnameSeq_var HostsRunningAgents =
    the_agent->all_agent_locations();
  cout << "Located " << HostsRunningAgents->length()
```

```
      << " Hosts running Agents" << endl;
    for (CORBA::ULong k=0; k<HostsRunningAgents->length(); k++) {
      cout << "\tHost #" << (k+1) << ": "
        << (const char*) HostsRunningAgents[k] << endl;
    }
    cout << endl;
    // Find  and display all Repository Ids
    ObjLocation::RepositoryIdSeq_var repIds = the_agent->all_repository_ids();
    cout << "Located  " << repIds->length() <<
      " Repository Ids" << endl;
    for (CORBA::ULong j=0; j<repIds->length(); j++) {
      cout << "\tRepository ID #" << (j+1) << ": "
        << repIds[j] << endl;
    }
    // Find all Object Descriptors for each Repository Id
    for (CORBA::ULong i=0; i < repIds->length(); i++) {
      descriptors = the_agent->all_instances_descs(repIds[i]);
      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;
      }
    }
    } catch (const CORBA::Exception& e) {
        cout << "CORBA Exception during execution of find_all: " << e << endl;
        return 0;
    }
    return 1;
  }
```

## Writing 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::ORB::shutdown()` method which directs the BOA to exit the main program's `impl_is_ready()` method so the program can terminate.

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.

Implementing a trigger handler:

```
// AccountTrigger.c
#include "locate_s.hh"

// USE_STD_NS is a define set up by VisiBroker to use the std namespace
   USE_STD_NS
// 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_var 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:
      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 (yes/no)? " << endl;
         char prompt[256];
         cin >> prompt;
         if ((prompt[0] == 'y') || (prompt[0] == 'Y')) {
            try {
               _agent->unreg_trigger(_initial_desc, this);
            }
            catch (const ObjLocation::Fail& e) {
               cout << "Failed to unregister trigger with reason=["
```

```
                            << (int) e.reason << "]" << endl;
                  }
                  cout << "exiting..." << endl;
                  CORBA::ORB::shutdown();
               }
            }

         private:
            ObjLocation::Agent_var _agent;
            ObjLocation::TriggerDesc _initial_desc;
      };

      int main(int argc, char* const * argv)
      {
         try {
            CORBA::ORB_var the_orb = CORBA::ORB_init(argc, argv);
            CORBA::BOA_var boa = the_orb->BOA_init(argc, argv);
            CORBA::Object_var obj = the_orb->
            resolve_initial_references("LocationService");
            if ( CORBA::is_nil(obj) ) {
               cout << "Unable to locate initial LocationService" << endl;
               return 0;
            }
            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;
            desc.repository_id = (const char*) "IDL:Bank/AccountManager:1.0";
            desc.instance_name = (const char*) "";
            desc.host_name = (const char*) "";

            ObjLocation::TriggerHandler_var trig = new TriggerHandlerImpl(the_agent,
               desc);
            boa->obj_is_ready(trig);
            the_agent->reg_trigger(desc,trig);
            boa->impl_is_ready();
         }
         catch (const CORBA::Exception& e) {
            cout << "account_trigger caught Exception: " << e << endl;
            return 0;
         }
         return 1;
      }
```

15

# Using the VisiNaming Service

This section describes the usage of the VisiBroker VisiNaming Service which is a complete implementation of the CORBA Naming Service Specification Version 1.2 (formal/02–09–02).

## Overview

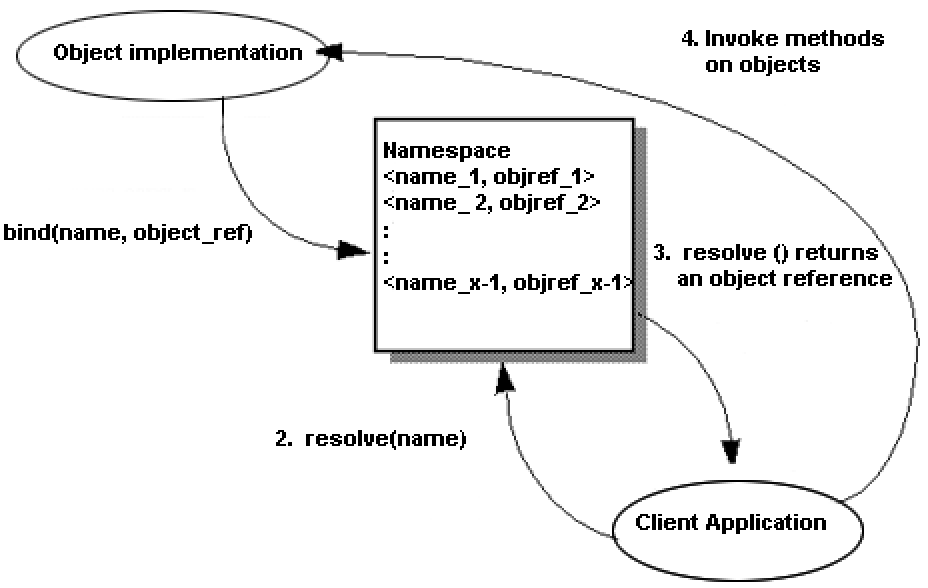
The VisiNaming Service allows you to associate one or more *logical* names with an object reference and store those names in a *namespace*. With the VisiNaming Service, your client applications can obtain an object reference by using the logical name assigned to that object.

The figure below contains a simplified view of the VisiNaming 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 15.1** 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 VisiNaming Service as opposed to the Smart Agent.

– Smart Agent uses a flat namespace, while the VisiNaming Service uses a hierarchical one.

– If you use the Smart Agent, an object's interface name is defined at the time you compile your client and server applications. This means that if you change an interface name, you must recompile your applications. In contrast, the VisiNaming service allows object implementations to bind logical names to its objects at runtime.

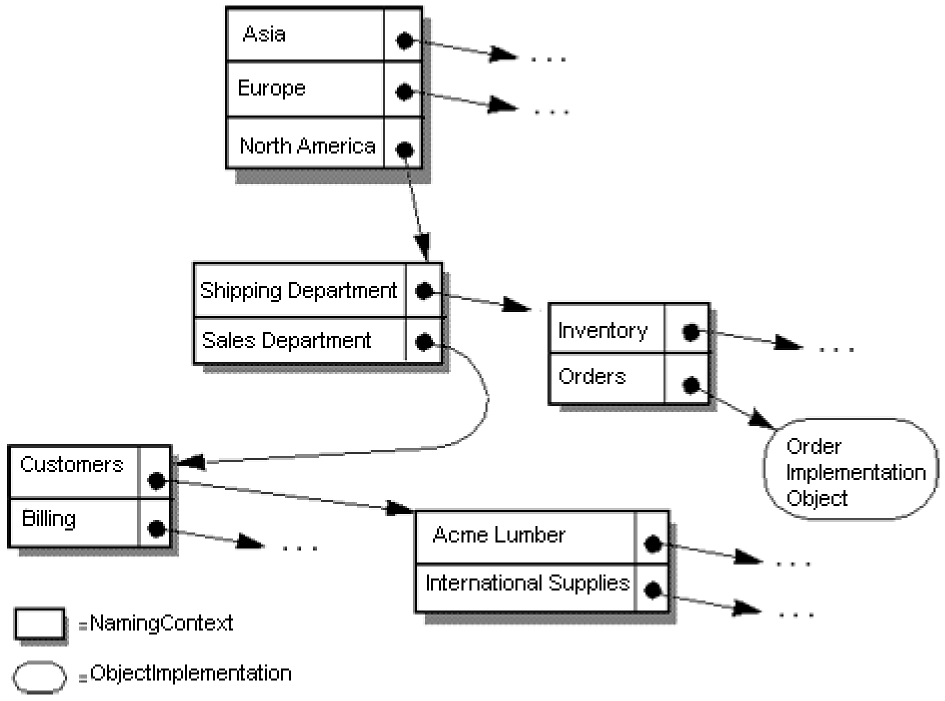
– If you use the Smart Agent, an object may implement only one interface name. The VisiNaming service allows you to bind more than one logical name to a single object.

For more information about the Smart Agent (osagent), see “Using the Smart Agent.”

## Understanding the namespace

The figure below shows how the VisiNaming 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 VisiNaming 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 15.2** Naming scheme for an order entry system



## Naming contexts

To implement the namespace shown above with the VisiNaming Service, each of the shadowed boxes in the diagram above, 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.

## Naming context factories

A naming context factory provides the interface for *bootstrapping* the VisiNaming Service. It has operations for shutting down the VisiNaming Service and creating new contexts when there are none. Factories also have an additional API that returns the root context. The root context provides a very critical role as a reference point. This is the common starting point to store all data that are supposed to be publicly available.

Two classes are provided with the VisiNaming Service that allow you to create a namespace; the default naming context factory and the extended naming context factory. The default naming context factory creates an empty namespace that has no root `NamingContext`. You may find it more convenient to use the extended naming context factory because it creates a namespace with a root `NamingContext`.

You must obtain at least one of these `NamingContext` objects before your object implementations can bind names to their objects and before client applications can resolve a name to an object reference.

Each of the `NamingContext` objects shown in the figure above could be implemented within a single *name service* process, or they could be implemented within as many as five distinct name server processes.

## 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.”

```
module CosNaming
   typedef string Istring;
   struct NameComponent {
      Istring id;
      Istring kind;
   };
   typedef sequence<NameComponent> Name;
};
```

The `id` and `kind` attributes of `NameComponent` in the code example above, 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 `NameComponent` can exceed 255 characters. Furthermore, the VisiNaming Service does not support `NameComponent` which uses 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).

```
"Borland.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 `NorthAmerica/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`.

The code sample below 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.”

```
...
// 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 VisiNaming Service

The VisiNaming Service can be started with the following commands. Once you have started the Naming service, you may browse its contents by using the VisiBroker Console.

### Installing the VisiNaming Service

The VisiNaming Service is installed automatically when you install VisiBroker. It consists of a file `nameserv`, which for Windows is a binary executable and for UNIX is a script, and Java class files which are stored in the `vbjorb.jar` file.

### Configuring the VisiNaming Service

In previous versions of VisiBroker, the VisiNaming Service maintained persistence by logging any modifying operations to a flat-file. From version 4.0 onward, the VisiNaming Service works in conjunction with backing store adapters. It is important to note that not all backing store adapters support persistence. The default `InMemory` adapter is non-persistent while all the other adapters are. For more details about adapters, see “Pluggable backing store”.

**Note**

A Naming Server is designed to register itself with the Smart Agent. In most cases you should to run the Smart Agent to bootstrap the VisiNaming Service. This allows clients to retrieve the initial root context by calling the `resolve_initial_references` method. The resolving function works through the Smart Agent for the retrieval of the required references. Similarly, Naming Servers that participate in a federation also uses the same mechanism for setting up a federation.

For more information about the Smart Agent, see “Using the Smart Agent.”

### Starting the VisiNaming Service

You can start the VisiNaming Service by using the `nameserv` launcher program in the `/bin` directory. The `nameserv` launcher uses the `com.inprise.vbroker.naming.ExtFactory` factory class by default.

**UNIX**

```
nameserv [driver_options] [nameserv_options] <ns_name> &
```

**Windows**

```
start nameserv [driver_options] [nameserv_options] <ns_name>
```

See "General options" for descriptions of the driver options available to all of the VisiBroker programmer tools.

| nameserv_option | Description |
|---|---|
| `-?`, `-h`, `-help`, `-usage` | Print out the usage information. |
| `-config <properties_file>` | Use <properties_file> as the configuration file when starting up the VisiNaming Service. |
| `<ns_name>` | The name to use for this VisiNaming Service. This is optional; the default name is `NameService`. |

In order to force the VisiNaming Service to start on a particular port, the VisiNaming Service must be started with the following command line option:

```
prompt> nameserv -J-Dvbroker.se.iiop_tp.scm.iiop_tp.listener.port=<port number>
```

The default name for VisiNaming is "NameService", if you want to specify a name other than this, you can start VisiNaming in the following way:

```
prompt> nameserv -J-Dvbroker.se.iiop_tp.scm.iiop_tp.listener.port=<port number>
<ns_name>
```

## Invoking the VisiNaming Service from the command line

The VisiNaming Service Utility (`nsutil`) provides the ability to store and retrieve bindings from the command line.

### Configuring nsutil

To use `nsutil`, first configure the Naming service instance using the following commands:

```
prompt>nameserv <ns_name>

prompt>nsutil -VBJprop <option> <cmd> [args]
```

| Option | Description |
|---|---|
| `ns_name` | Configure the Naming service to contact |
| `SVCnameroot=<ns_name>` | **Note:** Before using `SVCnameroot`, you must first run OSAgent. |
| `ORBInitRef=NameService=<url>` | File name or URL, prefixed by its type, which may be (`corbaloc:`, `corbaname:`, `file:`, `ftp:`, `http:`, or `ior:`). For example, to assign a file in a local directory, the `ns_config` string would be:`-VBJprop ORBInitRef=NameService=<file:ns.ior>` |
| `cmd` | Any CosNaming operation, and, in addition, ping and shutdown. |

## Running nsutil

The VisiNaming Service Utility supports all the CosNaming operations as well as three additional commands. The CosNaming operations supported are:

| cmd | Parameter(s) |
|---|---|
| `bind` | name objRef |
| `bind_context` | name ctxRef |
| `bind_new_context` | name |
| `destroy` | name |
| `list` | [name1 name2 name3...] |
| `new_context` | No parameter |
| `rebind` | name objRef |
| `rebind_context` | name ctxRef |
| `resolve` | name |
| `unbind` | name |

**Note**

For the operations `destroy` and `list`, the `name` parameter must refer to existing naming contexts. For the operation `list` only, there can be zero or more naming contexts, whose contents will be listed. In the case where no naming context is specified, the content of the root naming context will be listed.

The additional `nsutil` commands are:

| cmd | Parameter | Description |
|---|---|---|
| `ping` | name | Resolves the stringified `name` and contacts the object to see if it is still alive. |
| `shutdown` | <naming context factory name or stringified ior> | Shuts the VisiNaming Service down gracefully from the command line. The mandatory parameter of this operation specifies either the naming context factory's name as registered with the osagent or the stringified IOR of the factory. |
| `unbind_from_cluster` | name objRef | Unbinds a specific object in an implicit cluster. The `name` is the object's logical name and the `objRef` is the stringified object reference that is to be unbound. |

To run an operation from the `nsutil` command, place the operation name and its parameters as the `<cmd>` parameter. For example:

```
prompt>nsutil -VBJprop ORBInitRef=NameService=file://ns.ior resolve myName
```

## Shutting down the VisiNaming Service using nsutil

To shut down the VisiNaming Service using `nsutil`, use the `shutdown` command:

```
prompt>nsutil -VBJprop ORBInitRef=NameService=file://ns.ior shutdown <ns_name>
```

## Bootstrapping the VisiNaming Service

There are three ways to start a client application to obtain an initial object reference to a specified VisiNaming Service. You can use the following command-line options when starting the VisiNaming Service:

- `ORBInitRef`
- `ORBDefaultInitRef`
- `SVCnameroot`

The following example illustrates how to use these options.

Suppose there are three VisiNaming Services running on the host `TestHost`:

```
ns1, ns2, and ns3
```

running on the ports 20001, 20002 and 20003 respectively.

And there are three server applications:

```
sr1, sr2, sr3.
```

Server `sr1` binds itself in `ns1`, Server `sr2` binds itself in `ns2`, and server `sr3` in `ns3`.

### Calling resolve_initial_references

The VisiNaming 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.

```
...
CORBA::ORB_ptr orb = CORBA::ORB_init(argv, argc, NULL);
CORBA::Object_var rootObj = orb->resolve_initial_references("NameService");
...
```

### Using -DSVCnameroot

You use the `-DSVCnameroot` option to specify into which VisiNaming Service instance (especially important if several unrelated Naming service instances are running) you want to bootstrap.

For instance, if you want to bootstrap into `ns1`, you would start your client program as:

```
<client_application> -DSVCnameroot=ns1
```

You can then obtain the root context of `ns1` by calling the `resolve_initial_references` method on an ORB reference inside your client application as illustrated below. The Smart Agent must be running in order to use this option.

Keep in mind that the `-DSVCnameroot` bootstrapping mechanism is based on the proprietary functionality that VisiBroker Smart Agent provides and it is not interoperable with other CORBA implementations.

### Using -ORBInitRef

You can use either the `corbaloc` or `corbaname` URL naming schemes to specify which VisiNaming Service you want to bootstrap. This method does not rely on the Smart Agent.

#### Using a corbaloc URL

If you want to bootstrap using VisiNaming Service `ns2`, then start your client application as follows:

```
<client_application> -ORBInitRef=NameService=corbaloc://TestHost:20002/
NameService
```

You can then obtain the root context of ns2 by calling the resolve_initial_references method on the VisiBroker ORB reference inside your client application as illustrated in the example above.

**Note**

The deprecated iioploc and iiopname URL schemes are implemented by corbaloc and corbaname, respectively. For backwards compatibility, the old schemes are still supported.

### Using a corbaname URL

If you want to bootstrap into ns3 by using corbaname, then you should start your client program as:

```
<client_application> -ORBInitRef NameService=corbaname://TestHost:20003/
```

You can then obtain the root context of ns3 by calling the resolve_initial_references method on the VisiBroker ORB reference inside your client application as illustrated above.

## -ORBDefaultInitRef

You can use either a corbaloc or corbaname URL to specify which VisiNaming Service you want to bootstrap. This method does not rely on the Smart Agent.

### Using -ORBDefaultInitRef with a corbaloc URL

If you want to bootstrap into ns2, then you should start your client program as:

```
<client_application> -ORBDefaultInitRef corbaloc://TestHost:20002
```

You can then obtain the root context of ns2 by calling the resolve_initial_references method on the VisiBroker ORB reference inside your client application as illustrated in the sample above.

### Using -ORBDefaultInitRef with corbaname

The combination of -ORBDefaultInitRef or -DORBDefaultInitRef and corbaname works differently from what is expected. If -ORBDefaultInitRef or -DORBDefaultInitRef is specified, a slash and the stringified object key is always appended to the corbaname.

For example, if the URL is corbaname::TestHost:20002, then by specifying -ORBDefaultInitRef, resolve_initial_references in C++ will result in a new URL: corbaname::TestHost:20003/NameService.

## NamingContext

This object is used to contain and manipulate a list of names that are bound to VisiBroker 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 sample below shows the IDL specification for the NamingContext.

```
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.

```
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);
   };
};
```

## 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 can be contained by another context.

### Obtaining the default context

The VisiBroker 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 `SVCnameroot` or `ORBInitRef` command-line argument when the client application was started. The sample below shows how a C++ client application could invoke this method.

```
#include "CosNaming_c.hh"
...
int main(int argc, char* const* argv) {
   try {
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
      ...
      CORBA::Object_var ref = orb->resolve_initial_references("NameService");
```

```
        CosNaming::NamingContext_var rootContext =
              CosNaming::NamingContext::_narrow(ref);
        ...
     } catch(const CORBA::Exception& e) {
        cout << "Failure: " << e << endl;
        exit(1);
     }
     exit(0);
  }
```

## Obtaining naming context factories

If there is no osagent running on the network, a naming service client can get a reference to the naming context factory by resolving the initial reference of the factory as follows:

```
...
CORBA::Object_var factRef = orb-
>resolve_initial_references("VisiNamingContextFactory");
CosNamingExt::NamingContextFactory_var factory =
        CosNamingExt::NamingContextFactory::_narrow(factRef);
...
```

Start this client as shown in the following example:

```
Client -ORBInitRef = VisiNamingContextFactory =
corbaloc::<host>:<port>/VisiNamingContextFactory
```

## VisiNaming Service properties

The following tables list the VisiNaming Service properties:

| Property | Default | Description |
|---|---|---|
| vbroker.naming.adminPwd | inprise | Password required by administrative VisiBroker Naming service operations. |
| vbroker.naming.enableSlave | 0 | If 1, enables master/slave naming services configuration. See “VisiNaming Service Clusters for Failover and Load Balancing” for information about configuring master/slave naming services. |
| vbroker.naming.iorFile | ns.ior | This property specifies the full path name for storing the Naming service IOR. If you do not set this property, the Naming service will try to output its IOR into a file named ns.ior in the current directory. The Naming service silently ignores file access permission exceptions when it tries to output its IOR. |

| Property | Default | Description |
|---|---|---|
| `vbroker.naming.logLevel` | `emerg` | This property specifies the level of log messages to be output from Naming service. Acceptable values are:<br>■ vbroker.log.enable=true<br>■ vbroker.log.filter.default.enable=false<br>■ vbroker.log.filter.default.register=naming<br>■ vbroker.log.filter.default.naming.enable=true<br>■ vbroker.log.filter.default.naming.logLevel=debug |
| `vbroker.naming.logUpdate` | `false` | This property allows special logging for all of the update operations on the `CosNaming::NamingContext`, `CosNamingExt::Cluster`, and `CosNamingExt::ClusterManager` interfaces.<br>The `CosNaming::NamingContext` interface operations for which this property is effective are:<br>`bind`, `bind_context`, `bind_new_context`, `destroy`, `rebind`, `rebind_context`, `unbind`.<br>The `CosNamingExt::Cluster` interface operations for which this property is effective are:<br>`bind`, `rebind`, `unbind`, `destroy`.<br>The `CosNamingExt::ClusterManager` interface operation for which this property is effective is:<br>`create_cluster`<br>When this property value is set to `true` and any of the above methods is invoked, the following log message is printed (the output shows a bind operation being executed):<br>`00000007,5/26/04 10:11 AM,127.0.0.1,00000000,`<br>`VBJ-Application,VBJ ThreadPool Worker,INFO,`<br><br>`OPERATION NAME : bind`<br>`CLIENT END POINT : Connection[socket=Socket`<br>`[addr=/127.0.0.1, port=2026, localport=1993]]`<br>`PARAMETER 0 : [(Tom.LoanAccount)]`<br>`PARAMETER 1 : Stub[repository_id=IDL:Bank/`<br>`LoanAccount:1.0, key=TransientId[poaName=/,`<br>`id={4 bytes: (0)(0)(0)(0)},sec=505,usec=990917734,`<br>`key_string=%00VB%01%00%00%00%02/%00%20%20%00%00%00%`<br>`04%00%00%00%00%00%00%01%f9;%104f],codebase=null]` |

For more information see “Object Clusters”.

| Property | Default | Description |
|---|---|---|
| `vbroker.naming.enableClusterFailover` | `true` | When set to `true`, it specifies that an interceptor be installed to handle fail-over for objects that were retrieved from the VisiNaming Service. In case of an object failure, an attempt is made to transparently reconnect to another object from the same cluster as the original. |

| Property | Default | Description |
|---|---|---|
| `vbroker.naming.propBindOn` | `0` | If `1`, the implicit clustering feature is turned on. |
| `vbroker.naming.smrr.pruneStaleRef` | `1` | This property is relevant when the name service cluster uses the Smart Round Robin criterion. When this property is set to `1`, a stale object reference that was previously bound to a cluster with the Smart Round Robin criterion will be removed from the bindings when the name service discovers it. If this property is set to `0`, stale object reference bindings under the cluster are not eliminated. However, a cluster with Smart Round Robin criterion will always return an active object reference upon a `resolve()` or `select()` call if such an object binding exists, regardless of the value of the `vbroker.naming.smrr.pruneStaleRef` property. By default, the implicit clustering in the name service uses the Smart Round Robin criterion with the property value set to `1`. If set to `2`, this property disables the clearing of stale references completely, and the responsibility of cleaning up the bindings belongs to the application, rather than to VisiNaming. |

For more information see the VisiNaming Service Clusters for Failover and Load Balancing section.

| Property | Default | Description |
|---|---|---|
| `vbroker.naming.enableSlave` | `0` | See "VisiNaming Service properties". |
| `vbroker.naming.slaveMode` | No default.<br>Can be set to `cluster` or `slave`. | This property is used to configure VisiNaming Service instances in the cluster mode or in the master/slave mode. The `vbroker.naming.enableSlave` property must be set to `1` for this property to take effect.<br>Set this property to `cluster` to configure VisiNaming Service instances in the cluster mode. VisiNaming Service clients will then be load balanced among the VisiNaming Service instances that comprise the cluster. Client failover across these instances are enabled.<br>Set this property to `slave` to configure VisiNaming Service instances in the master/slave mode. VisiNaming Service clients will always be bound to the master server if the master is running but failover to the slave server when the master server is down. |
| `vbroker.naming.serverClusterName` | `null` | This property specifies the name of a VisiNaming Service cluster. Multiple VisiNaming Service instances belong to a particular cluster (for example, `clusterXYZ`) when they are configured with the cluster name using this property. |

| Property | Default | Description |
|---|---|---|
| `vbroker.naming.serverNames` | `null` | This property specifies the factory names of the VisiNaming Service instances that belong to a cluster. Each VisiNaming Service instance within the cluster should be configured using this property to be aware of all the instances that constitute the cluster. Each name in the list must be unique. This property supports the format: `vbroker.naming.serverNames= Server1:Server2:Server3` See the related property, `vbroker.naming.serverAddresses`. |
| `vbroker.naming.serverAddresses` | `null` | This property specifies the host and listening port for the VisiNaming Service instances that comprise a VisiNaming Service cluster. The order of VisiNaming Service instances in this list must be identical to that of the related property `vbroker.naming.serverNames`, which specifies the names of the VisiNaming Service instances that comprise a VisiNaming Service Cluster. This property supports the format: `vbroker.naming.serverAddresses= host1:port1;host2:port2;host3:port3` |
| `vbroker.naming.anyServiceOrder` `(To be set on VisiNaming Service clients)` | `false` | This property must be set to `true` on the VisiNaming Service client to utilize the load balancing and failover features available when VisiNaming Service instances are configured in the VisiNaming Service cluster mode. The following is an example of how to use this property: `client -DVbroker.naming. anyServiceOrder=true` |

## Pluggable backing store

The VisiNaming Service maintains its namespace by using a pluggable backing store. Whether or not the namespace is persistent, depends on how you configure the backing store: to use JDBC adapter, the Java Naming and Directory Interface (JNDI, which is certified for LDAP), or the default, in-memory adapter.

### Types of backing stores

The types of backing store adapters supported are:

- In-memory adapter
- JDBC adapter for relational databases
- DataExpress adapter
- JNDI (for LDAP only)

**Note**

For an example using pluggable adapters, see the code located in the directory:

```
<install dir>/vbroker/examples/ins/pluggable_adaptors
```

### In-memory adapter

The in-memory adapter keeps the namespace information in memory and is not persistent. This is the adapter used by the VisiNaming Service by default.

### JDBC adapter

JDBC adapter Relational databases are supported via JDBC. The following databases have been certified to work with the VisiNaming Service JDBC adapter:

- DataStore 7
- Oracle 10G, Release 2
- Microsoft SQLServer 2005
- DB 2 8.2
- InterBase 7

Multiple VisiNaming Service instances can use the same back-end relational database if one of these is true:

- The VisiNaming Service instances are independent of each other and use different factory names, or,
- The VisiNaming Service instances are all part of the same VisiNaming Service Cluster.

### DataExpress adapter

In addition to the JDBC adapter, there is also a DataExpress adapter which allows you to access JDataStore databases natively. It is much faster than accessing JDataStore through JDBC, but the DataExpress adapter has some limitations. It only supports a local database running on the same machine as the Naming Server. To access a remote JDataStore database, you must use the JDBC adapter.

### JNDI adapter

A JNDI adapter is also supported. Sun's JNDI (Java Naming and Directory Interface) provides a standard interface to multiple naming and directory services throughout the enterprise. JNDI has a Service Provider Interface (SPI) with which different naming and service vendors must conform. There are different SPI modules available for Netscape LDAP server, Novell NDS, WebLogic Tengah, etc. By supporting JNDI, the VisiNaming Service allows you to have portable access to these naming and directory services and other future SPI providers.

The VisiNaming JNDI adapter is certified with the following LDAP implementations:

- iPlanet Directory Server 5.0
- OpenLdap 2.3.35

You must use Sun and Netscape JNDI Driver version 1.2 to leverage LDAP.

## Configuration and use

Backing store adapters are pluggable, which means that the type of adapter used can be specified by user-defined information stored in a configuration (properties) file used when starting up the VisiNaming Service. All adapters, except the in-memory one, provide persistence. The in-memory adapter should be used when you want to use a lightweight VisiNaming Service which keeps its namespace entirely in memory.

**Note**

For the current version of the VisiNaming Service, you cannot change settings while the VisiNaming Service is running. To change a setting, you must bring down the service, make the change to the configuration file, and then restart the VisiNaming Service.

## Properties file

As with the VisiNaming Service in general, which adapter is to be used and any specific configuration of it is handled in VisiNaming Service properties file. The default properties common to all adapters are:

| Property | Default | Description |
|---|---|---|
| `vbroker.naming.backingStoreType` | `InMemory` | Specifies the Naming service adapter type to use. This property specifies which type of backing store you want the VisiNaming Service to use. The valid options are: `InMemory`, `JDBC`, `Dx`, `JNDI`. The default is `InMemory`. |
| `vbroker.naming.cacheOn` | `0` | Specifies whether to use the Naming Service cache. A value of `1` (one) enables caching. |
| `vbroker.naming.cache.connectString` | | This property is required when the Naming Service cache is enabled (`vbroker.naming.cacheOn=1`) and the Naming Service instances are configured in Cluster or Master/Slave mode. It helps locate an Event Service/VisiNotify instance in the format `<hostname>:<port>`. For example: `vbroker.naming.cache.connectString=127.0.0.1:14500` See “Caching facility” for details about enabling the caching facility and setting the appropriate properties. |
| `vbroker.naming.cache.size` | `2000` | This property specifies the size of the Naming Service cache. Higher values will mean caching of more data at the cost of increased memory consumption. |
| `vbroker.naming.cache.timeout` | `0` (no limit) | This property specifies the time, in seconds, since the last time a piece of data was accessed, after which the data in the cache will be purged in order to free memory. The cached entries are deleted in LRU (Least Recently Used) order. |

## JDBC Adapter properties

The following sections describe the JDBC Adapter properties.

```
vbroker.naming.backingStoreType
```

This property should be set to `JDBC`. The `poolSize` , `jdbcDriver`, `url`, `loginName`, and `loginPwd` properties must also be set for the JDBC adapter.

```
vbroker.naming.jdbcDriver
```

This property specifies the JDBC driver that is needed to access the database used as your backing store. The VisiNaming Service loads the appropriate JDBC driver specified. The default is the Java DataStore JDBC driver.

| JDBC driver class name | Description |
|---|---|
| `com.borland.datastore.jdbc.DataStoreDriver` | JDataStore JDBC Driver 7.0 |
| `com.sybase.jdbc2.jdbc.SybDriver` | Sybase driver (jConnect Version 5.0) |
| `oracle.jdbc.driver.OracleDriver` | Oracle driver (using classes12.zip Version 8.1.7.0.0) |
| `interbase.interclient.Driver` | Interbase driver (using InterClient.jar Version 3.0.12) |

| JDBC driver class name | Description |
|---|---|
| `weblogic.jdbc.mssqlserver4.Driver` | WebLogic MS SQLServer 2005 JDBC driver (Version 1.1) |
| `com.ibm.db2.jcc.DB2Driver` | IBM DB2 driver (using db2jcc.jar Version 1.2.117) |

`vbroker.naming.minReconInterval`

This property sets the database reconnection retry time by the Naming Service in seconds. The default value is `30`. The Naming Service will ignore the request and throw a `CannotProceed` exception if the time interval between this request and the last reconnection time is less than the value set by this property. The valid value for this property is 0 (zero) or a greater integer. If the property value is `0` (zero), the VisiNaming Service will try to reconnect to the database for every request, once disconnected.

`vbroker.naming.loginName`

This property is the login name associated with the database. The default is `VisiNaming`.

`vbroker.naming.loginPwd`

This property is the login password associated with the database. The default value is `VisiNaming`.

`vbroker.naming.poolSize`

This property specifies the number of database connections in your connection pool when using the JDBC Adapter as our backing store. The default value is `5`, but it can be increased to whatever value the database can handle. If you expect many requests will be made to the VisiNaming Service, you should make this value larger.

`vbroker.naming.url`

This property specifies the location of the database which you want to access. The setting is dependent on the database in use. The default is `JDataStore` and the database location is the current directory and is called `rootDB.jds`. You can use any name you like not necessarily `rootDB.jds`. The configuration file needs to be updated accordingly.

| URL value | Description |
|---|---|
| `jdbc:borland:dslocal:<db_name>` | JDataStore URL |
| `jdbc:sybase:Tds:<host>:<port>/<db_name>` | Sybase URL |
| `jdbc:oracle:thin:@<host>:<port>:<sid>` | Oracle URL |
| `jdbc:interbase://<server>/<full_db_path>` | Interbase URL |
| `jdbc:weblogic:mssqlserver4:<db_name>@<host>:<port>` | WebLogic MS SQLServer URL |
| `jdbc:db2://<host_name>:<port-number>/<db_name>` | IBM DB2 URL |
| `<full_path_JDataStore_db>` | DataExpress URL for the native driver |

[1]You should start InterServer before accessing InterBase via JDBC. If the InterBase server resides on the local host, specify `<server>` as `localhost`; otherwise specify it as the host name. If the InterBase database resides on Windows NT, specify the `<full_db_path>` as `driver:\\dir1\dir2\\db.gdb` (the first backslash [\] is to escape the second backslash [\]). If the InterBase database resides on UNIX, specify the `<full_db_path>` as `\dir1\dir2\db.gdb`. You can get more information from `http://www.borland.com/interbase/`.

[2]Before you access DB2 via JDBC, you must register the database by its alias <db_name> using the Client Configuration Assistant. After the database has been registered, you do not have to specify `<host>` and `<port>` for the `vbroker.naming.url` property.

[3] If the JDataStore database resides on Windows, the `<full path of the JDataStore database>` should be `Driver:\\dir1\\dir2\\db.jds` (the first backslash [\] is to escape the second backslash [\]). If the JDataStore database resides on UNIX, the `<full path of the JDataStore database>` should be `/dir1/dir2/db.jds`.

### DataExpress Adapter properties

The following table describes the DataExpress Adapter properties:

| Property | Description |
|---|---|
| `vbroker.naming.backingStoreType` | This property should be set to `Dx`. |
| `vbroker.naming.loginName` | This property is the login name associated with the database. The default is `VisiNaming`. |
| `vbroker.naming.loginPwd` | This property is the login password associated with the database. The default value is `VisiNaming`. |
| `vbroker.naming.url` | This property specifies the location of the database. |

### JNDI adapter properties

The following is an example of settings that can appear in the configuration file for a JNDI adapter:

| Setting | Description |
|---|---|
| `vbroker.naming.backingStoreType=JNDI` | This setting specifies the backing store type which is `JNDI` for the JNDI adapter. |
| `vbroker.naming.loginName=<user_name>` | The user login name on the JNDI backing server. |
| `vbroker.naming.loginPwd=<password>` | The password for the JNDI backing server user. |
| `vbroker.naming.jndiInitialFactory=com.sun.jndi.ldap.LdapCtxFactory` | This setting specifies the JNDI initial factory. |
| `vbroker.naming.jndiProviderURL=ldap://<hostname>:389/<initial root context>` | This setting specifies the JNDI provider URL |
| `vbroker.naming.jndiAuthentication=simple` | This setting specifies the JNDI authentication type supported by the JNDI backing server. |

## Configuration for OpenLDAP

OpenLDAP is one of the supported VisiNaming back-end stores. When using `OpenLDAP`, additional configuration is required on the OpenLDAP server. You must perform the following actions:

1. Add `corba.schema` in the `OpenLDAP` server's config file (the default is `slapd.conf`). The `corba.schema` is included with your `OpenLDAP` server installation.
2. Add `openldap_ns.schema` in the OpenLDAP config file. `openldap_ns.schema` is provided with VisiBroker and is located in

   ```
   <install-dir>/etc/ns_schema/
   ```

**Note**

The user must have the necessary privilege to add schemas/attributes to the Directory Server.

## Caching facility

By enabling the caching facility you can improve the performance of the Naming Service when it uses a backing store. For example, in the case of the JDBC adapter, directly accessing the database every time there is a resolve or bind operation is relatively slow. If you cache the results, you can reduce the number of times you access the database. You will only see improvement in the performance of the backing store if the same piece of data is accessed multiple times.

**Note**

Multiple Naming Service instances can access the same backing store if they are configured in the Naming Service Cluster mode or in the Master/Slave mode. In order to use the caching facility in these two modes, each Naming Service instance must be specially configured using the `vbroker.naming.cache.connectString` property. The

VisiBroker Event Service or VisiNotify is used to coordinate the caching facility amongst the various Naming Service instances.

To enable the caching facility set the following property in your configuration file:

```
vbroker.naming.cacheOn=1
```

If multiple Naming Service instances in Cluster or Master/Slave mode will access the cache, set the `vbroker.naming.cache.connectString` property so that the Naming Services can locate the Event Service (or VisiNotify).

The format for `vbroker.naming.cache.connectString` is:

```
vbroker.naming.cache.connectString=<host>:<port>
```

Where `<host>` is the hostname or IP address of the machine where VisiBroker Event Service is running and `<port>` is the port used by VisiBroker Event Service/VisiNotify (default is 14500 for Event Service and 14100 for VisiNotify).

For example:

```
vbroker.naming.cache.connectString=127.0.0.1:14500
```

or

```
vbroker.naming.cache.connectString=myhost:14100
```

If the host address is an IPv6 style address then enclose it in square brackets.

To support IPv6 on windows, a new property has been introduced. See “Properties related to Server-side resource usage” for more information.

**Note**

The VisiBroker Event Service (version 6.5 or later) should be started before starting the Naming Service instances. If VisiNotify is used instead, VisiNotify should be started. Start the Event Service/VisiNotify without any channel name (so the default name is used) before Naming Service instances are started.

If the cache needs tuning, set the following properties:

```
vbroker.naming.cache.size
vbroker.naming.cache.timeout
```

See “Properties file” for more information about the caching facility properties.

### Important Notes for users of Caching Facility

**Consistent configuration is very important.** It is extremely important to configure all Naming Service instances in a Cluster to use the Caching Facility in a consistent manner. Naming Service instances that constitute a Cluster must either **all** use the caching facility or **none** use it. If certain Naming Service instances use the caching facility while others do not, the behaviour of the Cluster will be inconsistent. This is also true for Naming Services configured in the Master-Slave mode. If the Master is configured to use the caching facility, it is required that the Slave also be configured to use it, and vice versa.

**The distributed cache depends on the Event Service/VisiNotify.** If the Caching Facility is used in Naming Service Cluster mode (or the Master-Slave mode), the distributed cache needs synchronization across the multiple Naming Services instances. This is achieved using the Event Service (or VisiNotify). Please note that in such a configuration, the cached data might be stale. The quality of data would depend on the health of the Event Service/VisiNotify. Applications that do not find this acceptable are advised to avoid using the Caching Facility. It is advisable to perform tests to gauge the suitability of the distributed Caching Facility for a particular application.

## Object Clusters

VisiBroker supports a clustering feature which allows a number of object bindings to be associated with a single name. The VisiNaming Service can then perform load balancing among the different bindings in a cluster. You can decide on a load balancing criterion at the time a cluster is created. Clients, which subsequently resolve name-object bindings against a cluster, are load balanced amongst different cluster server members. These clusters of object bindings should not be confused with "VisiNaming Service Clusters for Failover and Load Balancing".

A cluster is a multi-bind mechanism that associates a `Name` with a group of object references. The creation of a cluster is done through a `ClusterManager` reference. At creation time, the `create_cluster` method for the `ClusterManager` takes in a string parameter which specifies the criterion to be used. This method returns a reference to a cluster, which you can add, remove, and iterate through its members. After deciding on the composition of a cluster, you can bind its reference with a particular name to any context in a VisiNaming Service. By doing so, subsequent resolve operations against the `Name` will return a particular object reference in this cluster.

## Object Clustering criteria

The VisiNaming Service uses a `SmartRoundRobin` criterion with clusters by default. After a cluster has been created, its criterion cannot be changed. User-defined criteria are not supported, but the list of supported criteria will grow as time goes on. `SmartRoundRobin` performs some verifications to ensure that the CORBA object reference is an active one; that the object reference is referring to a CORBA server which is in a ready state.

### Cluster and ClusterManager interfaces

Although a cluster is very similar to a naming context, there are certain methods found in a context that are not relevant to a cluster. For example, it would not make sense to bind a naming context to a cluster, because a cluster should contain a set of object references, not naming contexts. However, a cluster interface shares many of the same methods with the `NamingContext` interface, such as `bind`, `rebind`, `resolve`, `unbind` and `list`. This common set of operations mainly pertains to operations on a group. The only cluster-specific operation is `pick`. Another crucial difference between the two is that a cluster does not support compound names. It can only use a single component name, because clusters do not have a hierarchical directory structure, rather it stores its object references in a flat structure.

#### IDL Specification for the Cluster interface

```
CosNamingExt module {
   typedef sequence<Cluster> ClusterList;
   enum ClusterNotFoundReason {
      missing_node,
      not_context,
      not_cluster_context
   };
   exception ClusterNotFound {
      ClusterNotFoundReason why;
      CosNaming::Name rest_of_name;
   };
   exception Empty {};
   interface Cluster {
      Object select() raises(Empty);
      void bind(in CosNaming::NameComponent n, in Object obj)
         raises(CosNaming::NamingContext::CannotProceed,
            CosNaming::NamingContext::InvalidName,
            CosNaming::NamingContext::AlreadyBound);
      void rebind(in CosNaming::NameComponent n, in Object obj)
         raises(CosNaming::NamingContext::CannotProceed,
            CosNaming::NamingContext::InvalidName);
      Object resolve(in CosNaming::NameComponent n)
         raises(CosNaming::NamingContext::NotFound,
            CosNaming::NamingContext::CannotProceed,
            CosNaming::NamingContext::InvalidName);
      void unbind(in CosNaming::NameComponent n)
         raises(CosNaming::NamingContext::NotFound,
            CosNaming::NamingContext::CannotProceed,
            CosNaming::NamingContext::InvalidName);
      void destroy()
         raises(CosNaming::NamingContext::NotEmpty);
      void list(in unsigned long how_many,
         out CosNaming::BindingList bl,
         out CosNaming::BindingIterator BI);
};
```

## IDL Specification for the ClusterManager interface

```
CosNamingExt module {
   interface ClusterManager
      Cluster create_cluster(in string algo);
      Cluster find_cluster(in CosNaming::NamingContext ctx, in CosNaming::Name
n)
         raises(ClusterNotFound, CosNaming::NamingContext::CannotProceed,
      CosNaming::NamingContext::InvalidName);
      Cluster find_cluster_str(in CosNaming::NamingContext ctx, in string n)
         raises(ClusterNotFound, CosNaming::NamingContext::CannotProceed,
      CosNaming::NamingContext::InvalidName);
      ClusterList clusters();
   };
};
```

## IDL Specification for the NamingContextExtExtended interface

The `NamingContextExtExtended` interface, which extends `NamingContextExt`, provides some operations required to remove an object reference from an implicit cluster. You must narrow a `NamingContext` to `NamingContextExtExtended` in order to use these operations. Note that these operations are proprietary to VisiBroker only.

```
module CosNamingExt {
  interface NamingContextExtExtended : NamingContextExt {
    void unbind_from_cluster(in Name n, in Object obj)
     raises(NotFound, CannotProceed, InvalidName);
    boolean is_ncluster_type(in Name n, out Object cluster)
     raises(NotFound, CannotProceed, InvalidName);
  };
}
```

`unbind_from_cluster()`

The `unbind_from_cluster()` method allows user to unbind a specific object in a cluster. The object's logical name (such as “London.Branch/Jack.SavingAccount”) and the object reference to be unbound need to be passed into this method. Whenever the number of objects in the cluster reaches zero, the cluster is deleted as well.

This method is useful when automatic pruning of stale object references in a cluster is not required. Call this method to unbind an object in a cluster based on the application's specific rules.

**Note**

The `unbind_from_cluster()` method can only be used when the VisiNaming Service is running in the implicit clustering mode and automatic pruning of stale object reference is disabled. This means that the following two properties must be set at the VisiNaming Service side:

```
vbroker.naming.smrr.pruneStaleRef=0
vbroker.naming.propBindOn=1
```

`is_ncluster_type()`

The `is_ncluster_type()` method lets you check whether a context is of a cluster type. The object's logical name need to be passed into this method. It returns `true` when the context is a cluster type and set the cluster object in the second argument value. It returns `false` when the context is not a cluster type and set the second argument value to null.

## Creating an object cluster

To create a cluster, use the Cluster Manager interface. A single `ClusterManager` object is automatically created when a Naming Server starts up. There is only one `ClusterManager` per Naming Server. The role of a `ClusterManager` is to create, retrieve, and keep track of the clusters that are in the Naming Server. Here are the general steps in creating an object cluster:

1. Bind to the Naming Server with which you wish to create cluster objects.
2. Get a reference to the Cluster Manager by calling `get_cluster_manager` method on the factory reference.
3. Create a cluster using a specified cluster criterion.
4. Bind objects to an Name using the cluster.
5. Bind the `Cluster` object itself to a Name.
6. Resolve through the Cluster reference for the specified cluster criterion.

```
...
ExtendedNamingContextFactory_var myFactory =
      ExtendedNamingContextFactory::_bind(orb, "NamingService");
ClusterManager_var clusterMgr = myFactory->get_cluster_manager();
Cluster_var clusterObj = clusterMgr->create_cluster("RoundRobin");
clusterObj->bind(new NameComponent("member1", "aCluster"), obj1);
clusterObj->bind(new NameComponent("member2", "aCluster"), obj2);
clusterObj->bind(new NameComponent("member3", "aCluster"), obj3);
NameComponent_var myClusterName = new NameComponent("ClusterName", "");
root->bind(myClusterName, clusterObj);
root->resolve(myClusterName); // a member of the Cluster is returned
root->resolve(myClusterName); // the next member of the Cluster is returned
root->resolve(myClusterName); // the last member of the Cluster is returned
...
```

### Explicit and implicit object clusters

The clustering feature can be turned on automatically for a VisiNaming Service. The caveat is that once this facility is on, a cluster is created transparently to bind the object. The round robin criterion is used. The implication is that it is possible to bind several objects to the same name in the Naming Server. Conversely, resolving that name will return one of those objects, and an `unbind` operation would destroy the cluster associated with that name. This means that the VisiNaming Service is no longer compliant to the CORBA specification. The *Interoperable Naming Specification* explicitly forbids the ability to bind several objects to the same name. For a compliant VisiNaming Service, an `AlreadyBound` exception is thrown if a client tries to use the same name to bind to a different object. You must decide whether to use this feature for a dedicated server only.

**Note**

Do not switch from an implicit cluster mode to an explicit cluster mode as this can corrupt the backing store.

Once a Naming Server is used with the implicit clustering feature, it must be activated with that feature turned “on”. To turn on the clustering feature, define the following property value in the configuration file:

```
vbroker.naming.propBindOn=1
```

**Note**

For an example of both explicit and implicit clustering, see the code located in the following directories:

```
<install_dir>/examples/vbroker/ins/implicit_clustering
<install_dir>/examples/vbroker/ins/explicit_clustering
```

## Load balancing

Both the ClusterManager and the Smart Agent provide RoundRobin criterion load balancing facilities, however, they are of very different nature. You get load balancing from the Smart Agent transparently. When a server starts, it registers itself automatically with the Smart Agent, and this in turn allows VisiBroker ORB to provide an easy and proprietary way for the client to get a reference to the server. However, you have no choice in determining what constitutes a group and the members of a group. The Smart Agent makes all the decisions for you. This is where a Cluster provides an alternative. It enables a programmatic way to define and create the properties of a Cluster. You can define the criterion for a Cluster, including choosing the members of a Cluster. Though the criterion is fixed at creation time, the client can add or remove members from the Cluster throughout its existance.

## Object failover

An advantage of using object clustering is the failover capability among the objects clustered together in a VisiNaming service. These clustered objects support the same interface. Once such a cluster is created and bound to a naming context, the failover behavior is transparently handled by the ORB. Typically when a naming service client does a resolution against this cluster, the VisiNaming service returns a member from the cluster. In case any member of the cluster has crashed or is temporarily unavailable, ORB and VisiNaming service perform transparent failover by handing over the next available cluster member to the client. This ensures high availability and fault-tolerance.

Failover capability using object clustering is demonstrated in the example contained in the following directory:

```
<install_dir>/examples/vbroker/ins/cluster_failover
```

## Pruning stale object references in VisiNaming object clusters

Object references in VisiNaming service can become stale due to unavailability of the servers. Implicit object clustering provides different strategies, which can be used to configure the pruning of stale references. Note that this pruning facility only works in implicit clustering using smart round-robin technique. VisiNaming service is started with a pruning configuration using the property `vbroker.naming.smrr.pruneStaleRef`. This property can take values 0, 1 (default) and 2. The working of pruning facility can be understood as follows:

VisiNaming service holds the mapping between the names and object references in the memory. When a client requests for an object reference against a name, VisiNaming resolves the name, modifies the IOR and hands over the object reference to the client. The modification pertains to putting the logic that in case, the server represented by the object reference in unavailable, the client ORB, to which this object reference is being handed to, can revert back to the VisiNaming service to look for an alternate object reference (fail-over to another candidate). If the client is unable to find the server and it does revert back to the VisiNaming service, VisiNaming marks that object reference as stale.

Depending on the value of the property `vbroker.naming.smrr.pruneStaleRef`, VisiNaming decides whether to keep the object reference or remove it. Following are the possible values:

- `vbroker.naming.smrr.pruneStaleRef =0`
  In this case, if an object reference has been detected stale, VisiNaming only marks it as stale but does not remove it from its in-memory hold. However, VisiNaming does

not ever hand over this reference to the client unless the server rebinds the object reference against the same name.

- `vbroker.naming.smrr.pruneStaleRef =1`
  VisiNaming service immediately removes the object reference both from the memory and persistent backstore (if backing store is being used) as soon as the client bounces back to the VisiNaming service indicating the object reference as stale.
- `vbroker.naming.smrr.pruneStaleRef =2`
  In this case, VisiNaming does not modify the IOR before handing it over to the client. In case the client is not able to contact the server represented by the object reference, client ORB throws OBJECT_NOT_EXISTS exception back to the client application. VisiNaming services does not take guarantee of providing the client application with an active object reference.

## VisiNaming Service Clusters for Failover and Load Balancing

Multiple instances of the VisiNaming Service can be clustered to provide for load balancing and failover. These clusters of VisiNaming Service instances should not be confused with the clustering of object bindings described in "Object Clusters". Clients can bind to any one of the VisiNaming Service instances that comprise the cluster, which allows for load sharing across multiple VisiNaming Service instances. If a particular VisiNaming Service instance becomes inactive or terminates, the client will automatically fail over to another VisiNaming Service instance within the same cluster.

All instances of the VisiNaming Service within a cluster must use the common underlying data in a persistent backing store. The caching facility is available to Naming Service instances provided that a VisiBroker Event Service (or VisiNotify) instance is made available to the Naming Service instances via the `vbroker.naming.cache.connectString` property. There are certain restrictions regarding the choice of backing store. See the following Note that discusses these restrictions.

When failover occurs, it is transparent to the client, but there can be a slight delay because server objects might have to be activated on demand by the requests that are coming in. Also, object reference transients like iterator references are no longer valid. This is normal because clients using transient iterator references must be prepared for those references becoming invalid. In general, a VisiNaming Service instance never keeps too many resource-intensive iterator objects, and it may invalidate a client's iterator reference at any time. Other than these transient references, any other client request using persistent references will be rerouted to another VisiNaming Service instance.

In addition to the VisiNaming Service cluster, a Master/Slave model is also supported. This is a special cluster with the configuration of two VisiNaming Service instances. It is useful only when failover is required. The two VisiNaming Services instances must be running at the same time; the master in active mode and the slave in standby mode. If both VisiNaming Services are active, the master is always preferred by clients that are using VisiNaming Service. In the event that the master terminates unexpectedly, the slave VisiNaming Service takes over. This changeover from master to slave is seamless and transparent to clients. However, the slave VisiNaming Service does not become the master server. Instead, it provides temporary backup when the master server is unavailable. You must take whatever remedial actions necessary to revive the master server. After the master comes back up again, only requests from the new clients are sent to the master server. Clients that are already bound to a slave naming server will not automatically switch back to the master.

**Note**

Clients that are bound to a slave naming server provide only one level of failover support. They will not switch back to the master, therefore, if the slave naming server terminates, the VisiNaming Service also becomes unavailable.

**Note**

VisiNaming Service Clusters configured in the Master/Slave mode may use either the JNDI adapter or the JDBC adapter. Clusters not configured in the Master/Slave mode must use the JDBC adapter for RDBMS. Each clustered service must obviously point to the same backing store. See “Pluggable backing store” for information on configuring the backing store for the cluster.

## Configuring the VisiNaming Service Cluster

The VisiNaming Service instances that comprise the cluster must be started with the relevant properties set as illustrated in the code sample below. The configuration is set to cluster mode using the `enableSlave` and the `slaveMode` properties. The instances of the VisiNaming Service that comprise the cluster have to be started on the hosts and ports specified using the `serverAddresses` property. The snippet shows the host and port entries for the three VisiNaming Service instances in the sample cluster. The `serverNames` property lists the factory names of the VisiNaming Service instances. These names are unique and the ordering identical to the `serverAddresses` property. Finally, the `serverClusterName` property names the cluster.

**Note**

Starting from VisiBroker 6.0, VisiNaming Service contains several properties for proxy support:

- `vbroker.naming.proxyEnable` allows the VisiNaming Service to use a proxy. Turn off this property (default is turned off), and the VisiNaming Service will ignore other Naming service properties for the proxy.
- `vbroker.naming.proxyAddresses` gives each Naming service in the cluster a proxy host and a proxy port. The ordering of the proxyAddresses is identical to the serverAddresses.

C++ clients need to set the boolean property `vbroker.naming.anyServiceOrder` in order to benefit from the load-balancing and failover capabilities provided by VisiNaming Service clusters. Clients must use the `corbaloc` mechanism to resolve to a VisiNaming Service instance within the cluster, provided osagent is being used.

The Naming Service instances comprising a Cluster can benefit from the Naming Service Caching Facility. Use the `vbroker.naming.cacheOn` and `vbroker.naming.cache.connectString` properties to configure caching for a Naming Service cluster. See “Caching facility” for details.

The following code sample shows the configuration of the VisiNaming Service cluster:

```
vbroker.naming.enableSlave=1
vbroker.naming.slaveMode=cluster
vbroker.naming.serverAddresses=host1:port1;host2:port2;host3:port3
vbroker.naming.serverNames=Server1:Server2:Server3
vbroker.naming.serverClusterName=ClusterX
vbroker.naming.proxyEnable=1 //Any value other than 1 means proxy is not
enabled.
vbroker.naming.proxyAddresses=proxyHost1:proxyPort1;proxyHost2:proxyPort2;proxy
Host3:proxyPort3
```

**Note**

When using the `vbroker.naming.proxyAddresses` property, place a semicolon (;) separator between each host and port pair.

## Configuring the VisiNaming Service in Master/Slave mode

The two VisiNaming Services must be running. You must designate one as the master and the other as the slave. The same property file can be used for both the servers. The relevant property values in the property file are shown in the following code sample to configure for the Master/Slave mode.

```
vbroker.naming.enableSlave=1
vbroker.naming.slaveMode=slave
vbroker.naming.masterServer=<Master Naming Server Name>
vbroker.naming.masterHost=<host ip address for Master>
vbroker.naming.masterPort=<port number that Master is listening on>
vbroker.naming.slaveServer=<Slave Naming Server Name>
vbroker.naming.slaveHost=<host ip address for Slave>
vbroker.naming.slavePort=<Slave Naming Server port address>
vbroker.naming.masterProxyHost=<proxy host ip address for Master>
vbroker.naming.masterPortPort=<proxy port number for Master>
vbroker.naming.slaveProxyHost=<proxy host ip address for Slave>
vbroker.naming.slavePortPort=<proxy port number for slave>
```

**Note**

There is no restriction in the start sequence of the master and the slave servers.

## Starting up with a large number of connecting clients

In a production environment with a large number of clients it may be impossible to avoid clients trying to connect to a Naming Service which is still in the startup phase (still initializing and not yet ready to service requests). When a Naming Service is not yet completely started up it may receive incoming requests and discard them. Depending on the number of requests, which must be received then discarded, this activity can use too many CPU resources which can disturb the startup process itself, resulting in a long startup time for the Naming Service.

To solve this particular problem, and let the Naming Service start quickly, the following configuration settings can be used:

1 Set the following property to `true`:

   ```
   vbroker.se.iiop_tp.scm.iiop_tp.listener.deferAccept=true
   ```

2 Use a fixed listener port by setting the following properties:

   ```
   vbroker.se.iiop_tp.scm.iiop_tp.scm.listener.port=<port_number>
   vbroker.se.iiop_tp.scm.iiop_tp.listener.portRange=0
   ```

   For this to succeed, make sure that the <port_number> is available on the host on which the Naming Service is running. Make sure that the `portRange` property is set to `0` (zero). You can leave it at its default setting or explicitly set the property. Note that both the `port` and `portRange` settings described above should be applied.

Clients that try to connect to a Naming Service configured in this manner while it is starting up will be denied any connection. If they are accessing a Naming Service Cluster, then they would fail over to another Naming Service that has finished its initialization. If no Naming Services are up and running, the client application would get an `OBJECT_NOT_EXIST` exception.

These settings are per SCM (Server Connection Manager). If needed, all SCMs can be set to take advantage of this feature.

If SSL is involved in the Naming Service, in addition to the settings described above, the following settings might also be needed:

```
vbroker.se.iiop_tp.scm.ssl.listener.deferAccept=true
vbroker.se.iiop_tp.scm.ssl.listener.port=<port_number_for_ssl>
vbroker.se.iiop_tp.scm.ssl.listener.portRange=0
```

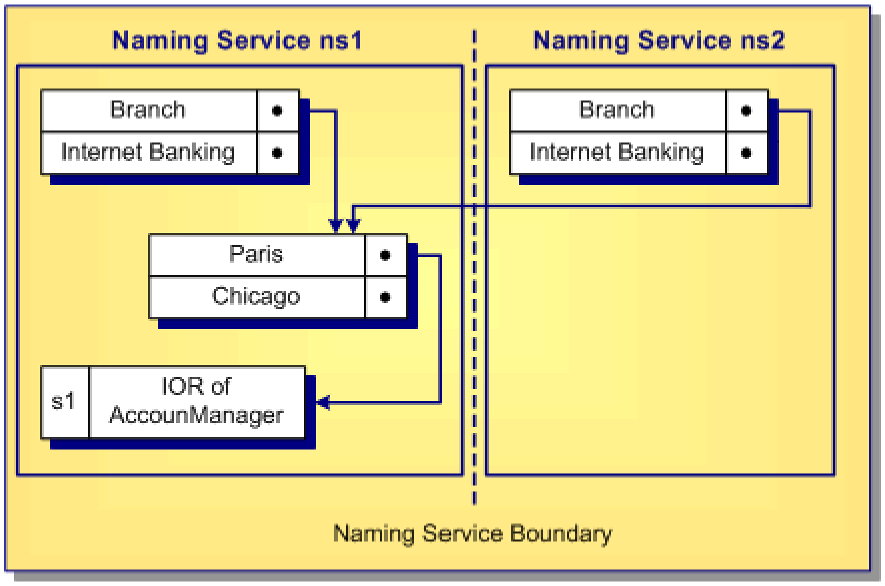
**Note**

The `deferAccept` property should only be used for Naming Services. Using for other services or user written servers can result in undefined behavior.

## VisiNaming service federation

Federation enables more than one VisiNaming services to be configured to act as a distributed namespace. This involves having a naming context in a name service bound to the names in the naming contexts of other naming services, thereby providing more than one naming hierarchy to access an object. The figure below shows two instances of naming service `ns1` and `ns2`. Grayed naming contexts are the initial contexts of the respective naming services. An AccountManager object `s1` is placed in a naming context under `ns1`.

**Figure 15.3** Naming contexts with multiple access hierarchies



As shown in the figure, naming context containing `Paris` is bound to `Branch` under naming service `ns1` and also bound to `Remote` under naming service `ns2`. Client can retrieve the IOR of the `AccountManager` object against `s1` either by resolving `ns1: Branch/Paris/s1` or `ns2: Branch/Paris/s1`. In both cases, it gets the same IOR.

Setting up federation is as easy as binding the name `Branch` in the root context of `ns2` in the above example to the naming context containing the name `Paris` in `ns1`. The example in the following location shows the working of VisiNaming federation:

```
<install_dir>/examples/vbroker/ins/federation
```

# VisiNaming Service Security

The VisiNaming Service in the VisiBroker integrates with the Security Service, providing two levels of security: Client authentication and Method level authorization. This allows fine grained control over which clients can use the VisiNaming Service and what methods they can call. The following properties are used to enable or disable security and to configure the Security Service.

| Property | Value | Default | Description |
|---|---|---|---|
| `vbroker.naming.security.disable` | boolean | `true` | This property indicates whether the security service is disabled. |
| `vbroker.naming.security.authDomain` | string | "" | This property indicates the authorization domain name to be used for the Naming service method access authorization. |

| Property | Value | Default | Description |
|---|---|---|---|
| `vbroker.naming.security.transport` | int | `3` | This property indicates what transport to be used. The available values are: `ServerQoPPolicy.SECURE_ONLY=1` `ServerQoPPolicy.CLEAR_ONLY=0` `ServerQoPPolicy.ALL=3` |
| `vbroker.naming.security.requireAuthentication` | boolean | `false` | This property indicates whether naming client authentication is required. When `vbroker.naming.security.disable` is `true`, no client authentication will be performed regardless what value this property takes. |
| `vbroker.naming.security.enableAuthorization` | boolean | `false` | This property indicates whether method access authorization is enabled. |
| `vbroker.naming.security.requiredRolesFile` | string | (none) | This property points to the file containing the required roles that are necessary for invocation of each method in the protected object types. For more information see "Method Level Authorization". |

## Naming client authentication

**Note**

For detailed information on authentication and authorization, see the "Authentication" and "Authorization" chapters of the *Borland VisiBroker Security Guide*.

## Configuring VisiNaming to use SSL

Depending on the security requirements, different properties can be set to configure the VisiNaming service. For the full list of security properties and their descriptions, go to the Security Guide, "Security Properties for Java" or the "Security Properties for C++" section.

**Important**

In order to enable security in the VisiNaming Service, you must have a valid VisiSecure license.

The following is a sample of the properties that can be used to configure the VisiNaming Service to use SSL:

```
  # Enable Security in Naming Service
  vbroker.naming.security.disable=false

  # Enabling Security Service
  vbroker.security.disable=false

  # Setting SSL Layer Attributes
  vbroker.security.peerAuthenticationMode=REQUIRE_AND_TRUST
  vbroker.se.iiop_tp.scm.ssl.listener.trustInClient=true
  vbroker.security.trustpointsRepository=Directory:./trustpoints

  # Set the certificate identity for the VisiNaming Service using wallet
properties
  vbroker.security.wallet.type=Directory:./identities
  vbroker.security.wallet.identity=delta
  vbroker.security.wallet.password=Delt@$$$
```

For information about how to configure the client to use SSL, go to the Security Guide, “Making secure connections (Java)” or the “Making secure connections (C++)” section.

**Note**

Currently, there is no way to specify security and secure transport components in an IOR using corbaloc. So, when using SSL, bootstrapping a VisiNaming Service using the corbaloc method at the Naming client side is not possible. However, the SVCnameroot and stringified IOR methods can still be used.

## Method Level Authorization

Method level authorization is supported for the following object types:

- `Context`
- `ContextFactory`
- `Cluster`
- `ClusterManager`

When security is enabled for the Naming service and `enableAuthorization` is set to `true`, only authorized users of each method of these object types can invoke the corresponding method.

The Naming service predefines two roles to support the method level authorization:

- Administrator role
- User role

Other roles can be defined if required. Users need to configure the roles map for these two roles, assigning roles to clients. The following is an example role map definition:

```
Administrator {
*CN=admin
*group=admin
uid=*, group=admin
}

User {
*CN=admin
*group=user
uid=*, group=user
}
```

You need to specify the roles before invoking each method of the objects listed above. This is done using the `required_roles` property for each method. Below is the list of these properties and the corresponding default values. These default values are used only when you do not define any `required_roles` specified using the property `vbroker.naming.security.requiredRolesFile`. The values of these properties are space or comma separated:

```
#
# naming_required_roles.properties
#

# all roles
required_roles.all=Administrator User

required_roles.Context.bind=Administrator
required_roles.Context.rebind=Administrator
required_roles.Context.bind_context=Administrator
required_roles.Context.rebind_context=Administrator
required_roles.Context.resolve=Administrator User
required_roles.Context.unbind=Administrator
required_roles.Context.new_context=Administrator User
required_roles.Context.bind_new_context=Administrator User
required_roles.Context.list=Administrator User
required_roles.Context.destroy=Administrator

required_roles.ContextFactory.root_context=Administrator User
required_roles.ContextFactory.create_context=Administrator
required_roles.ContextFactory.get_cluster_manager=Administrator User
required_roles.ContextFactory.remove_stale_contexts=Administrator
required_roles.ContextFactory.list_all_roots=Administrator
required_roles.ContextFactory.shutdown=Administrator

required_roles.Cluster.select=Administrator User
required_roles.Cluster.bind=Administrator
required_roles.Cluster.rebind=Administrator
required_roles.Cluster.resolve=Administrator User
required_roles.Cluster.unbind=Administrator
required_roles.Cluster.destroy=Administrator
required_roles.Cluster.list=Administrator User

required_roles.ClusterManager.create_cluster=Administrator
required_roles.ClusterManager.find_cluster=Administrator User
required_roles.ClusterManager.find_cluster_str=Administrator User
required_roles.ClusterManager.clusters=Administrator User
```

## Compiling and linking programs

C++ applications that use the Naming service need to include the following generated files:

```
#include "CosNaming_c.hh"
#include "CosNamingExt_c.hh"
```

### UNIX

The UNIX applications need to be linked with the `cosnm_r.so` (multi-threaded) library.

### Windows

The Windows applications need to be linked with the `cosnm_r.lib (cosnm_r_6.dll)` (multi-threaded) library.

## Sample programs

Several example programs that illustrate the use of the VisiNaming Service are provided with VisiBroker. They show all of the new features available with the VisiNaming Service and are found in the `<install_dir>/examples/vbroker/ins` directory. In addition, a Bank Naming example illustrates basic usage of the VisiNaming Service is found in the `<install_dir>/examples/vbroker/basic/bank_naming` directory.

Before running the example programs, you must first start the VisiNaming Service, as described in "Running the VisiNaming Service". Furthermore, you must ensure that at least one naming context has been created by doing one of the following:

- Start the VisiNaming Service, as described in "Running the VisiNaming Service" which will automatically create an initial context.
- Use the VisiBroker Console.
- Have your client bind to the `NamingContextFactory` and use the `create_context` method.
- Have your client use the `ExtendedNamingContextFactory`.

**Important**

If no naming context has been created, a `CORBA::NO_IMPLEMENT` exception is raised when the client attempts to issue a `CosNaming::NamingContext::bind`.

## Configuring VisiNaming with JdataStore HA

This section helps you configure JDataStore High Available (HA) to work with VisiNaming.

The Explicit Clustering example used throughout this section illustrates the usage of JDataStore HA with VisiNaming. In this example, JDataStore will be configured to have the following mirror types:

- One Primary mirror. This is the only mirror type that can accept both read and write transactions. Only one Primary mirror at a time is allowed.
- Three Read-only mirrors. These can only perform read transactions, and they provide a transactionally consistent view of the Primary mirror database.
- One Directory mirror. This contains only the mirror configuration table and other system security tables. It redirects read-only connection requests to Read-only mirrors, and writable connection requests to the Primary mirror. It also provides an important feature for load balancing all read connections across all available Read-only mirrors. However, this feature is not supported by Naming Service at this version.

JDataStore HA supports automatic failover in the following circumstances:

- If a connection to the Primary mirror was made before the failure, this connection can trigger an automatic failover by calling the rollback method on the connection object. Note that this scenario is not described in this section.
- If the connection request is not for read-only operation, and the current Primary mirror is not accessible, the Directory mirror automatically triggers the failover operations to satisfy the request for a writable connection. This is done by promoting one of the Read-only mirrors to the Primary mirror.

VisiNaming works with JDataStore HA when a connection is made to the Directory mirror. When the Primary mirror is inaccessible, it will failover to one of the Read-only mirrors. VisiNaming must work with one Primary, and at least two Read-only mirrors at all times.

**Notes**

- The Directory Mirror is a single point of failure in the scenario described in this section. Higher availability could be achieved by configuring Master and Slave Naming Services to point to a different directory mirror.
- JDataStoreHA only works with JDataStore Version 7.04 or later.

### Create a DB for the Primary mirror

To make use of the JDataStore Explorer (JdsExplorer) to create a new DB, select `New` from the File menu.

### Invoke JdsServer for each listening connection

In this example, the following connections are used:

- JdsServer –port 2511  (Primary mirror)
- JdsServer –port 2512  (Read-only mirror)
- JdsServer –port 2513  (Read-only mirror)

- JdsServer –port 2514 (Read-only mirror)
- JdsServer –port 2515 (Directory mirror)

**Note**

Always start JdsServer from the location where the `AutoFailover_*` jds files are located. Never start JdsServer from `<JdataStore Install Directory>/bin` unless `vbroker.naming.url` is set according. The required jar files are:

- dbtools.jar
- dbswing.jar
- jdsremote.jar
- jdsserver.jar
- jds.jar

## Configure JDataStore HA

To configure JDataStore HA, complete the following steps:

1. Invoke the JDS Server Console to configure JDataStore.
2. Create a new project named `NS_AutoFailover` in the JDataStore Server Console.

**Note**

When creating a new DataSource, it is best to set its Protocol to Remote and include the machine IP in the ServerName

3. Click `DataSource1` (in the Structure pane) to open it for editing.
4. Right-click `DataSource1` and select `Connect` from the context menu.
5. Right-click `Mirror` (in the Structure pane) and select `Add mirror` from the context menu.
6. Edit Mirror1 so that the `Type` property is set to `PRIMARY`.

   Each of the mirrors should also ensure that the host uses the IP of the machine where they are located instead the default value of `localhost`. You can use a different IP address for each of the mirrors, as long as the JdsServer is started for that mirror at the IP. The Directory mirror must have access to each of the mirrors.
7. Set the `Auto Failover` and Instant Synchronization properties to `true`.
8. Add Mirror2 and edit it to be a Read-only mirror.

   Note that you do not need to create `AutoFailover_Mirror2` beforehand. It is created automatically by JDataStore HA.
9. Set the `Auto Failover` and Instant Synchronization properties to `true` for all Read-only mirrors.
10. Repeat the previous two steps for Mirror3 and Mirror4.
11. Add Mirror5 and edit it be the Directory mirror.
12. Set the `Auto Failover` and Instant Synchronization properties to `false` for this Directory mirror.
13. Choose `Save Project "NS_AutoFailover.datasources"` from the File menu to save the project.
14. Right-click `Mirrors` (in the Structure pane) and choose `Synchronize all mirrors`.
15. Click `Mirror Status` (in the Structure pane) and verify that `Validate Primary` is checked for Mirror1 only.

## Run the VisiNaming Explicit Clustering example

To run the VisiNaming Explicit Clustering example, complete the following steps:

1 Start osagent with the following command:

```
osagent
```

2 Create a file named `autofailover.properties` with the following properties:

```
vbroker.naming.backingStoreType=JDBC
vbroker.naming.poolSize=5
vbroker.naming.jdbcDriver=com.borland.datastore.jdbc.DataStoreDriver
vbroker.naming.url=jdbc:borland:dsremote://143.186.141.14/
AutoFailover_Mirror5.jds
vbroker.naming.loginName=SYSDBA
vbroker.naming.loginPwd=masterkey
vbroker.naming.traceOn=0
vbroker.naming.jdsSvrPort=2515
vbroker.naming.logLevel=debug
```

3 Start Naming Service with the following command:

```
nameserv -VBJclasspath <JDS_Install>\lib\
jdsserver.jar -config autofailover.properties
```

4 Start ServerA with the following command:

```
Server ServerA -ORBpropStorage ns_client.properties &
```

5 Start ServerB with the following command:

```
Server ServerB -ORBpropStorage ns_client.properties &
```

6 Start Client with the following command:

```
Client -ORBInitRef NameService=<nsIOR>
```

7 Repeat the previous step several times and observe the output.

To verify the minimum requirement of one Primary and two Read-only mirrors, complete the following steps:

1 Stop the JdsServer listening to port 2513.

2 Repeat the Start Client step several times.

Note that the behavior is the same as in the previous procedure.

3 Stop the JdsServer listening to port 2514.

4 Repeat the Start Client step several times.

Note that Client begins to raise a `BAD_PARAM exception`. This is as expected because a failover requires that at least two read-only mirrors are available.

5 Restart the JdsServer listening to port 2513 and 2514.

This restores the original configuration, with three Read-only mirrors.

To verify the autofailover of JDatastore HA, complete the following steps:

1 Stop the JdsServer listening to port 2511, configured for Primary mirror, and repeat the Start Client step several times.

Note that one of the Read-only mirrors has been promoted to Primary mirror.

2 Stop another active Read-only mirror and repeat the Start Client step several times.

Note that Client begins to raise a `BAD_PARAM exception` because a failover requires that at least two read-only mirrors are available.

3 Restart the JdsServer listening to port 2511.

Note that this was previously configured for Primary mirror.

4 Repeat the Start Client step several times.

Note that Mirror1 is now configured as Read-only mirror. You can check this from the JDS Server Console by making a datasource connection to the Directory mirror that the Naming Service uses.

## Run the VisiNaming Naming Failover example

Run the following example to observe the failover capability of the VisiNaming service.

**Note**

Before using this procedure, create a JDataStore HA with one Primary mirror at port 1111, three Read-only mirrors at ports 1112, 1113, 1114 and two Directory mirrors at ports 1115 and 1116.

1 Start osagent with the following command:

```
osagent
```

2 Create a file named `autofailover.properties` with the following properties:

```
# Naming
vbroker.naming.backingStoreType=JDBC
vbroker.naming.poolSize=5
vbroker.naming.jdbcDriver=com.borland.datastore.jdbc.DataStoreDriver
vbroker.naming.loginName=SYSDBA
vbroker.naming.loginPwd=masterkey
vbroker.naming.traceOn=0
vbroker.naming.jdsSvrPort=1115
#vbroker.naming.logLevel=debug
#default value of enableslave is 0. '1' Indicates cluster or
master-slave configuration
vbroker.naming.enableSlave=1
#indicate master-slave configuration
vbroker.naming.slaveMode=slave
vbroker.naming.masterHost=143.186.141.14
vbroker.naming.masterPort=12372
vbroker.naming.masterServer=Master
vbroker.naming.slaveHost=143.186.141.14
vbroker.naming.slavePort=12373
vbroker.naming.slaveServer=Slave
```

3 Start the JDataStore Servers as shown in the following example:

```
JdsServer.exe -port=1111
JdsServer.exe -port=1112
JdsServer.exe -port=1113
JdsServer.exe -port=1114
JdsServer.exe -port=1115
JdsServer.exe -port=1116
```

4 Start the Naming Service Master with the following command:

```
nameServ -VBJclasspath <JDS_Install>\lib\
jdsserver.jar -config autofailover.properties -VJprop
vbroker.naming.url=jdbc:borland:dsremote:// 143.186.141.12
AutoFailover_Mirror 5.jds

VBJprop vbroker.se.iiop_tp.scm.iiop_tp.listener.port=12372 Master
```

5 Start the Naming Service Slave with the following command:

```
nameserv -VBJclasspath <JDS_Install>\lib\

jdsserver.jar -config autofailover.properties -VBJprop

vbroker.naming.url=jdbc:borland:dsremote://143.186.141.14/
AutoFailover_Mirror6.jds

-VBJprop vbroker.se.iiop_tp.scm.iiop_tp.listener.port=12373 -VBJprop
vbroker.naming.jdsSvrPort=1116 Slave
```

6 Start Server with the following command:

```
Server -ORBInitRef NameService=<Master Server IOR>
```

7 Start Client with the following command:

```
NamingClient -ORBInitRef NameService=<Master Server IOR>
```

8 Press the *Enter* key and observe the output.
Note that the balance returns a value.

9 Stop the Naming Service Master, repeat the previous step, and observe the output.

Note that the balance returns a value.

10 Press the *Enter* key to exit, and observe the output.

Note that the balance returns a value

To see how two Directory mirrors handle a single point of failure, complete the following steps:

1 Stop the JdsServer listening to port 1115.

2 Without starting the Naming Service Master, repeat the Start Client step.
The `CannotProceed` exception is raised, which is the expected behavior.

3 Repeat the Start Client step several times.

Note that the balance will return a value. Once it can return a value, you can observe that it is using the Directory mirror that is listening on port 1117.

4 Repeat the Start Client step and press the *Enter* key three times.

Note that the balance returns a value for three times.

To see how autofailover functions with two Directory mirrors, complete the following steps:

1 Stop the JdsServer that is listening on port 1111.

2 Repeat the Start Client step.

3 Press the *Enter* key three times.

The `CannotProceed` exception is raised several times before it starts returning a value. Once it returns a value, you can see that one of the mirrors is promoted to be a Primary mirror. This can only be viewed using the JDS Server Console.

16

# Using the Event Service

This section describes the VisiBroker Event Service.

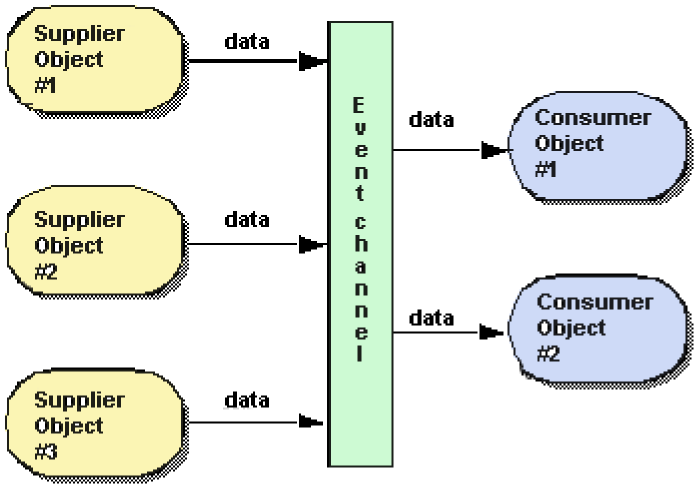
**Note**

The OMG Event Service has been superseded by the OMG Notification Service. The VisiBroker Event Service is still supported for backward compatibility and light weight purposes. For mission critical applications, we strongly recommend using VisiBroker VisiNotify. For more information, see "Introduction to VisiNotify."

## 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 16.1** Supplier-Consumer communication model



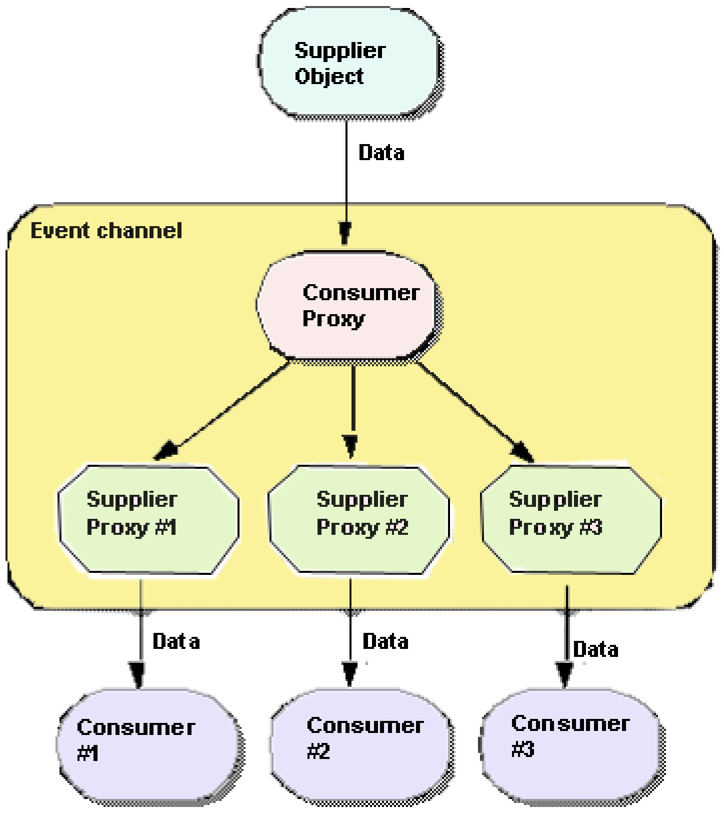
The figure above shows three supplier objects communicating through an event channel with two consumer objects. The flow of data into the event channel is handled by the supplier objects, while the flow of data out of the event channel is handled by the consumer objects. If each of the three suppliers shown in the figure above sends 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 is 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 interacting with each other directly, 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. The figure below shows how one supplier can distribute data to multiple consumers.

**Figure 16.2** Consumer and supplier proxy objects



**Note**

The event channel is shown above as a separate process, but it may also be implemented as part of the supplier object's process.

### OMG Common Object Services specification

The VisiBroker Event Service implementation conforms to the OMG Common Object Services Specification, with the following exceptions:

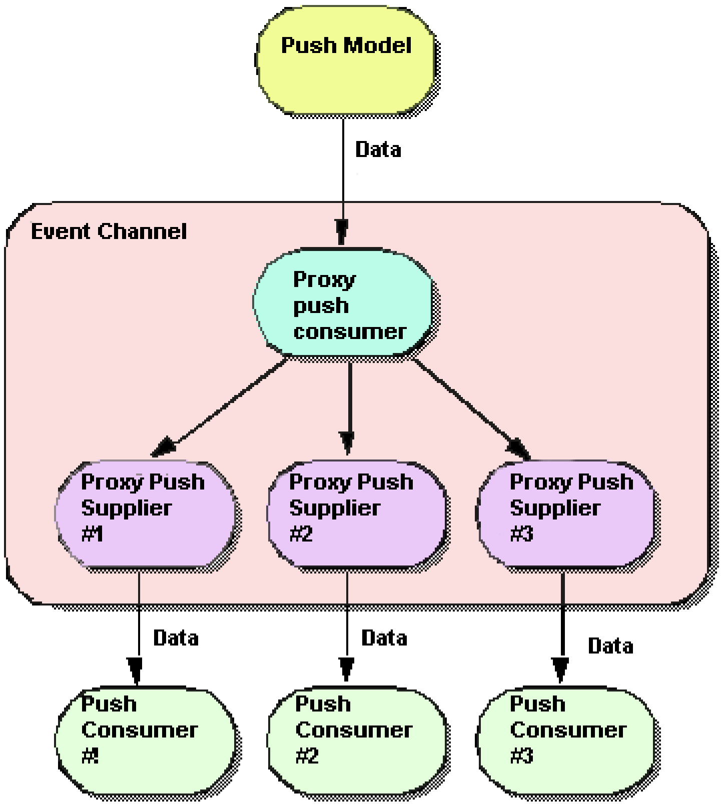
- The VisiBroker Event Service only supports generic events. There is currently no support for typed events in the VisiBroker Event Service.
- The VisiBroker Event Service offers no confirmation of the delivery of data to either the event channel or to consumer applications. TCP/IP is used to implement 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 is actually 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 16.3** Push model



**Note**

The `EventChannel` is shown above as a separate process, but it may also be implemented as part of the supplier object's process.

## 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 push consumer spends most of its time in an event loop, waiting for data to arrive from the `ProxyPushSupplier`. The `EventChannel` facilitates the transfer of data from the `ProxyPushSupplier` to the `ProxyPushConsumer`.

The figure below shows a push supplier and its corresponding `ProxyPushConsumer` object. It also shows three push consumers and their respective `ProxyPushSupplier` objects.

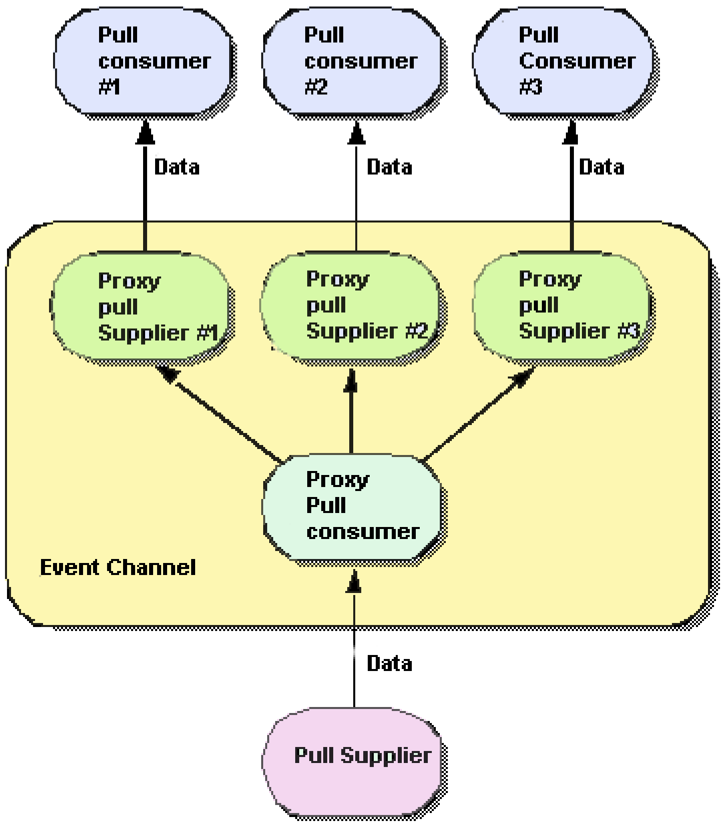
## Pull model

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 spends most of its time in an event loop waiting for data requests to be received from the `ProxyPullConsumer`. 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`.

The figure below shows a pull supplier and its corresponding `ProxyPullConsumer` object. It also shows three pull consumers and their respective `ProxyPullSupplier` objects.

**Figure 16.4** Pull model



**Note**

The event channel is shown above as a separate process, but it may also be implemented as part of the supplier object's process.

## Using event channels

To create an `EventChannel`, connect a supplier or consumer to it and use it:

**1** Create and start the EventChannel:

**Windows**

```
prompt> start vbj com.inprise.vbroker.CosEvent.EventServer -ior <iorFilename>
               <channelName>
```

**UNIX**

```
prompt> vbj com.inprise.vbroker.CosEvent.EventServer -ior <iorFilename>
               <channelName> &
```

**Note**

Only one instance of the EventChannel is supported. All binding to the `EventChannel` is done through the call to `orb.resolve_initial_references("EventService")`, where `EventService` is the hardcoded `EventChannel` name.

**2** Connect to the EventChannel.

**3** Obtain an administrative object from the channel and use it to obtain a proxy object.

**4** Connect to the proxy object.

**5** 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. The table below shows the appropriate methods for suppliers.

| Steps | Push supplier | Pull supplier |
|---|---|---|
| Bind to the EventChannel | `CosEventChannelAdmin::EventChannel::_narrow(orb::resolve_initial_references("EventService"))` | `CosEventChannelAdmin::EventChannel::_narrow(orb::resolve_initial_references("EventService"))` |
| Get a SupplierAdmin | `EventChannel::for_suppliers()` | `EventChannel::for_suppliers()` |
| Get a consumer proxy | `SupplierAdmin::obtain_push_consumer()` | `SupplierAdmin::obtain_pull_consumer()` |
| Add the supplier to the EventChannel | `ProxyPushConsumer::connect_push_supplier()` | `ProxyPullConsumer::connect_pull_supplier()` |
| Data transfer | `ProxyPushConsumer::push()` | `Implements pull() and try_pull()` |

The table below shows the appropriate methods for consumers.

| Steps | Push consumer | Pull consumer |
|---|---|---|
| Bind to the EventChannel | `CosEventChannelAdmin::EventChannel::_narrow(orb::resolve_initial_references("EventService"))` | `CosEventChannelAdmin::EventChannel::_narrow(orb::resolve_initial_references("EventService"))` |
| Get a ConsumerAdmin | `EventChannel::for_consumers()` | `EventChannel::for_consumers()` |
| Obtain a supplier proxy | `ConsumerAdmin::obtain_push_supplier()` | `ConsumerAdmin::obtain_pull_supplier()` |
| Add the consumer to the EventChannel | `ProxyPushSupplier::connect_push_consumer()` | `ProxyPushSupplier::connect_pull_consumer()` |
| Data transfer | `Implements push()` | `ProxyPushSupplier::pull() and try_pull()` |

## Creating event channels

VisiBroker provides a proprietary interface called EventChannelFactory in the CosEventChannelAdmin module to allow Event Service clients to create event channels on demand. To enable this feature, start the event service for your operating system as follows:

**Windows**

```
start vbj -Dvbroker.events.factory=true
   com.inprise.vbroker.CosEvent.EventServer <factoryName>
```

**UNIX**

```
vbj -Dvbroker.events.factory=true
   com.inprise.vbroker.CosEvent.EventServer <factoryName>
```

The property vbroker.events.factory instructs the service to create a factory object with the name <factoryName> (with a default value of VisiEvent) instead of a channel object. To write the IOR of the factory to a file, use the –ior option to provide the file name. By default, the IOR is written to the console.

The factory object created can then be bound by the client, either using the IOR written to the file (or console) or using the osagent bind mechanism to pass the factory object name. Once the factory object reference is obtained, it can be used to create, look up, or destroy event channel objects. An event channel object obtained from the factory object can be used to connect suppliers and consumers.

## Examples of push supplier and consumer

This section describes the example of the *push* supplier and the consumer applications.

### Push supplier and consumer example

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 `PushModel.C` file and the push consumer is implemented in the `PushView.C` file. These files can be found in the `<install_dir>/examples/vbroker/events` directory.

#### Deriving a PushSupplier class

The first step in implementing a supplier is to derive our own `PushModel` class from the `PushSupplier` interface, shown below.

```
module CosEventComm {
         interface PushSupplier {
            void disconnect_push_supplier();
         };
};
```

The code sample below 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.

```
// 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;
      try {
         PortableServer::ObjectId_var objId =
         PortableServer::string_to_ObjectId("PushModel");
         _myPOA->deactivate_object(objId);
      }
      catch(const CORBA::Exception& e) {
         cout << e << endl;

      }
   }
};
```

## Implementing the PushSupplier

The first portion of the supplier implementation is fairly routine. After doing some initialization, a local scope is set, resulting in a locally-scoped `PushModel` object.

```
int main(int argc, char* const* argv)
{
   try {
      // Initialize the ORB.
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

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

      CPushModel* model = NULL;
      CosEventChannelAdmin::ProxyPushConsumer_var pushConsumer = NULL;

      model = new PushModel(orb, pushConsumer, serverPOA);
      CORBA::String_var supplier_name(CORBA::string_dup("PushModel"));
      PortableServer::ObjectId_var objId =
      PortableServer::string_to_ObjectId(supplier_name);
      serverPOA->activate_object_with_id(objId, model);
      // Activate the POA Manager
      serverPOA->the_POAManager()->activate();
      CORBA::Object_var reference = serverPOA->servant_to_reference(model);
      cout << "Created model: " << reference << endl;
   }
   ...
}
```

The example uses command line options to implement the `PushSupplier`. When the command line option is `m`, it initializes and instantiates the `PushModel` object.

If the command line option is `p`, the example binds to the `EventChannel` and obtains a `SupplierAdmin` object from the `EventChannel`. Note that the application could specify an object name for a specific `EventChannel`. In a real implementation, the object could be passed as an argument to the application or obtained from the naming service

(VisiNaming), if it is available. For more information, see "Using the VisiNaming Service." Next the SupplierAdmin object is used to obtain a proxy for the pushConsumer object from the EventChannel.

If the command line option is c, the pushSupplier object is connected to the EventChannel.

```
...
if (cmd == 'p') {
   if (channel == NULL) {
      cout << "Need to locate an [e]vent channel" << endl;
   }
   else {
      pushConsumer = channel->for_suppliers()->obtain_push_consumer();
      cout << "Obtained push consumer: " << pushConsumer << endl;
      continue;
   }
}
else if (cmd == 'c' ) {
   if (model == NULL) {
      cout << "Need to create a [m]odel" << endl;
   }
   else if (pushConsumer == NULL) {
      cout << "Need to obtain a [p]ush consumer" << endl;
   }
   else {
      cout << "Connecting..." << endl;
      pushConsumer->connect_push_supplier(model->_this());
      model->start();
      continue;
   }
}
```

A different thread of the supplier application prompts the user for a string, waits for a string to be entered and converts the string to an Any object. Lastly, the data is "pushed" to the consumer proxy object.

```
...
while(true) {
   VISPortable::vsleep(_delay);
   try {
      char buf[81];
      std::string str;
      sprintf(buf, "%s%d", "Hello #", ++_counter);
      str = buf;

      CORBA::Any_var message = new CORBA::Any();
      *message <<= str.c_str();
      cout << "Supplier pushing: " << str.c_str() << endl;

      _pushConsumer->push(*message);
   }
   catch(CosEventComm::Disconnected e) {
      cout << "Disconnected #" << _counter << endl;
   }
   catch(CORBA::OBJECT_NOT_EXIST e)
   {
      cout << "Push Consumer has been disconnected" << endl;
      return;
   }
   catch(const CORBA::Exception& e) {
      cout << e << endl;
      disconnect_push_supplier();
```

```
         return;
      }
      catch(...) {
         cout << "Unexpected exception" << endl;
         disconnect_push_supplier();
         return;
      }
   }
...
```

## Complete implementation for a sample push supplier

```
#include "corba.h"
#include "CosEventComm_s.hh"
#include "CosEventChannelAdmin_c.hh"
#include "vport.h"
#include <string>

USE_STD_NS
class PushModel : public POA_CosEventComm::PushSupplier, public VISThread{
   public:
      PushModel(CORBA::ORB_ptr orb,
               CosEventComm::PushConsumer_ptr pushConsumer,
               PortableServer::POA_ptr myPOA) :
      _orb(orb), _pushConsumer(pushConsumer), _myPOA(myPOA), _counter(0),
                  _delay(1)
      {}
      void delay(int time) { delay = time; }
      void start() {
         // start the thread
         run();
      }
      void disconnect_push_supplier() {
         cout << "Model::disconnect_push_supplier()" << endl;
         try {
            PortableServer::ObjectId_var objId =
               PortableServer::string_to_ObjectId("PushModel");
            _myPOA->deactivate_object(objId);
         }
         catch(const CORBA::Exception& e) {
            cout << e << endl;
         }
      }
      // implement begin() callback
      void begin() {
         while(true) {
         VISPortable::vsleep(_delay);
         try {
            char buf[81];
            std::string str;
            sprintf(buf, "%s%d", "Hello #", ++_counter);
            str = buf;
            CORBA::Any_var message = new CORBA::Any();
            *message <<= str.c_str();
            cout << "Supplier pushing: " << str.c_str() << endl;
            _pushConsumer->push(*message);
         }
         catch(CosEventComm::Disconnected e) {
            cout << "Disconnected #" << _counter << endl;
         }
         catch(CORBA::OBJECT_NOT_EXIST e)
         {
            cout << "Push Consumer has been disconnected" << endl;
            return;
         }
         catch(const CORBA::Exception& e) {
            cout << e << endl;
            disconnect_push_supplier();
            return;
         }
         catch(...) {
            cout << "Unexpected exception" << endl;
            disconnect_push_supplier();
            return;
         }
      }
   }

private :
      int _delay;
      int _counter;
      CORBA::ORB_var _orb;
      PortableServer::POA_var _myPOA;
      CosEventComm::PushConsumer_var _pushConsumer;
};

int main(int argc, char* const* argv)
{
   try {
      // Initialize the ORB.
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

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

      // Create policies for our persistent POA
      CORBA::PolicyList policies;
      policies.length(1);
      policies[(CORBA::ULong)0] =
         rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);

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

      // Create serverPOA with the right policies
      PortableServer::POA_var serverPOA =
         rootPOA->create_POA("event_service_poa", poa_manager, policies);

      CosEventChannelAdmin::EventChannel_var channel = NULL;
      PushModel* model = NULL;
CosEventChannelAdmin::ProxyPushConsumer_var pushConsumer = NULL;

      while(true) {
         try {

            cout << "-> ";
            cout.flush();
            char cmd;
            if (cin >> cmd ) {
               if (cmd == 'e') {
                  obj = orb->resolve_initial_references("EventService");
                  channel = CosEventChannelAdmin::EventChannel::_narrow(obj);
                  cout << "Located event channel: " << channel << endl;
                  continue;
               }
               else if (cmd == 'p') {
                  if (channel == NULL) {
                     cout << "Need to locate an [e]vent channel" << endl;
                  }
                  else {
                     pushConsumer = channel->

for_suppliers()_>obtain_push_consumer();
                     cout << "Obtained push consumer: " << pushConsumer <<
endl;
                     continue;
                  }
               }
               else if (cmd == 'm') {
                  if (pushConsumer == NULL) {
                     cout << "Need to obtain a [p]ush consumer" << endl;
                  }
                  else {
                     model = new PushModel(orb, pushConsumer, serverPOA);
                     CORBA::String_var

supplier_name(CORBA::string_dup("PushModel"));
                     PortableServer::ObjectId_var objId =
                        PortableServer::string_to_ObjectId(supplier_name);
                     serverPOA->activate_object_with_id(objId, model);
                     // Activate the POA Manager
                     serverPOA->the_POAManager()->activate();
                     CORBA::Object_var reference = serverPOA->
                                          servant_to_reference(model);
                     cout << "Created model: " << reference << endl;
                     continue;
                  }
               }
               else if (cmd == 's') {
                  if (model == NULL) {
                     cout << "Need to create a [m]odel" << endl;
                  }
                  else {
                     int delay;
                     if (cin >> delay ) {
                        if (delay < 0)
                           cout << "[s]leep delay must be positive" ;
                        else
                           model->delay(delay);
                     }
                     else {
                        cerr << "Invalid argument to [s]leep" << endl;
                     }
                  }
               }
               else if (cmd == 'c' ) {
                  if (model == NULL) {
                     cout << "Need to create a [m]odel" << endl;
                  }
                  else if (pushConsumer == NULL) {
                     cout << "Need to obtain a [p]ush consumer" << endl;
                  }
                  else {
                     cout << "Connecting..." << endl;
                     pushConsumer->connect_push_supplier(model->_this());
                     model->start();
                     continue;
                  }
               }
               else if (cmd == 'd') {
                  if (pushConsumer == NULL) {
                     cout << "Need to obtain a [p]ush consumer" << endl;
                  }
               else {
                  cout << "Disconnecting..." << endl;
                  pushConsumer->disconnect_push_consumer();
                  continue;
               }
            }
            else if (cmd == 'q') {
               cout << "Quitting..." << endl;
               CORBA::ORB::shutdown();
               break;
            }
            else {
 cout << "Commands: e [e]vent channel" << endl
                  << "         s <# seconds> set [s]leep delay" << endl
                  << "         p              [p]ush consumer"   << endl
                  << "         m              [m]odel"           << endl
                  << "         c              [c]onnect"         << endl
                  << "         d              [d]isconnect"      << endl
                  << "         q              [q]uit"            << endl;
            }
         }
      }
      catch(const CORBA::SystemException& e) {
         cerr << e << endl;
      }
   }
   }
   catch(const CORBA::Exception& e) {
      cerr << e << endl;
   }
   return 0;
}
```

## Deriving a PushConsumer class

The code sample below shows the first part of the supplier application, which defines a `PushView` class that is derived from the `PushConsumer` interface, shown below.

```
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.

```
// PushView.C
#include "CosEventComm_s.hh"
#include "CosEventChannelAdmin_c.hh"
class PushView : public POA_CosEventComm::PushConsumer
{
   public:
      void push(const CORBA::Any& data) {
         cout << "Consumer being pushed: " << data << endl;
      }

      void disconnect_push_consumer() {
         cout << "PushView::disconnect_push_consumer" << endl;
      }
};
```

## Implementing the PushConsumer

If the command line is `v`, then the `PushConsumer` object is instantiated and activated. Different command line options cause it to bind to the `EventChannel`, obtain the supplier proxy object and connect to the consumer object and wait to receive push requests.

```
// PushView.C
#include "CosEventComm_s.hh"
#include "CosEventChannelAdmin_c.hh"
...
int main(int argc, char* const* argv)
{
   try {
      // Initialize the ORB.
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

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

      // Create policies for our persistent POA
      CORBA::PolicyList policies;
      policies.length(1);
      policies[(CORBA::ULong)0] =
      rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);

      PortableServer::POAManager_var poa_manager = rootPOA->the_POAManager();
      // Create serverPOA with the right policies
      PortableServer::POA_var serverPOA =
         rootPOA->create_POA("event_service_poa", poa_manager, policies);
      CosEventChannelAdmin::EventChannel_var channel = NULL;
      PushView* view = NULL;
      CosEventChannelAdmin::ProxyPushSupplier_var pushSupplier = NULL;

      while(true) {
         try {
            cout << "-> ";
            cout.flush();
            char cmd;
            if (cin >> cmd) {
               if (cmd == 'e') {
                  obj = orb->resolve_initial_references("EventService");
                  channel = CosEventChannelAdmin::EventChannel::_narrow(obj);
                  cout << "Located event channel: " << channel << endl;
                  continue;
               }
               else if (cmd == 'v') {
                  view = new PushView();
                  CORBA::String_var
consumer_name(CORBA::string_dup("PushView"));
                  PortableServer::ObjectId_var objId =
                  PortableServer::string_to_ObjectId(consumer_name);
                  serverPOA->activate_object_with_id(objId, view);
                  // Activate the POA Manager
                  serverPOA->the_POAManager()->activate();
                  CORBA::Object_var reference = serverPOA
                                                        -
>servant_to_reference(view);
                  cout << "Created view: " << reference << endl;
                  continue;
               }
               else if (cmd == 'p') {
                  if (channel == NULL) {
                     cout << "Need to locate an [e]vent channel" << endl;
                  }
                  else {
                     pushSupplier = channel->for_consumers()
                                             ->obtain_push_supplier();
                     cout << "Obtained push consumer: " << pushSupplier <<
endl;
                     continue;
                  }
               }
               else if (cmd == 'c' ) {
                  if (view == NULL) {
                     cout << "Need to create a [v]iew" << endl;
                  }
                  else if (pushSupplier == NULL) {
                     cout << "Need to obtain a [p]ush supplier" << endl;
                  }
                  else {
                     cout << "Connecting..." << endl;
                     pushSupplier->connect_push_consumer(view->_this());
                     continue;
                  }
               }
               else if (cmd == 'd') {
                  if (pushSupplier == NULL) {
                     cout << "Need to obtain a [p]ush supplier" << endl;
                  }
```

```
                    else {
                       cout << "Disconnecting..." << endl;
                       pushSupplier->disconnect_push_supplier();
                       continue;
                    }
                 }
                 else if (cmd == 'q') {
                    cout << "Quitting..." << endl;
                    break;
                 }
                  cout << "Commands: e             [e]vent channel"    << endl
                       << "          p             [p]ush supplier"    << endl
                       << "          v             [v]iew"             << endl
                       << "          c             [c]onnect"          << endl
                       << "          d             [d]isconnect"       << endl
                       << "          q             [q]uit"            << endl;
              }
           }
           catch(const CORBA::SystemException& e) {
              cerr << e << endl;
           }
        }
     }
     catch(const CORBA::Exception& e) {
        cerr << e << endl;
     }
  }
```

## 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 code sample below shows how to limit each consumer to 15 outstanding messages.

```
CosEventChannel -maxQueueLength=15 MyChannel
```

**Note**

If `maxQueueLength` is not specified or if an invalid number is specified, a default queue length of `100` is used.

## Compiling and linking programs

Applications that use the Event Service need to include the following generated files:

```
#include "CosEventComm_s.hh"
#include "CosEventChannelAdmin_c.hh"
```

### UNIX

UNIX applications need to be linked with one of the libraries:

- `libcosev.a`
- `libcosev.so`

### Windows

Windows applications need to be linked with the `cosev_r.lib (cosev_r.dll)` library.

17

# Using the VisiBroker Server Manager

The VisiBroker Server Manager allows client applications to monitor and manage object servers, view and set properties at runtime for those servers, and view and invoke methods on Server Manager objects. The Server Manager uses elements known as *containers* which represent each major ORB component. A container can contain properties, operations, and even other containers.

**Note**

Do not confuse the Server Manager container with J2EE containers. The Server Manager container is simply a logical grouping of ORB components and selected runtime properties.

## Getting Started with the Server Manager

This section covers enabling the Server Manager on a server, obtaining a Server Manager reference, working with containers, the Storage interface and the Server Manager IDL.

### Enabling the Server Manager on a server

A VisiBroker server is not enabled to be managed by default. The command which starts the server must set the following property to manage the server:

```
vbroker.orb.enableServerManager=true
```

The property can be specified either through the command-line or through the server's properties file.

## Obtaining a Server Manager reference

The first step in interacting with a Server Manager is to obtain a reference to a server's Server Manager. This reference points to the top level container. A client can obtain the reference in two ways:

1 A server runner can choose to name the Server Manager using the property option `vbroker.serverManager.name`. For example, the command:

    ```
    prompt> Server -Dvbroker.serverManager.name=BigBadBoss
    ```

    registers the Server Manager name “BigBadBoss” to the Smart Agent namespace. From this point onward, the client can bind to that name and start invoking operations on the reference. This property can be set in the properties file as well. This method of locating a Server Manager can be used when the client does not have object references to any other objects implemented by the server, for example:

    ```
    ServerManager::Container_var cont =
    ServerManager::Container::_bind("BigBadBoss");
    ```

2 If the client has an object reference to some other object implemented by the server, then the client can perform `_resolve_reference("ServerManager")` on that object to obtain the ServerManager for the ORB corresponding to the object. The following code fragment obtains the Server Manager's top-level container from the `Bank::AccountManager` object.

    ```
    Bank::AccountManager_var manager = Bank::AccountManager::_bind("/
    bank_agent_poa", managerId);
    ServerManager::Container_var cont;
    CORBA::Object_var objCont = manager->_resolve_reference("ServerManager");
    ```

    The client code needs to include the `servermgr_c.hh` to use the Server Manager interfaces.

## Working with Containers

Once a client application has obtained the reference to the top level container, it can:

– get, set, or add properties on top level container.

– iterate through containers container inside top level container.

– get, set, or add containers.

– invoke operations defined in containers.

– get or set storage on the containers.

– restore or persist properties to property storage.

The top-level container does not support any properties or operations but just contains the ORB container. The ORB container in turn contains few ORB properties, a `shutdown` method, and other containers like RootPOA, Agent, OAD, and so forth.

See “The Container Interface” for information on how to interact with containers. The “Server Manager examples” shows Java and C++ interactions as well.

### The Storage Interface

Server Manager provides an abstract notion of storage that can be implemented in any fashion. Individual containers may choose to store their properties in the different ways. Some containers may choose to store their properties in a database, while others may choose to store them in files or in some other method. The `Storage` interface is defined in Server Manager IDL.

Every container uses the same methods to get and set storage, along with the ability to optionally set storage on all child containers of the parent. Similarly, each container uses the same methods to read and write its properties from the storage.

For information on the Storage Interface and its methods, see "The Storage Interface".

## The Container Interface

The container interface defines an interface and associated methods for logically grouping sets of objects, properties, operations, and so forth.

### Container Methods

A container can hold properties, operations, and other containers. Each major ORB component is represented as a container. The top-level container corresponds to the ORB itself and includes a few ORB properties, the `shutdown` method, and a few other commonly used containers like `RootPOA` and `Agent`.

This section explains the C++ methods that can be executed on the container interface. There are four categories:

- Methods related to property manipulation and quereies
- Methods related to operations
- Methods related to children containers
- Methods related to storage

#### Methods related to property manipulation and queries

```
CORBA::StringSequence list_all_properties();
```

Returns the names of all the properties in the contianer as a `StringSequence`.

```
PropertySequence get_all_properties();
```

Returns the `PropertySequence` containing the names, values, and read-write status of all the properties in the container.

```
Property get_property(in string name raises(NameInvalid);
```

Returns the value of the property **name** passed as an input parameter.

```
void add_property(in string name, in any value) raises(NameInvalid,
ValueInvalid, ValueNotSettable);
```

Sets the value of the property **name** to the requested **value**.

```
void persist_properties(in boolean recurse) raises(StorageException);
```

Causes the container to actually store its properties to the associated storage. If no storage is associated with the container, a `StorageException` will be raised. When it is invoked with the parameter `recurse=true`, the properties of the children containers are also stored into the storage. It is up to the container to decide if it has to store all the properties or only the changed properties.

```
void restore_properties(in boolean recurse) raises(StorageException);
```

Instructs the container to obtain its properties from the storage. A container knows exactly what properties is manages and it attempts to read those properties from the storage. The containers shipped with the ORB do not support restoring from the storage. You must create containers that support this feature yourself.

### Methods related to operations

```
::CORBA::StringSeqence list_all_operations();
```

Returns the names of all the operations suppored in the container.

```
OperationSequence get_all_operations();
```

Returns all the operations along with the parameters and the type code of the parameters so that the operation can be invoked with the appropriate parameters.

```
Operation get_operation(in string name) raises(NameInvalid);
```

Returns the parameter information of the operation specified by **name** which can be used to invoke the operation.

```
any do_operation(in Operation op) raises(NameInvalid, ValueInvalid,
OperationFailed);
```

Invokes the method in the operation and returns the result.

### Methods related to children containers

```
::CORBA::StringSequence list_all_containers();
```

Returns the names of all the children containers of the current container.

```
NamedContainerSequence get_all_containers();
```

Returns all the children containers.

```
NamedContainer get_container(in string name) raises(NameInvalid);
```

Returns the child container identified by the **name** parameter. If there is not any child container with this name, a `NameInvalid` exception is raised.

```
void add_container(in NamedContainer container) raises(NameAlreadyPresent,
ValueInvalid);
```

Adds the container as a child container of this **container**.

```
void set_container(in string name, in Container value)  raises(NameInvalid,
ValueInvalid, ValueNotSettable);
```

Modifies the child container identified by the **name** parameter to one in the **value** parameter.

### Methods related to storage

```
void set_storage(in Storage s, in boolean recurse);
```

Sets the storage of this container. If `recurse=true`, it also sets the storage for all its children as well.

```
Storage get_storage();
```

Returns the current storage of the container.

## The Storage Interface

The Server Manager provides an abstract notion of *storage* that can be implemented in any fashion. Individual containers may choose to store their properties in databases, flat files, or some other means. The storage implementation included with the VisiBroker ORB uses a flat-file-based approach.

### Storage Interface Methods

```
void open() raises (StorageException);
```

Opens the storage and makes it ready for reading and writing the properties. For the database-based implementation, logging into the database is performed in this method.

```
void close() raises (StorageException);
```

Closes the storage. This method also updates the storage with any properties that have been changed since the last `Container::persist_properties` call. In database implementations, this method closes the database connection.

```
Container::PropertySequence read_properties() raises(StorageException);
```

Reads all the properties from the storage.

```
Container::Property read_property(in string propertyName)
raises(StorageException, Container::NameInvalid);
```

Returns the property value for **propertyName** read from the storage.

```
void write_properties(in Container::PropertySequence p)
raises(StorageException);
```

Saves the property sequence into the storage.

```
void write_property(in Container::Property p) raises(StorageException);
```

Saves the single property into the storage.

## Limiting access to the Server Manager

A client that obtains the Server Manager can control the entire ORB and hence security is paramount. The following properties can limit a user's access to the Server Manager functionality:

| Property | Default Value | Description |
|---|---|---|
| `vbroker.orb.enableServerManager` | `false` | Setting this property to `True` enables the Server Manager. |
| `vbroker.serverManager.enableOperations` | `true` | Controls the permission to invoke operations in the containers. If set to `false`, the client will not be able to invoke `do_operation` on any container. |
| `vbroker.serverManager.enableSetProperty` | `true` | Controls the setting of properties from the client. If set to `false`, clients cannot modify any of the container properties. |

## Server Manager IDL

Server Manager IDL is as shown below:

```
module ServerManager {
interface Storage;

exception StorageException {
    string reason;
};

interface Container
{
    enum RWStatus {
        READWRITE_ALL,
        READONLY_IN_SESSION,
        READONLY_ALL
    };

    struct Property {
        string name;
        any value;
        RWStatus rw_status;
    };
    typedef sequence<Property> PropertySequence;

    struct NamedContainer {
        string name;
        Container value;
        boolean is_replaceable;
    };
    typedef sequence<NamedContainer> NamedContainerSequence;

    struct Parameter {
        string name;
        any value;
    };
    typedef sequence<Parameter> ParameterSequence;

    struct Operation {
        string name;
        ParameterSequence params;
        ::CORBA::TypeCode result;
    };
    typedef sequence<Operation> OperationSequence;

    struct VersionInfo {
        unsigned long major;
        unsigned long minor;
    };

    exception NameInvalid{};
    exception NameAlreadyPresent{};
    exception ValueInvalid{};
    exception ValueNotSettable{};
    exception OperationFailed{
        string real_exception_reason;
    };
    ::CORBA::StringSequence list_all_properties();
    PropertySequence get_all_properties();
    Property get_property(in string name) raises (NameInvalid);
    void add_property(in Property prop)
    raises(NameAlreadyPresent, NameInvalid, ValueInvalid);
    void set_property(in string name, in any value)
    raises(NameInvalid, ValueInvalid, ValueNotSettable);

    ::CORBA::StringSequence get_value_chain(in string propertyName) raises
(NameInvalid);
    void persist_properties(in boolean recurse) raises (StorageException);
    void restore_properties(in boolean recurse) raises (StorageException);

    ::CORBA::StringSequence list_all_operations();
    OperationSequence get_all_operations();
    Operation get_operation(in string name)
    raises (NameInvalid);
    any do_operation(in Operation op)
    raises(NameInvalid, ValueInvalid, OperationFailed);

    ::CORBA::StringSequence list_all_containers();
    NamedContainerSequence get_all_containers();
    NamedContainer get_container(in string name)
    raises (NameInvalid);
    void add_container(in NamedContainer container)
    raises(NameAlreadyPresent, ValueInvalid);
    void set_container(in string name, in Container value)
    raises(NameInvalid, ValueInvalid, ValueNotSettable);


    void set_storage(in Storage s, in boolean recurse);
    Storage get_storage();

    readonly attribute VersionInfo version;
};

interface Storage
{
    void open() raises (StorageException);
    void close() raises (StorageException);
    Container::PropertySequence read_properties() raises
    (StorageException);
    Container::Property read_property(in string propertyName)
    raises (StorageException, Container::NameInvalid);
    void write_properties(in Container::PropertySequence p) raises
    (StorageException);
    void write_property(in Container::Property p) raises (StorageException);
};

};
```

## Server Manager examples

The following examples demonstrate how to:

1 Obtain a reference to the top-level container.

2 Get all containers and their properties recursively.

3 Getting, setting, and saving properties on different containers.

4 Invoke the `shutdown()` method on the ORB container.

These example files can be found in:

```
<install_dir>/examples/vbroker/ServerManager/
```

The following example uses the `bank_agent` server. This server should be started by passing the property storage file. Initially the property file contains the properties to enable the Server Manager and set its name. The file is used by the Server Manager to update the properties if the user changes them. The properties to enable the Server Manager and set its name can be passed as command-line options, but the property file is required if any of the properties are to be modified and saved during the session.

Initially, the property file contains the following:

```
# server properties
vbroker.orb.enableServerManager=true
vbroker.serverManager.name=BigBadBoss
```

The server is started from the command-line:

```
prompt> Server -ORBpropStorage prop.txt
```

### Obtaining the reference to the top-level container

This example uses the second, or `bind` method since the Server Manager has been started with a name (see "Obtaining a Server Manager reference").

```
ServerManager::Container_var cont =
ServerManager::Container::_bind("BigBadBoss");
```

### Getting all the containers and their properties

The following example shows how `get_all_properties`, `get_all_operations`, and `get_all_containers` can be used to query all the properties and operations of all the containers below the current container recursively.

```
void SrvrmgrUtil::displayContainer(char * name, ServerManager::Container_ptr
cont) {
  try {
        ServerManager::Container::PropertySequence * props = cont-
>get_all_properties();
        for (int i =0; i < props->length() ; i++) {
                printProperty((*props)[i]);
        }
        ServerManager::Container::OperationSequence * ops = cont-
>get_all_operations();
        for (int j = 0; j < ops->length(); j++)
                printOperation((*ops)[j]);
        }
        catch (ServerManager::Container::NameInvalid& ne) {
        cerr << ne <<endl;


        } catch (ServerManager::Container::ValueInvalid & ve) {
                cerr << ve <<endl;
        } // Pass the remaining exceptions to the main function
        ServerManager::Container::NamedContainerSequence* nc = cont-
>get_all_containers();
        for (int j =0 ; j < nc->length(); j++) {
                displayContainer((*nc)[j].name, (*nc)[j].value);
        }
}
```

## Getting and Setting properties and saving them into the file

The following code fragment shows how to query a property of a container. If the container is not the top-level container, it needs to be reached first by traversing through all its parents from the top container. The get and set methods can be called only on the container which owns the property.

**Note**

Properties with `RW_STATUS` values of `READONLY_ALL` are not settable.

```
// querying for properties
ServerManager::Container::NamedContainer_var orbCont = cont-
>get_container("ORB");
ServerManager::Container::NamedContainer_var sesCont =
                orbCont->value->get_container("ServerEngines");
ServerManager::Container::NamedContainer_var seCont =
                sesCont->value->get_container("iiop_tp");
ServerManager::Container::NamedContainer_var scmCont =
                seCont->value->get_container("iiop_tp");
SrvrmgrUtil::displayProperty("vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.inUseTh
reads",
                scmCont->value);

CORBA::Any_var a = new CORBA::Any;
a <<= (CORBA::ULong) 34001UL;
scmCont->value->set_property("vbroker.se.iiop_tp.scm.iiop_tp.listener.port",
a);
scmCont->value->persist_properties(true);
```

## Invoking an operation in a Container

The `ORB` container supports the operation `shutdown`. The operation can be obtained by calling `get_operation` on the container.

```
void SrvrmgrUtil::invokeShutdown(ServerManager::Container_ptr orbCont)
{
        ServerManager::Container::Operation_var shutOp = orbCont-
>get_operation("shutdown");
        shutOp->params[0].value <<= CORBA::Any::from_boolean(0UL);
        orbCont->do_operation(shutOp.in());
}
```

The `operation` returned by the `get_operation` call has the default parameters. If the default values of the parameters are not the intended ones, these values should be modified before calling the `do_operation` method.

## Custom Containers

It is possible for a user application to define containers and add them to the Server Manager. The container manages two properties and defines one operation. It also uses its own storage for storing the properties. The two properties are:

| Property | Description |
|---|---|
| `manager.lockAllAccounts` | This property has a read-write status of `READWRITE_ALL`, so it can be modified and takes effect while the server is running. The purpose of this property is to make AccountManager unavailable for client applications. The initial value of this property is read by the server on startup and saved to the same file when server shuts down/restarts. |
| `manager.numAccounts` | This property has a read-write status of `READONLY_ALL`, so it can only be read. The purpose of this property is to provide the number of Accounts in the AccountManager. The value of this property is not written to the storage. |

The operation is:

| Operation | Description |
|---|---|
| shutdown | Shuts down the server without starting it again. Before shutdown, the `manager.lockAllAccounts` property is written (persisted) to the property file. |

For a complete example, go to:

<install_dir>/examples/vbroker/ServerManager/custom_container/

The main steps in writing custom containers is follows:

1. Implement the Container interface defined in Serve Manager IDL.
2. Instantiate the servant that implements the Container interface and activate it on a POA.
3. Obtain the reference to Server Manager top level container. Add the custom container to the Container hierarchy.

The server then can be started with the Server Manager enabled and a client can interact with the custom container.

If you want your application to implement its own storage, it has to implement the `Storage` interface defined in Server Manager IDL. The basic steps are same as implementing custom containers

18

# Using VisiBroker Native Messaging

## Introduction

Native Messaging is a language independent, portable, interoperable, server side transparent, and OMG compliant two-phase invocation framework for CORBA and RMI/J2EE (RMI-over-IIOP) applications.

### Two-phase invocation (2PI)

In object-oriented vocabulary, invocations are method calls made on target objects. Conceptually, an invocation is made up of two communication phases:

- sending a request to a target in the first phase
- receiving a reply from the target in the second phase

In classic object-oriented distributed frameworks, such as CORBA, RMI/J2EE, and .NET, invocations on objects are *one-phased* (1PI), in which the sending and receiving phases are encapsulated together inside a single operation rather than exposed individually. In a one-phased invocation the client calling thread blocks on the operation after the first phase until the second phase completes or aborts.

If a client can be unblocked after the first phase, and the second phase can be carried out separately, the invocation is called *two-phased* (2PI). The operation unblocking before completing its two invocation phases is called a *premature return* (PR) in Native Messaging.

A 2PI allows a client application to unblock immediately after the request sending phase. Consequently, the client does not have to halt its calling thread and retain the transport connection while waiting for a reply. The reply can be retrieved or received by the client from an independent client execution context and/or through a different transport connection.

### Polling-Pulling and Callback models

In a two-phase invocation scenario, after sending out each request the client application can either actively poll and pull the reply using a poller object provided by the infrastructure, or the client can passively wait for the infrastructure to notify it and send back the reply on a specified asynchronous callback handler. These two scenarios are usually called the synchronous *polling-pulling* model and the asynchronous *callback* model respectively.

## Non-native messaging and IDL mangling

In non-native messaging, such as CORBA Messaging, two-phase invocations are not made with native operation signatures on native IDL or RMI interfaces. Instead, at different invocation phases, and with different reply retrieve models, client applications have to call various mangled operations.

For instance, in CORBA Messaging, to make a two-phase invocation of operation `foo (<parameter_list>)` on a target, the request sending is not made with the native signature `foo()` itself, but it is made with either of the following mangled signatures:

```
// in polling-pulling model
sendp_foo(<input_parameter_list>);

// in callback model
sendc_foo(<callback_handler>, <input_parameter_list>);
```

The reply polling operation signature is:

```
foo(<timeout>, <return_and_output_parameter_list_as_output>);
```

The reply delivery callback operation signature is:

```
foo(<return_and_output_parameter_list_reversed_as_input>);
```

These mangled operations are either additional signatures added to the original application specified interface, or defined in additional type specific interfaces or valuetypes.

Problems of this non-native and mangling approach are:

- It ruins the intuitiveness of the original IDL interface and operation signatures.
- It could conflict with other operation mangling, for instance, in case of Java RMI.
- It could collide with operation signatures already used by the original IDL interface.
- It introduces interface binary incompatibility. For instance, interfaces with and without mangled signatures are not necessarily binary compatible in their language mapping.
- It does not respect the natural mapping between IDL operations and native GIOP messages, and therefore, introduces inconsistency and dilemmas when used with other OMG CORBA features, such as PortableInterceptor.

## Native Messaging solution

Native Messaging only uses *native* IDL language mapping and *native* RMI interfaces defined by applications, without any interface mangling and without introducing any additional application specific interface or valuetype.

For instance, in Native Messaging, sending a request to `foo(<parameter_list>)` and retrieving (or receiving) its reply in either the polling-pulling or callback models are made with the exact native operation `foo(<parameter_list>)` itself and are made on native IDL or RMI interfaces. No mangled operation signature and interfaces or valuetypes are introduced or used.

This pure native and non-mangling approach is not only elegant and intuitive but completely eliminates conflicts, name collision, and inconsistencies of operation signature mangling.

## Request Agent

Similar to the OMG Security and Transaction Services, Native Messaging is an object service level solution, which is based on an fully interoperable broker server, the *Request Agent*, and a client side portable request interceptor fully compliant with the OMG Portable Interceptor specification.

When making two-phase invocations, Native Messaging applications do not send requests directly to their target objects. Instead, request invocations are made on delegate *request proxies* created on a specified Request Agent. The request proxy is responsible for delegating invocations to their specified target objects, and delivering replies to client callback handlers or returning them later on client polling-pulling.

Therefore, a request agent needs to be known by client applications. Usually, this is accomplished by initializing the client ORB using OMG standardized ORB initialization command arguments:

```
-ORBInitRef  RequestAgent=<request_agent_ior_or_url>
```

This command allows client applications to resolve the request agent reference from this ORB as an initial service, for instance:

```
// Getting Request Agent reference in C++
CORBA::Object_var ref
    = orb->resolve_initial_references("RequestAgent");
NativeMessaging::RequestAgentEx_var agent
    = NativeMessaging::RequestAgentEx::_narrow(ref);
```

By default, the URL of a request agent is:

```
corbaloc::<host>:<port>/RequestAgent
```

Here, <host> is the host name or dotted IP address of a RequestAgent server, and <port> is the TCP listener port number of this server. By default, NativeMessaging RequestAgent uses port 5555.

### Native Messaging Current

Similar to the OMG Security and Transaction Services, Native Messaging uses a thread local Current object to provide and access additional supplemental parameters for making two-phase invocations. These parameters include blocking timeout, request tag, cookie, poller reference, reply availability flag, and others. Semantic definitions and usage descriptions of these parameters are given in later sections. Similarly, the Native Messaging Current object reference can be resolved from an ORB as an initial service, for instance:

```
// Getting Current object reference in C++
CORBA::Object_var ref
    = orb->resolve_initial_references("NativeMessagingCurrent");
NativeMessaging::Current_var current
    = NativeMessaging::Current::_narrow(ref);
```

### Core operations

A two-phase framework allows all normal invocations to be carried out in two separate phases manageable by client applications. Nevertheless, on fulfilling or using this two-phase invocation service, the framework and/or client may need some other primitive core functions from the framework. Operations used to access primitive core functions are called *core operations*. It is desirable that:

- Core operations are always accomplished in a single phase. An invocation on a core operation always blocks until it completes or aborts.
- Core operations are always orthogonal to any normal two-phase invocations that they are involved in.

In Native Messaging, all pseudo operations are reserved as core operations.

**Note**

In this document, if not explicitly stated, “invocation” or “operation” implies a non-core two way operation.

## StockManager example

The StockManager example is used in this section to illustrate the Native Messaging usage scenarios. This example is abridged from the full scale version that is shipped with the product in the `<install_dir>/examples/vbroker/NativeMessaging/stock_manager` directory, and it is provided to illustrate functionality that is equivalent to the CORBA Messaging StockManager example.

The following example assumes a server object has its IDL interface, StockManager, defined as follows:

```
// from: <install_dir>/examples/vbroker/NativeMessaging/
//       stock_manager/StockManager.idl
interface StockManager {
    boolean add_stock(in string symbol, in float price);
    boolean find_closest_symbol(inout string symbol);
};
```

A conventional single-phase `add_stock()` or `find_closest_symbol()` call adds a stock symbol to or finds a symbol in the targeted stock manager server. The following is an example of the invocation code:

```
// invoke and block until return
CORBA::Boolean stock_added
    = stock_manager->add_stock("ACME", 100.5);
CORBA::String_var symbol = (const char*)"ACMA";
CORBA::Boolean closest_found
    = stock_manager->find_closest_symbol(symbol.inout());
```

In the above one-phase invocation case, the invocations are blocked until the client receives its returns or exceptions.

Using Native Messaging, two-phase invocations can be made on the same stock manager server. Replies to these invocations can be retrieved or returned using the synchronous polling-pulling model or the asynchronous callback model, as illustrated in the following sections, “Polling-pulling model”, and “Callback model”.

**Note**

This document illustrates the StockManager example code in C++. The corresponding Java code is available in the “Using VisiBroker Native Messaging” chapter of the *VisiBroker for Java Developer's Guide*.

## Polling-pulling model

In the polling-pulling model, the result of a two-phase invocation is pulled back by client applications. The steps for Native Messaging polling-pulling two-phase invocations are summarized below.

1 Create a request proxy from a Native Messaging Request Agent. This proxy is created for a specific target object (a stock manager server in our example) and is used to delegate requests to the target.

2 Get the *typed receiver* or *<I> interface* of this proxy. This typed receiver is used by the client application to send requests to the proxy. The typed receiver of a proxy supports the same IDL interface as the target object. In this example, the typed receiver supports the StockManager interface and can be narrowed down to a typed StockManager stub.

3 Perform the first invocation phase, making several invocations on the typed receiver stub. By default, invocations on a typed receiver are returned with dummy output and return values. This is called a *premature return*. Receiving a premature return from proxy's typed receiver without raising an exception indicates that a two-phase invocation has been successfully initiated. It indicates that the request has been accepted and assigned to a distinct poller object by the request agent. The poller object of a two-phase invocation is available from the local NativeMessaging Current. Like the typed receiver, all poller objects also support the same IDL interface as the target (in this example the StockManager).

4 Carry out the second phase of the invocation, polling availability and pulling replies back from the poller objects. The client application narrows the poller objects to their

corresponding typed receiver stubs (StockManager in this example) and invokes the same operations as those invoked in the request sending phase. When making an invocation on poller objects input parameters are ignored. Also, the agent does not deliver new requests to the delegated target object. The agent treats all invocations made on the poller object as polling-pulling requests. Usually, a timeout value can be provided as a supplemental parameter through NativeMessaging Current to specify the maximum polling blocking timeout. If the reply is available before the timeout, the polling invocation will receive a *mature return* with output parameters and a return result from the real invocation. Otherwise, if the reply is not available before the timeout expires, the poll ends up with a premature return again. Applications should use the `reply_not_available` attribute of Native Messaging Current to determine whether a polling return is premature.

The following code sample illustrates how to use Native Messaging to make polling-pulling two-phase invocations on a stock manager object:

```
// from: <install_dir>/examples/vbroker/NativeMessaging/
//       stock_manager/polling_client_main.C

// 1. create a request proxy from the request agent for making
//    non-blocking requests on targeted stock_manager server.
NativeMessaging::RequestProxy_var proxy
    = agent->create_request_proxy(
        stock_manager, "",
        NULL, NativeMessaging::PropertySeq(0));

// 2. Get the request (typed) receiver of the proxy
CORBA::Object_var ref;
StockManager_var stock_manager_rcv
    = StockManager::_narrow(ref = proxy->the_receiver());

// 3. send several requests to the typed receiver, and
//    get their reply pollers from Native Messaging Current.
StockManager_var pollers[2];
stock_manager_rcv->add_stock("ACME", 100.5);
pollers[0] = StockManager::_narrow(ref = current->the_poller());
CORBA::String_var symbol = (const char*)"ACMA";
stock_manager_rcv->find_closest_symbol(symbol.inout());
pollers[1] = StockManager::_narrow(ref = current->the_poller());

// 4. Poll/pull the two associated replies
current->wait_timeout(max_timeout);

CORBA::Boolean stock_added;
do { stock_added = pollers[0]->add_stock("", 0.0); }
while(current->reply_not_available());

CORBA::Boolean closest_found;
do { closest_found = pollers[1]->find_closest_symbol(symbol.inout()); }
while(current->reply_not_available());
```

**Note**

- In Native Messaging, the request sending phase and the reply polling-pulling phase of a two-phase invocation all use the same operation signature. This operation used by both phases of a two-phase invocation is exactly the same *native* operation defined on the actual target's IDL interface.
- Poller objects are normal CORBA objects with location transparency. Therefore, in Native Messaging, it is not necessary to carry out the request sending phase and the reply polling phase of a two-phase invocation in same client execution context and through same transport connection.

- If there is an exception in polling-pulling phase, the application should use the Current `reply_not_available` attribute to determine whether the exception is the result of a reply polling-pulling failure, or the successful pulling of a real exceptional result of the delegated request. `TRUE` indicates that the exception is a polling-pulling failure between the client and agent. `FALSE` indicates that the exception is the real result of the delegated request.
- In a premature return, Native Messaging sets all non-primitive output parameters and the return value to null. This is similar to the OMG non-exception handling C++ mapping except Native Messaging uses a local Current object rather than the CORBA Environment.

Additional features, variances of the polling-pulling model, and Native Messaging API syntax and semantics specification are discussed in "Advanced Topics" and "Native Messaging API Specification".

## Callback model

Using the Native Messaging callback model, applications are unblocked immediately after they send out requests to a proxy's typed receiver. Replies to these invocations are delivered to a callback reply recipient that is specified upon creating the request proxy.

The steps to make Native Messaging two-phase invocations in the callback model are summarized below:

1. Create a request proxy from a Native Messaging Request Agent. This proxy is created for a specific target object. Like the polling-pulling model, this proxy will be used to delegate requests to the specified target. A reply recipient callback handler, which is a null reference in the polling-pulling model, is also specified on creating this request proxy. The request agent will deliver to the callback handler any newly available replies to requests delegated by this proxy.
2. Like the second step in the polling-pulling model, get the *typed receiver*, or *<I> interface*, of this proxy and narrow it down to a typed <I> stub (a StockManager stub in this example).
3. Like the third step in the polling-pulling model, perform the first invocation phase by making several invocations on the proxy's typed receiver stub. By default, invocations on a typed receiver are returned with dummy output and return values. This is called a *premature return*. A premature return on a proxy's typed receiver without an exception indicates a two-phase invocation has been successfully initiated.
4. Complete the second phase of the invocation, which is to receive replies. In the callback model, this is done asynchronously in a completely independent execution context. Client applications implement and activate a reply recipient object. This callback object is type unspecific, that is it does not depend on the real target's IDL interface. The key operation of this callback handler is the `reply_available()` method which is discussed below after the code sample.

The following code sample illustrates the first three steps for using Native Messaging to make callback model two-phase invocations on a stock manager object:

```
// from: <install_dir>/examples/vbroker/NativeMessaging/
//       stock_manager/callback_client_main.C

// get type independent callback handler reference
NativeMessaging::ReplyRecipient reply_recipient = …;

// 1. create a request proxy from the request agent for
//    making asynchronous requests on targeted stock_manager server.
NativeMessaging::RequestProxy_var proxy
    = agent->create_request_proxy(
        stock_manager, "", reply_recipient,
        NativeMessaging::PropertySeq(0));

// 2. Get the request (typed) receiver of the proxy
StockManager_var stock_manager_rcv
    = StockManager::_narrow(obj = proxy->the_receiver());

// 3. send two requests to the receiver
stock_manager_rcv->add_stock("ACME", 100.5);
CORBA::String_var symbol = (const char*)"ACMA";
stock_manager_rcv->find_closest_symbol(symbol.inout());
```

Here, the `reply_recipient` callback handler is a `NativeMessaging::ReplyRecipient` object regardless the specific application target types. The `ReplyRecipient` interface is defined as

```
// from: <install_dir>/idl/NativeMessaging.idl

interface NativeMessaging::ReplyRecipient {
    void reply_available(
        in object reply_holder,
        in string operation,
        in sequence<octet> the_cookie);
    );
};
```

The `reply_holder` parameter of `reply_available()` is called a *reflective callback* reference, which is the same as a reply poller object of the polling-pulling model and can be used by the `reply_available()` implementation to pull back the reply result in the same way a polling-pulling client would pull back a reply result from a poller object.

**Note**

In delivering replies to a callback handler, Native Messaging uses the *double dispatch* pattern to *reverse* the callback model into a polling-pulling model. Here, a reply recipient implementation makes a second (reflective) callback on a typed `reply_holder` reference to retrieve the reply.

The following code sample is an example implementation of `reply_available()` method:

```
// from: <install_dir>/examples/vbroker/NativeMessaging/
//        stock_manager/AsyncStockRecipient.C

void StockManagerReplyRecipientImpl::reply_available(
CORBA::Object_ptr reply_holder,
    const char* operation,
    const CORBA::OctetSequence& cookie)
{
    StockManager_var poller
        = StockManager::_narrow(reply_poller);

    // retrieve response using reflective callback
    if( strcmp(operation, "add_stock") == 0 ) {
        // retrieve a add_stock() return.
        CORBA::Boolean stock_added
            = poller->add_stock("", 0.0);
        ...
    }
    else
    if( strcmp(operation, "find_closest_symbol") == 0 ) {
        CORBA::String_var symbol = (const char*)"";
        // retrieve a find_closest_symbol() return
        CORBA::Boolean closest_found
            = poller->find_closest_symbol(symbol.inout());
        ...
        }
    ...
    }
```

**Note**

- In Native Messaging, the request sending phase and the reply receiving phase of a two-phase invocation both use the same operation. The operation used by both phases of a two-phase invocation is exactly the same *native* operation defined on the actual target's IDL interface.
- Reply recipient objects are normal CORBA objects and are location transparent. Therefore, in Native Messaging, the reply recipient callback object is not necessarily located within the request sending client process.
- If an exception is raised when the `reply_available()` implementation retrieves a reply from the `reply_holder`, the application should use the Current `reply_not_available` attribute to determine whether the exception reports retrieving a failure or a successful reply retrieval of a real exceptional result of the delegated request. `TRUE` indicates that this exception is the result of a reply retrieval failure between the client and agent. `FALSE` indicates that this exception is a real result of delegated request.
- Reply retrieval operations on `reply_holder` should only be made within the scope of the `reply_available()` method. Once the application returns from `reply_available()`, the `reply_holder` may no longer be valid.

Additional features, variances of the polling-pulling model, and the Native Messaging API specification are discussed in “Advanced Topics” and “Native Messaging API Specification”.

## Advanced Topics

### Group polling

As illustrated in previous sections, multiple requests can be delegated by a given request proxy. However, as different requests take different processing time, replies from them are not necessarily ready in the order in which they were invoked. Instead of polling individual requests one by one, group polling allows a polling client application, which has multiple requests delegated by a given request proxy, to determine the availability of replies in an multiplexed aggregation.

In order to participate in group polling, a request sent to a given proxy needs to be tagged. Request tags are assigned by clients to identify requests in the scope of their group, namely the request proxy. Native Messaging does not impose any constraint on request tag content, except that they must be unique within the scope (request proxy). Untagged requests (requests with empty tags) do not participate in group polling, and the availability of their replies is not reported by group polling results.

The steps for using group polling are summarized below.

1. Send tagged requests. To tag a request, a client application simply sets the `request_tag` attribute of the local Native Messaging Current object before making each invocation on the typed receiver interface (before delivering each request). The content of each request tag is specified by application for its own convenience, as long as it is unique within its scope (proxy).
2. Poll reply availability on the request proxy, instead of on any individual poller, by calling the proxy's `poll(max_timeout, unmask)` operation. This operation will block until timeout, or until any tagged requests delegated by this proxy are ready for mature return, at which time their tags will be put in the returned request tag sequence. An empty tag sequence return indicates a timeout has expired.
3. Retrieve reply results from individual pollers, which have reported that they are ready for mature return by the group polling return result.

The following code sample illustrates above steps of using Native Messaging group polling feature:

```
// from: <install_dir>/examples/vbroker/NativeMessaging/
//       stock_manager/group_polling_client.C
// send one tagged request
current->request_tag(NativeMessaging::RequestTag(2,2, (CORBA::Octet*)"0"));
stock_manager_rcv->add_stock("ACME", 100.5);
pollers[0] = StockManager::_narrow(ref = current->the_poller());
// send another tagged request
current->request_tag(NativeMessaging::RequestTag(2,2, (CORBA::Octet*)"1"));
CORBA::String_var symbol = (const char*)"ACMA";
stock_manager_rcv->find_closest_symbol(symbol.inout());
pollers[1] = StockManager::_narrow(ref = current->the_poller());
...
// polling request availability on proxy and retrieve their replies
NativeMessaging::RequestTagSeq_var tags;
while(TRUE) {
    // polling availability
    try {
        tags = proxy->poll(max_timeout, TRUE);
    }
    catch(NativeMessaging::RequestAgent::PollingGroupIsEmpty&) {
        proxy->destroy(TRUE);
        break;
    }
    // retrieve replies
    for(int i=0;i<tags->length();i++) {
        int id = atoi((const char*)((tags.in())[i].get_buffer()));

        switch(id) {
            case 0: // the first tagged request sent above
                CORBA::Boolean stock_added;
                stock_added = pollers[0]->add_stock("", 0.0);
                break;

            case 1: // the second tagged request sent above
                CORBA::Boolean closest_found;
                closest_found = pollers[1]-
>find_closest_symbol(symbol.inout());
                break;

            default:
                break;
        }
    }
}
```

**Note**

- After each invocation, the Current `request_tag` attribute is automatically reset to empty or null.
- Try to initiate a 2PI on a proxy with a `request_tag` already used by another 2PI or the proxy will end up with a CORBA `BAD_INV_ORDER` exception with minor code `NativeMessaging::DUPLICATED_REQUEST_TAG`.
- The `unmask` parameter of the `poll()` operation on a request proxy specifies whether the `poll()` should unmask all mature requests. If they are unmasked, they will not be involved and reported by the next `poll()`.

– If all requests on a proxy are not tagged or unmasked, `poll()` will raise a `PollingGroupIsEmpty` exception.

## Cookie and reply de-multiplexing in reply recipients

As illustrated in previous sections, multiple requests can be delegated by a given request proxy. In the callback model, all replies to these requests will be sent back to the same reply recipient object specified on creating the proxy. The challenge is how the client demultiplexes different replies on one ReplyRecipient callback handler.

Applications using OMG CORBA Messaging also face the same challenge. To avoid activating many callback objects, CORBA Messaging suggests that applications use a POA default servant or servant manager to manipulate callback objects, and assign different object IDs to different callback references. Although this avoids many callback objects being activated in the reply recipient process, it is inflexible and far from an efficient scenario, because it requires an object reference to be created and marshaled for sending each callback request.

Native Messaging supports two demultiplexer mechanisms, which can be used either together or alone depending on the required demultiplexer granularity. A coarse grained demultiplex, but handy mechanism, is simply demultiplexing by operation signature, which is available within the ReplyRecipient's `reply_available()` callback method. This is the mechanism used in some of the previous examples.

A more effective demultiplexing mechanism in the Native Messaging callback scenario is using *request cookies*. A request cookie is an octet sequence (or byte array). Its content is specified by client applications on the Native Messaging's Current object before sending a request. The specified cookie is passed to the reply recipient's `reply_available()` method on delivering the reply of that request. There is no constraint on the content of a cookie, not even a uniqueness requirement. Contents of cookies are decided solely by applications for their own convenience and efficiency on callback demultiplexing.

The following code sample illustrates how to assign cookie to a request:

```
// send a requests with a cookie
current->the_cookie(CORBA::OctetSeq(9,9,"add stock"));
stock_manager_rcv->add_stock("ACME", 100.5);

// send another request with a different cookie
current->the_cookie(CORBA::OctetSeq(11,11,"find symbol"));
CORBA::String_var symbol = (const char*)"ACMA";
stock_manager_rcv.find_closest_symbol(symbol.inout());
```

The following code sample illustrates how to use attach cookies to demultiplex by reply recipient:

```
void StockManagerReplyRecipientImpl::reply_available(
    CORBA::Object_ptr reply_poller,
    const char* operation,
    const CORBA::OctetSequence& cookie)
{
    StockManager_var poller
        = StockManager::_narrow(reply_poller);

    CORBA::String_var id = PortableServer::
        ObjectId_to_string(cookie.get_buffer());

    // retrieve response using reflective callback
    if( strcmp(id, "add stock") == 0 ) {
        CORBA::Boolean stock_added
            = poller->add_stock("", 0.0);

        ...
 }
    else
    if( strcmp(id, "find symbol") == 0 ) {
        CORBA::String_var symbol = (const char*)"";
        CORBA::Boolean closest_found
            = poller->find_closest_symbol(symbol.inout());
        ...
    }
        ...
}
```

## Evolving invocations into two-phases

Compared to conventional single-phase invocations, two-phase invocations incur additional reply polling communication round trips. For a long duration heavyweight task, latency from few additional communication round trips is insignificant. However, for a lightweight transient invocation, this latency can be undesirable.

It is ideal for applications if lightweight transient invocations can be completed in a single-phase without incurring additional latency, and heavyweight long duration invocations can automatically be performed in two separated phases without holding client execution context and transport connection.

In Native Messaging, this can be achieved with the *evolve into two-phase invocation* feature. By default, invocations on a proxy's typed receiver always end up with premature returns along with their reply results to be polled back or delivered through callbacks later in a separate invocation phase. The *evolve into two-phase* feature allows invocations on a proxy's typed receiver to block and end up with a mature return if it can be accomplished before a specified timeout expires. Otherwise, if the invocation cannot complete before the timeout expires, it will evolve into a two-phase invocation by taking a premature return. To determine whether an invocation on a proxy's typed receiver has evolved into a two-phase invocation, the application can examine the `reply_not_available` attribute of the local Native Messaging Current object after the return.

To use this feature:

- The request proxy should be created with a `WaitReply` property with a value of `TRUE`.
- Set the `wait_timeout` attribute of Native Messaging Current to a non-zero value (milliseconds) before the invocations.
- After each invocation on the typed receiver, determine whether a return is premature by examining the `reply_not_available` attribute of the local Native Messaging Current object after each invocation.
- If a return is premature, get the returned poller object from the local Current to poll the reply in separate phase later.

The following code sample illustrates how to use the evolve invocations into two-phases:

```
// Create a request proxy with WaitReply property TRUE
NativeMessaging::PropertySeq props;
props.length(1);
props[0].id = (const char*)"WaitReply";
CORBA::Any::from_boolean fb((CORBA::Boolean)1);
props[0].value <<= fb;

NativeMessaging::RequestProxy_var proxy
    = agent->create_request_proxy(stock_manager, "", NULL, props);

// get the typed receiver of this proxy
CORBA::Object_var ref;
StockManager_var stock_manager_rcv
    = StockManager::_narrow(ref = proxy->the_receiver());

// Set wait_timeout attribute to 3 seconds
current->wait_timeout(3000);
// make an invocation on the receiver.
CORBA::Boolean stock_added = stock_manager_rcv->add_stock("ACME", 100.5);




// check whether it has evolved into a two-phase invocation.
if( ! current->reply_not_available() ) {
    // It is not evolved. The return above is mature.
```

```
        // The job has done.
        return;
    }

    // It has evolved into a two-phase invocation.
    // We should get the poller and poll its reply.
    StockManager_var poller
        = StockManager::_narrow(ref = current->the_poller());
    do { stock_added = pollers->add_stock("", 0.0); }
    while(current->reply_not_available())
```

**Note**

- If an operation on a proxy's typed receiver can be completed before it evolves into a two-phase invocation on timeout, there will be no poller generated, nor will a callback be made on the reply recipient to deliver the reply.
- If an exception is raised from blocking on a proxy or polling reply, the application should use the `reply_not_available` attribute of Native Messaging Current to determine whether the exception reports a request delivering or reply polling failure or if it is a real result of delegating the request. A value of `TRUE` for this attribute indicates that this exception is a reply delivering or polling failure between the client and agent. `FALSE` indicates that this exception is a real result of delegating the request.

## Reply dropping

In the callback model, by default, a request agent sends whatever result, return or exception, of the invocation back to the reply recipient. Reply dropping allows specified types of reply results to be filtered out. This is useful, for instance, if applications want to invoke one-way requests with no result to be returned, but would still be notified if any invocations fail.

Native Messaging allows applications to specify a `ReplyDropping` property on creating a request proxy. This property specifies which types of returns should be filtered out from being sent to the reply recipient. The value of this property is an octet (or byte) with the following filtering rules:

- if( value & 0x01 == 0x01 ) drop normal replies
- if( value & 0x02 == 0x02 ) drop system exceptions
- if( value & 0x04 == 0x04 ) drop user exceptions

For example, a value of `0x06` for this property lets the request agent drop all exceptions, system as well as user, on requests delegated by this proxy.

The following example code illustrates setting the `ReplyDropping` property:

```
    // Create a request proxy with ReplyDropping property
    // with value 0x01 (dropping all normal replies).
    NativeMessaging::PropertySeq props;
    props.length(1);
    props[0].id = (const char*)"ReplyDropping";
    CORBA::Any::from_octet fo((CORBA::Octet)0x01);
    props[0].value <<= fo;

    NativeMessaging::RequestProxy_var proxy
        = agent->create_request_proxy(stock_manager, "",
            reply_recipient, props);
    ...
```

**Note**

- Reply dropping only applies to the callback model. If the `reply_recipient` reference passed to the `create_request_proxy()` is null, the reply dropping property is ignored.

– If the value of the reply dropping property in `create_request_proxy()` is not `0x00`, and the `reply_recipient` reference is not *null*, invocation on this proxy's typed receiver will not return a poller object on Native Messaging Current.

## Manual trash collection

By default, a poller object will be trashed immediately after a polling operation on it results in a mature return. In the callback model, once the callback is returned, a request agent also trashes the poller regardless of whether the application has retrieved the reply within the callback `reply_available()` operation. Polling on a trashed object raises a CORBA `OBJECT_NOT_EXIST` exception and the Current `reply_not_available` attribute is set to `TRUE`.

If a request proxy is created with a `RequestManualTrash` property of value `TRUE`, poller objects of requests delegated by this proxy are not trashed automatically. Polling on these poller objects after a reply becomes available is idempotent, returning the same result every time.

These poller objects can be manually trashed if an application no longer needs them. To manually trash poller objects, applications simply call the `destroy_request()` operation on the request agent, with the poller to be trashed as a parameter. For example,

```
agent->destroy_request(poller);
```

**Note**

Pollers of requests delegated by an auto-trashing proxy can also be trashed manually. This makes sense when replies on these pollers are either not yet available or have not been polled back.

## Unsuppressed premature return mode

The key concept of Native Messaging is unblocking from a native operation after its first invocation phase. In Native Messaging, this is called *premature return*. There are two premature return modes in Native Messaging: *suppressed* mode and *unsuppressed* mode. All of the discussions so far used the default suppressed mode. In suppressed mode, the premature return is a normal operation return, except that it contains dummy output and return values. This is similar to an exceptional return in non-exception handling in the OMG C++ mapping, except that Native Messaging uses a thread local Current object instead of an additional Environment parameter.

Suppressed premature return mode is handy, however, it requires client-side mapping support. Namly, it assumes the IDL precompiler generated client-side stub code catches and suppresses premature return exceptions. To port client applications to an ORB, its IDL precompiler does not generate premature return suppressed client-side stub code, the unsuppressed premature return mode can be used.

In Native Messaging unsuppressed premature return mode, a native operation is unblocked by simply raising an `RNA` exception, that is a CORBA `NO_RESPONSE` exception with minor code `REPLY_NOT_AVAILABLE`. To use unsuppressed premature return mode, an application needs to turn off suppressed mode by calling `suppress_mode(false)` on Native Messaging Current, and it needs to catch and handle the `RNA` exceptions accordingly.

**Note**

To ensure that the code is portable to both suppressed and unsuppressed modes, it is recommended that applications use the Current `reply_not_available` attribute in unsuppressed mode, rather than the `RNA` exception and minor code to determine the maturity of a return.

The following example code illustrates the StockManager polling example in unsuppressed mode. This code is not only portable to all ORBs, but also portable to suppressed mode as well.

```
// from: <install_dir>/examples/vbroker/NativeMessaging/
//       stock_manager/polling_client_portable.C
```

```
static void yield_non_rna(const CORBA::NO_RESPONSE& e)
{
    if(e.minor() != NativeMessaging::REPLY_NOT_AVAILABLE ) {
        throw e;
    }
}
...
// This marco suppresses an RNA, namely a NO_RESPONSE exception
// with minor code of NativeMessaging::REPLY_NOT_AVAILABLE.
#define SUPPRESS_RNA(stmt)  \
    try { stmt; } \
    catch(const CORBA::NO_RESPONSE& e) { yield_non_rna(e); }
...
// turn off suppress mode
current->suppress_mode(FALSE);

// send several requests to the typed receiver, and
// get their reply pollers from Native Messaging Current.
StockManager_var pollers[2];
SUPPRESS_RNA( stock_manager_rcv->add_stock("ACME", 100.5) )
pollers[0] = StockManager::_narrow(ref = current->the_poller());

CORBA::String_var symbol = (const char*)"ACMA";
SUPPRESS_RNA(stock_manager_rcv->find_closest_symbol(symbol.inout()))
pollers[1] = StockManager::_narrow(ref = current->the_poller());

// poll the associated replies
current->wait_timeout(max_timeout);

CORBA::Boolean stock_added;
do { SUPPRESS_RNA(stock_added = pollers[0]->add_stock("", 0.0)) }
while(current->reply_not_available());

CORBA::Boolean closest_found;
do { SUPPRESS_RNA(closest_found
    = pollers[1]->find_closest_symbol(symbol.inout())) }
while(current->reply_not_available());
```

### Suppress poller generation in callback model

By default, pollers are generated even in the callback model. This allows:

- Applications to trash a request before it completes.
- Applications to retrieve replies independent of their reply recipients.

However, generating and sending back poller references incurs additional overhead. Native Messaging allows applications to suppress (disable) poller reference generation in the callback model.

To suppress a poller in the callback model, applications only need to create a request proxy with the `CallbackOnly` property set to `TRUE`. In this case null pollers are returned.

## Native Messaging API Specification

**Note**

Several operations and attributes in the Native Messaging IDL definition are not specified in this document. They are either value added features, depreciated features, or reserved for further extension.

## Interface RequestAgentEx

This is the interface of the Native Messaging Request Agent. A request agent is responsible for delegating invocations to their specified target object and delivering return results to client callback handlers or returning them later on client polling. See "Request Agent" for more information.

### create_request_proxy()

```
RequestProxy
create_request_proxy(
    in object target,
    in string repository_id,
    in ReplyRecipient reply_recipient,
    in PropertySeq properties)
  raises(InvalidProperty);
```

The `create_request_proxy()` method creates a request proxy to delegate two-phase invocations to the specified target object.

| Argument | Description |
|---|---|
| `target` | The target of all requests to be delegated by this proxy. |
| `repository_id` | This is the assigned repository ID of the typed receiver, reply poller, and reply holder from this proxy. If this parameter is an empty string, the target's repository ID is used. This ID is used by Native Messaging to fulfill `_is_a()` semantics on typed receiver, reply poller, and reply holder. |
| `reply_recipient` | The reply recipient callback handler. When replies become available the request agent calls back its `reply_available()` operation to send back reply results. A `null_reply_recipient` implies the polling-pulling model. |
| `properties` | Properties to specify non-default semantics of the proxy. Supported properties include:<br>■ `WaitReply`: A boolean property with default value `FALSE`. See "Reply dropping" for more information.<br>■ `RequestManualTrash`: A boolean property with default value `FALSE`. See "Manual trash collection" for more information.<br>■ `ReplyDropping`: An octet property with default value `0x00`. See "Reply dropping" for more information.<br>■ `CallbackOnly`: A boolean property with default value `FALSE`. See "Suppress poller generation in callback model" for more information. |

| Exception | Description |
|---|---|
| `InvalidProperty` | This exception indicates that an invalid property name or value is used in the properties list. The property name is available from the exception. |

### destroy_request()

```
void
destroy_request(
    in object poller)
  raises(RequestNotExist);
```

This method is used to manually trash a poller object. See "Manual trash collection" for more information.

| Argument | Description |
|---|---|
| `poller` | the poller to be trashed. |

| Exception | Description |
|---|---|
| `RequestNotExist` | This exception indicates the poller to be trashed is not available. |

## Interface RequestProxy

Request proxies are created by an application from a request agent in order to delegate requests to the specified target and with the specified semantic properties. See “create_request_proxy()”.

### the_receiver

```
readonly attribute object the_receiver;
```

This attribute is the proxy's typed receiver reference. The type receiver of a proxy supports the same IDL interface as the specified target and is where applications send their requests to be delegated by the proxy.

Note

- By default, calling operations on a proxy's typed receiver initiates two-phase invocations to be delegated by this proxy. These calls will be unblocked and yield distinct reply pollers.
- If the proxy is created with a `WaitReply` property value of `TRUE` and the request on `the_receiver` is called with a non-zero `wait_timeout`, the request agent will try to delegate the request as single-phase invocation before the timeout expires. If the agent does not receive a reply from the target before the timeout expires, it will unblock the client and the request will evolve into a two-phase invocation. After unblocking from a call on `the_receiver`, applications can use the Current `reply_not_available` attribute to determine whether the request has evolved into a two-phase invocation. See “reply_not_available”.
- IDL one-way operations only have one invocation phase intrinsically, therefore, one-way invocations on a proxy's typed receiver do not yield poller objects. The agent simply forwards them to their targets without going through a second invocation phase.
- Core operations on a proxy's typed receiver are handled synchronously; they will be blocked until a mature return or exception. Calling core operations on typed receivers does not imply initiating two-phase invocations. For instance, a `_non_existent()` call on a proxy's typed receiver only implies a ping on the receiver itself, not on the real target.

### poll()

```
RequestIdSeq
poll(
    in unsigned long timeout,
    in boolean unmask)
  raises(PollingGroupIsEmpty);
```

This method performs group polling. See “Group polling” for more information.

| Argument | Description |
|---|---|
| `timeout` | specifies the maximum length of time, in milliseconds, that this method will wait for any tagged request to become available. If no tagged request becomes available before the timeout expires an empty `RequestIdSeq` is returned. |
| `unmask` | specifies whether a tagged request, its tag is in the returned sequence, should be unmasked. Once unmasked, a tagged request will no longer be involved in subsequent group polling. |

| Exception | Description |
|---|---|
| `PollingGroupIsEmpty` | This exception indicates there are no tagged or unmasked requests pending on this proxy. |

### destroy()

```
void
destroy (
    in boolean destroy_requests);
```

This method destroys a request proxy.

| Argument | Description |
|---|---|
| destroy_requests | if TRUE, all requests delegated by this proxy are trashed. |

## Local interface Current

A local Native Messaging Current object is used by an application to specify and access additional information before and after a two-phase invocation. The Current object can be resolved from the local ORB as an initial reference. See "Native Messaging Current" for more information.

### suppress_mode()

```
void
suppress_mode(
    in boolean mode);
```

This sets the current premature return mode. In suppressed mode, two-phase invocations are unblocked after the first phase in a normal return, except that it contains dummy output and return values. In unsuppressed mode, two-phase invocations are unblocked after the first phase by an RNA exception (a CORBA NO_RESPONSE exception with minor code of NativeMessaging::REPLY_NOT_AVAILABLE). See "Unsuppressed premature return mode" for more information.

| Argument | Description |
|---|---|
| mode | specifies whether the suppressed mode is used. |

### wait_timeout

```
attribute unsigned long wait_timeout;
```

This attribute specifies the maximum number of milliseconds a two-phase invocation will block on sending a request or on polling a reply. On timeout, Native Messaging unblocks the call with a premature return.

### the_cookie

```
attribute Cookie the_cookie;
```

This attribute specifies the cookie to be sent immediately following the invocation on a proxy's typed receiver. By default, the cookie is empty. A non-empty cookie can be used by reply_recipient to do more application-specific demultiplexing. See "Cookie and reply de-multiplexing in reply recipients" for more information.

### request_tag

```
attribute RequestTag request_tag;
```

This attribute uniquely identifies the request immediately following an invocation on a proxy's typed receiver. By default the tag is initially empty, and it is reset to empty after sending the request. Requests with non-empty tags are involved in group polling. See "poll()" and "Group polling".

**Note**

– After each invocation, the Current request_tag attribute is automatically reset to empty or null.

– Attempting to initiate a 2PI on a proxy with a `request_tag` previously used by another 2PI on the proxy will result in a CORBA `BAD_INV_ORDER` exception with minor code `NativeMessaging::DUPLICATED_REQUEST_TAG`.

## the_poller

```
readonly attribute object the_poller;
```

This attribute returns the poller object reference just after delivering a request through an invocation made on a proxy's typed receiver. Poller objects are used by client applications to fulfill the reply polling-pulling phase of two-phase invocations.

**Note**

- A client application should call the same operation used in initiating the two-phase invocation on the given poller object to poll and retrieve the return result. Calling an operation on the poller that does not match the one used in initiating the two-phase invocation will result in a CORBA `BAD_OPERATION` exception, and the value of the Current `reply_not_available` attribute will be `TRUE`.
- Poller objects are normal CORBA objects with location transparency. Therefore, in Native Messaging, the request sending phase and the reply polling phase of a two-phase invocation are not necessarily carried out in same client execution context and through same transport connection. A client application can accomplish the first invocation phase and get the poller object, then perform the polling in a completely distinct client execution context, in a different process, and through a different transport connection.
- If an exception is raised in the reply polling-pulling phase, an application should use the Current `reply_not_available` attribute to determine whether the exception reports a reply polling-pulling failure or a successful reply pulling of a real exceptional result of the delegated request. `TRUE` indicates that this exception is a polling-pulling failure between the client and agent. `FALSE` indicates that this exception is the real result of the delegated request.
- Core operations made on poller objects are orthogonal to two-phase invocations pending on them. For instance, `_is_a()` or `_non_existent()` on a poller does not imply reply polling-pulling on the pending two-phase invocation, but only implies a repository ID comparison and non-existence check on the poller object itself.

## reply_not_available

```
readonly attribute boolean reply_not_available;
```

This attribute reports the consequence of an unblocked (either normal return or exception) call on a proxy's typed receiver, reply poller, or reply holder, as summarized by the following table.

| Reply_not_available | True | False | True | False |
|---|---|---|---|---|
| Called object | Proxy's typed receiver | Reply poller or holder | | |
| Normal return, no exception | 2PI initiated (premature) | 2PI completed | (poller only) Reply not available (premature) | 2PI completed |
| RNA exception (unsuppressed mode) | 2PI initiated (premature) | N/A | (poller only) Reply not available (premature) | N/A |
| Exception other than RNA | 2PI initiation failure | 2PI completed (target failure) | Polling-pulling failure | 2PI completed (target failure) |

The terms in the above table are defined as follows:

- **2PI initiated**: This is the result when an operation made on a proxy's typed receiver results in a normal return or an `RNA` exception (in unsuppressed mode), and the Current `reply_not_available` attribute is `TRUE`. This is one of the two premature return cases in Native Messaging. By default, a reply poller of this initiated two-phase invocation is available on Current after the call.
- **2PI initiation failure**: This is the result when an operation made on a proxy's typed receiver results in an exception other than `RNA`, and the Current `reply_not_available` attribute is `TRUE`. This outcome indicates either that the agent has rejected the two-phase invocation, or the client failed to receive agent's premature reply message. No reply poller is available on Current. If this is caused by a communication failure on receiving a premature reply message, the agent will still delegate the request and may even generate a callback to a reply recipient.
- **2PI completed**: This is the result when an operation made on a proxy's typed receiver, a reply poller or reply holder, results in either a normal return or any CORBA exception, and the Current `reply_not_available` attribute is `FALSE`. If the operation results in an exception other than `RNA`, a `TRUE` `reply_not_available` attribute indicates that this exception is a real result of a delegated request to target.
- **Reply not available**: This is the result when an operation made on a reply poller results in a normal return or an `RNA` exception, and the Current `reply_not_available` attribute is `TRUE`. This is one of the two premature return cases.
- **Polling-Pulling failure**: This is the result when an operation made on a reply poller or reply holder results in an exception other than `RNA`, and the Current `reply_not_available` attribute is `TRUE`. This outcome indicates a usage or system failure on retrieving the reply, such as calling an unmatched operation or the poller has already been trashed.
- **N/A**: Not an applicable outcome. It should never happen.

## Interface ReplyRecipient

`ReplyRecipient` objects are implemented by Native Messaging applications to receive reply results in the callback model. See the example in “Callback model” and “Cookie and reply de-multiplexing in reply recipients”.

### reply_available()

```
void
reply_available(
    in object reply_holder,
    in string operation,
    in Cookie the_cookie);
```

This method is callback by request agent on delivering a reply. The actual reply result, either a normal return or an exception, is held by the input `reply_holder` object and can be retrieved by making a callback on it. If an exception is raised from a call on the `reply_holder`, the application should use the Current `reply_not_available` attribute to determine whether the exception is reporting a retrieval failure or the real result of the delegated request. `TRUE` indicates that this exception is the result of a retrieval failure between the client and agent. `FALSE` indicates that this exception is a real result of the delegated request.

See the example in “Callback model”.

| Argument | Description |
|---|---|
| `reply_holder` | Within the scope of the `reply_available()` method, this object reference has the same semantics as a reply poller. A reply retrieving operation on `reply_holder` should only be made within the scope of the `reply_available()` method. Once the application returns from `reply_available()`, the `reply_holder` may no longer be valid. |
| `operation` | The original operation signature. It can be used by applications for coarse grained demultiplexing. A call made on the `reply_holder` reference should have same operation signature as this parameter. Making a call on the `reply_holder` with a different operation will end up with a CORBA `BAD_OPERATION` exception with Current `reply_not_available` attribute value of `TRUE`. |
| `the_cookie` | The original request cookie. Can be used by applications for fine grained demultiplexing. |

## Semantics of core operations

Native Messaging reserves all pseudo operations as *core operations*. Core operations meet the following rules:

- They are always accomplished in one phase. Core operations always block until a mature return or a non-`RNA` exception.
- They do not initiate a two-phase invocation to be forwarded to the real target when called on a proxy's typed receiver. For instance, calling `_non_existent()` on a proxy's typed receiver is only a ping to check the non-existence of the receiver itself, not the target.
- They are orthogonal to pending two-phase invocations on a reply poller or reply holder: For instance, calling `_is_a()` or `_non_existent()` on a reply poller or reply holder does not imply retrieving the reply result of the pending two-phase invocation, but only repository ID comparsion and existence checks on these poller or holder objects themselves.

## Native Messaging Interoperability Specification

The content of this section is not intended for Native Messaging application developers but for third party Native Messaging vendors.

## Native Messaging uses native GIOP

In non-native messaging, such as CORBA Messaging, the OMG GIOP protocol is not used as a direct message protocol; it is used as a tunneling protocol for another ad hoc message routing protocol.

For instance, in CORBA Messaging, calling a mangled operation

```
sendc_foo(<input_parameter_list>);
```

does not incur a native OMG GIOP Request message with operation `sendc_foo` in the head and `<input_parameter_list>` as payload. Instead, a routing message tunneling through GIOP Request is sent.

Native Messaging uses the native OMG GIOP directly as its message level protocol:

– A method call on an agent, request proxy's typed receiver, reply poller, reply recipient, or reply holder reference incurs a native GIOP Request message with the exact called operation name in head, and the exact input parameters as payload to be sent, as defined by OMG GIOP.

– A premature return is simply a native GIOP Reply message containing an `RNA` exception, specifically a CORBA `NO_RESPONSE` exception with minor code of `REPLY_NOT_AVAILABLE`.

– A mature return is simply a native GIOP Reply message with either the exact `<return_value_and_output_parameter_list>` or the exact exception from the target as payload.

## Native Messaging service context

Like the OMG Security and Transaction service, Native Messaging also uses a service context to achieve certain semantic results. The client-side Native Messaging engine, implemented in an OMG standardized PortableInterceptor for instance, is responsible for creating and adding required service contexts into certain outgoing requests and for extracting information from the same kind of service context inside incoming replies.

The `context_id` used by Native Messaging's service context is `NativeMessaging::NMService`. The `context_data` is an encapsulated `NativeMessaging::NMContextData` defined as:

```
module NativeMessaging {
...
    const IOP::ServiceID NMService = ...

    struct RequestInfo {
        RequestTag request_tag;
        Cookie the_cookie;
        unsigned long wait_timeout;
    };
    union NMContextData switch(short s) {
        case 0: RequestInfo req_info;
        case 1: unsigned long wait_timeout;
        case 2: object the_poller;
        case 3: string replier_name;
    };
};
```

Mandated usage of different context data in Native Messaging is summarized in the following table:

| Sending to or receiving from | Proxy's typed receiver | Reply poller | Reply holder |
|---|---|---|---|
| Request | req_info | wait_timeout | Not defined |
| Normal Reply (NO_EXCEPTION) | Not defined | | |
| RNA Exception | the_poller | No NMService context | N/A |

| Sending to or receiving from | Proxy's typed receiver | Reply poller | Reply holder |
|---|---|---|---|
| Non-RNA exception from calling target | replier_name | | |
| Non-RNA exception within agent | No NMService context | | |

The terms in the above table are defined as follows:

– **req_info**: NMContextData is mandated to all requests of two-way non-core operation sending to a proxy's typed receiver. This context has `request_tag`, `cookie` and `wait_timeout` from Native Messaging Current as supplement parameters for initiating a two-phase invocation. The content of this context should be used by the request agent to tag the request, to deliver callback with the cookie, and to wait before evolving into a two-phased invocation. See corresponding topics in the previous sections.

– **wait_timeout**: NMContextData is mandated to all normal (two-way non-core) requests sent to a reply poller, with `wait_timeout` from Native Messaging Current as supplement parameter for polling. The content, namely the `wait_timeout`, should be used by the request agent to block the call before a mature or premature return. See corresponding topics in previous sections.

– **the_poller**: NMContextData is mandated to all successful returns on initiating two-phase invocations on a proxy's typed receiver object. The content of the context, a poller reference, is extracted and copied to Native Messaging Current's `the_poller` attribute.

– **replier_name**: NMContextData is mandated to all exceptional returns as a successful return of an exceptional return result from delegating a request. This context should not appear if the exceptional return is a failure not resulting from delegating the request. The actual content of the string should be empty and preserved for further extension.

– **Not defined**: Native Messaging does not use NMService context in these cases.

– **N/A**: Not applicable. It should never happen.

### NativeMessaging tagged component

A tagged component with the `NativeMessaging::TAG_NM_REF` tag should be embedded in typed receivers of request proxies and poller references. The `component_data` of this tagged component encapsulates an octet. Namely the first octet of the `component_data` is the byte-order byte and second byte of it is the value octet. A value of `0x01` for this octet indicates the reference is a typed receiver of a request proxy, and a value of `0x02` indicates it is a poller reference.

This component is used by PortableInterceptor's `send_request()` method to determine whether a request is sending to a Native Messaging request proxy's `the_receiver` reference, a reply poller, or something else, and to decide whether and what service context to add to the outgoing request.

## Using Borland Native Messaging

### Using request agent and client model

#### Start the Borland Request Agent

To start the Request Agent service, run the command `requestagent`. Run it with `requestagent -?` to see the usage information.

### Borland Request Agent URL

To use Native Messaging, a request agent needs to be known by client applications. Usually, this is done by initializing the client ORB with the OMG standardized ORB initialize command arguments:

```
-ORBInitRef RequestAgent=<request_agent_ior_or_url>
```

This allows client applications to resolve the request agent reference from the ORB as an initial service, for instance:

```
// Getting Request Agent reference in C++
CORBA::Object_var ref
    = orb->resolve_initial_references("RequestAgent");
NativeMessaging::RequestAgentEx_var agent
    = NativeMessaging::RequestAgentEx::_narrow(ref);
```

By default, the URL of a request agent is:

```
corbaloc::<host>:<port>/RequestAgent
```

Here, `<host>` is the host name or dotted IP address of a Request Agent server, and `<port>` is the TCP listener port number of this server. By default, the Native Messaging Request Agent uses port 5555.

### Using the Borland Native Messaging client model

Borland Native Messaging client side models in C++ are implemented as OMG portable interceptors and are referred to as the Native Messaging Client Component. Native Messaging C++ Client Components are implicitly enabled/disabled by link (or load in) with a Native Messaging Client (including callback object) application.

## Borland Request Agent vbroker properties

### vbroker.requestagent.maxThreads

Specifies the maximum number of threads for request invocation. The default value is `0` (zero) which means no limit. Values cannot be negative.

### vbroker.requestagent.maxOutstandingRequests

Specifies the maximum queue size for requests waiting to get serviced. This property only takes effect if the `maxThreads` property is set to non-zero value. The default value is `0` (zero) which means no limit. Values cannot be negative. If a request arrives when the queue size is equal to maximum size, the request waits for a timeout until there is space in the queue. See "vbroker.requestagent.blockingTimeout".

### vbroker.requestagent.blockingTimeout

Specifies the maximum time, in milliseconds, that a request can wait before it is added to the queue. The default value is `0` (zero) which means no wait. Values cannot be negative. If the value is set to `0` (zero) and a request arrives and the queue is full, the Request Agent will raise `CORBA::IMP_LIMIT` exception. Otherwise, the request waits for the specified timeout. After the timeout, either the request gets executed immediately if the queue is empty and worker thread is available, or the request is enqueued in the waiting queue if the queue has space and the request remains there until it gets serviced, or if the queue is still full, `CORBA::IMP_LIMIT` exception is raised by the Request Agent.

### vbroker.requestagent.router.ior

Specifies the IOR of OMG messaging router. The default value is empty string.

#### vbroker.requestagent.listener.port

Specifies the TCP listener port to be used by the request agent. The default value is 5555.

#### vbroker.requestagent.requestTimeout

This property specifies the maximum time, in milliseconds, that the agent will hold the reply result for its client. If request agent has received reply results on a request, but the client does not pull the result or trash the request, the request agent will trash the request (together with its reply result) upon the expiration of the request timeout set by this property. The default value of this property is infinity, meaning the agent will preserve the reply results until they are trashed by client applications (manually or automatically).

### Interoperability with CORBA Messaging

The Native Messaging Request Agent is forward interoperable with the OMG untyped Messaging Router. Specifically, the Request Agent can be configured to route requests through an OMG untyped router instead of sending them directly to their specified targets. To do so, the request agent needs to be started with the "vbroker.requestagent.router.ior" property with a valid CORBA Messaging router IOR as value.

## Migrating from previous versions of VisiBroker Native Messaging

In VisiBroker 6.0 some changes in the Native Messaging IDL have been made that can impact source and binary level compatibility of the applications written using VisiBroker 5.x Native Messaging.

There are two changes that VisiBroker 5.x application developers have to pay attention to. One is in the `Property` structure and other is the definition of `OctetSeq`.

#### Property

In VisiBroker 5.x, Property structure was defined as follows:

```
module NativeMessaging {
  struct Property {
        string  name;
        any     value;
  };
};
```

In VisiBroker 6.0 Property has been typedef to CORBA::NameValuePair. That is:

```
typedef CORBA::NameValuePair    Property;
typedef CORBA::NameValuePairSeq PropertySeq;
```

Native Messaging 5.x applications therefore have to be migrated to use `CORBA::NameValuePair` for `Property`.

#### OctetSeq

In VisiBroker 5.x, `OctetSeq` was defined as:

```
typedef sequence<octet>      OctetSeq;
```

`RequestTag` and `Cookie` were defined as follows:

```
typedef OctetSeq             RequestTag;
typedef OctetSeq     Cookie;
```

In VisiBroker 6.0, this definition is removed and `RequestTag` and `Cookie` are defined as follows:

```
typedef CORBA::OctetSeq      RequestTag;
typedef CORBA::OctetSeq      Cookie;
```

With this change the reply_available() method of ReplyRecipient interface has to be changed from

```
void reply_available (CORBA::Object_ptr replyHolder,
 const char* operation, const NativeMessaging::OctetSeq& cookie)
```

to

```
void reply_available (CORBA::Object_ptr replyHolder,
 const char* operation, const CORBA::OctetSeq& cookie)
```

Therefore, NativeMessaging 5.x applications have to be migrated to use `CORBA::OctetSeq` for `RequestTag` and `Cookie`.

These changes need to be done manually; there is no migration tool available. Note that any VisiBroker 5.x Native Messaging application is “on-the-wire” compatible with VisiBroker 6.0 Request Agent.

## Migrating from previous versions of VisiBroker Native Messaging

In VisiBroker 6.0 some changes in the Native Messaging IDL have been made that can impact source and binary level compatibility of the applications written using VisiBroker 5.x Native Messaging. The main change is in the `Property` structure. In VisiBroker 5.x, this structure was defined as follows:

```
module NativeMessaging {
  struct Property {
        string  name;
        any     value;
  };
};
```

In VisiBroker 6.0 `Property` has been typedef to `CORBA::NameValuePair`. That is:

```
typedef CORBA::NameValuePair     Property;
typedef CORBA::NameValuePairSeq  PropertySeq;
```

Native Messaging 5.x applications therefore have to be migrated to use `CORBA::NameValuePair` for `Property`. These changes need to be done manually; there is no migration tool available. Note that any VisiBroker 5.x Native Messaging application is “on-the-wire” compatible with VisiBroker 6.0 Request Agent.

19

# Using the Object Activation Daemon (OAD)

This section discusses how to use the Object Activation Daemon (OAD).

## Automatic activation of objects and servers

The Object Activation Daemon (OAD) is the VisiBroker implementation of the Implementation Repository. The Implementation Repository provides a runtime repository of information about the classes a server supports, the objects that are instantiated, and their IDs. In addition to the services provided by a typical Implementation Repository, the OAD is used to automatically activate an implementation when a client references the object. You can register an object implementation with the OAD to provide this automatic activation behavior for your objects.

Object implementations can be registered using a command-line interface (`oadutil`). There is also a VisiBroker ORB interface to the OAD, described in "IDL interface to the OAD". In each case, the repository ID, object name, the activation policy, and the executable program representing the implementation must be specified.

**Note**

You can use the VisiBroker OAD to instantiate servers generated with VisiBroker for Java and C++.

The OAD is a separate process that only needs to be started on those hosts where object servers are to be activated on demand.

### Locating the Implementation Repository data

Activation information for all object implementations registered with the OAD are stored in the Implementation Repository. By default, the Implementation Repository data is stored in a file named `impl_rep` in the `<install_dir>/adm/impl_dir` directory.

### Activating servers

The OAD activates servers in response to client requests. VisiBroker clients and non-VisiBroker IIOP-compliant clients can activate servers through the OAD.

Any client that uses the IIOP protocol can activate a VisiBroker server when that server's reference is used. The server's exported Object Reference points to the OAD and the client can be forwarded to the spawned server in accordance with the rules of IIOP. To allow true persistence of the server's object references (such as through a Name Service), the OAD must always be started on the same port. For example, to start the OAD on port `16050`, enter the following:

```
prompt> oad -VBJprop vbroker.se.iiop_tp.scm.iiop_tp.listener.port=16050
```

**Note**

Port `16000` is the default port, but it can be changed by setting the `listener.port` property.

## Using the OAD

The OAD is an optional feature that allows you to register objects that are to be started automatically when clients attempt to access them. Before starting the OAD, you should first start the Smart Agent. For more information, see “Starting a Smart Agent (osagent)”.

### Starting the OAD

**Windows**

To start the OAD:

- Use the `oad.exe` located in `<install_dir>\bin\`

  or

- Enter the following at the command prompt:

```
prompt> oad
```

The `oad` command accepts the following command line arguments:

| Option | Description |
|---|---|
| `-verbose` | Turns on verbose mode. |
| `-version` | Prints the version of this tool. |
| `-path` <path> | Specifies the platform-specific directory for storing the Implementation Repository. This overrides any setting provided through the use of environment variables. |
| `-filename` <repository_filename> | Specifies the name of the Implementation Repository. If you do not specify it, the default is `impl_rep`. This overrides any user environment variable settings. |
| `-timeout` <#_of_seconds> | Specifies the amount of time the OAD will wait for a spawned server process to activate the requested VisiBroker ORB object. The default time-out is `20` seconds. Set this value to `0` (zero) if you wish to wait indefinitely. If a spawned server process does not activate the requested object within the time-out interval, the OAD will kill the spawned process and the client will see a `CORBA::NO IMPLEMENT` exception. Turn on the verbose option to see more detailed information. |
| -`IOR` <IOR_filename> | Specifies the filename to store the OAD's stringified IOR. |
| `-kill` | Stipulates that an object's child process should be killed once all of its object are unregistered with the OAD. |
| `-no_verify` | Turns off check for validity of registrations. |

| Option | Description |
|---|---|
| `-?` | Displays command usage. |
| `-readonly` | When the OAD is started with the `-readonly` option, no changes can be made to the registered objects. Attempts to register or unregister objects will return an error. The `-readonly` option is usually used after you've made changes to the Implementation Repository, and have restarted the OAD in `readonly` mode to the prevent any additional changes. |

The OAD is installed as Windows Service, allowing you to control it with the Service Manager provided with Windows.

**UNIX**

To start the OAD enter the following command:

```
prompt> oad &
```

## Using the OAD utilities

The `oadutil` commands provide a way for you to manually register, unregister, and list the object implementations available on your VisiBroker system. The `oadutil` commands are implemented in Java and use a command line interface. Each command is accessed by invoking the `oadutil` command, passing the type of operation to be performed as the first argument.

**Note**

An object activation daemon process (`oad`) must be started on at least one host in your network before you can use the `oadutil` commands.

The `oadutil` command has the following syntax:

```
oadutil {list|reg|unreg} [options]
```

The options for this tool vary, depending on whether you specify `list`, `reg` or `unreg`.

### Converting interface names to repository IDs

Interface names and repository IDs are two ways of representing the type of interface the activated object should implement. All interfaces defined in IDL are assigned a unique repository identifier. This string is used to identify a type when communicating with the Interface Repository, the OAD, and most calls to the VisiBroker ORB itself.

When registering or unregistering an object with the OAD, the `oadutil` commands allow you to specify either an object's IDL interface name or its repository id.

An interface name is converted to a repository ID as follows:

1. Prepend "`IDL:`" to the interface name.
2. Replace all non-leading instances of the scope resolution operator (`::`) with a slash (`/`) character.
3. Append "`:1.0`" to the interface name.

   For example, the IDL interface name

   ```
   ::Module1::Module2::IntfName
   ```

   would be converted to the following repository ID:

   ```
   IDL:Module1/Module2/IntfName:1.0
   ```

The `#pragma` ID and `#pragma` prefix mechanisms can be used to override the default generation of repository ID's from interface names. If the `#pragma` ID mechanism is used in user-defined IDL files to specify non-standard repository IDs, the conversion process outlined above will not work. In these cases, you must use `-r` repository ID argument and specify the object's repository ID.

To obtain the repository ID of the object implementation's most derived interface in C++, use the method `<interface_name>._repository_id()` defined for all CORBA objects.

## Listing objects with oadutil list

The `oadutil list` utility allows you to list all VisiBroker ORB object implementations registered with the OAD. The information for each object includes:

- Interface names of the VisiBroker ORB objects.
- Instance names of the object offered by that implementation.
- Full path name of the server implementation's executable.
- Activation policy of the VisiBroker ORB object (shared or unshared).
- Reference data specified when the implementation was registered with the OAD.
- List of arguments to be passed to the server at activation time.
- List of environment variables to be passed to the server at activation time.

The `oadutil list` command returns all VisiBroker ORB object implementations registered with the OAD. Each OAD has its own Implementation Repository database where the registration information is stored.

#### Note

An OAD process must be started on at least one host in your network before you can use the oadutil list command.

The `oadutil list` command has the following syntax:

```
oadutil list [options]
```

The `oadutil list` command accepts the following command line arguments:

| Option | Description |
|---|---|
| `-i <interface name>` | Lists the implementation information for objects of a particular IDL interface name. Only one of the following options may be specified at a particular time: `-i`, `-r`, `-s`, or `-poa`.<br>**Note:** All communications with the VisiBroker ORB reference an object's repository id instead of the interface name. For more information about the conversion performed when specifying an interface name, see "Converting interface names to repository IDs". |
| `-r <repository id>` | Lists the implementation information of a specific repository id. See "Converting interface names to repository IDs" for details on specifying repository IDs. Only one of the following options may be specified at a particular time: `-i`, `-r`, `-s`, or `-poa`. |
| `-s <service name>` | Lists the implementation information for a specific service name. Only one of the following options may be specified at a particular time: `-i`, `-r`, `-s`, or `-poa`. |
| `-poa <poa_name>` | Lists the implementation information for a specific POA name. Only one of the following options may be specified at a particular time: `-i`, `-r`, `-s`, or `-poa`. |
| `-o <object name>` | Lists the implementation information for a specific object name. You can use this only if the interface or repository id is specified in the command statement. This option is not applicable when an `-s` or `-poa` arguments is used. |
| `-h <OAD host name>` | Lists the implementation information for objects registered with an OAD running on a specific remote host. |
| `-verbose` | Turns verbose mode on, causing messages to be output to stdout. |
| `-version` | Prints the version of this tool. |
| `-full` | Lists the status of all implementations registered with the OAD. |

The following is an example of a local list request, specifying an interface name and object name:

```
oadutil list -i Bank::AccountManager -o BorlandBank
```

The following is an example of a remote list request, specifying a host IP address:

```
oadutil list -h 206.64.15.198
```

## Registering objects with oadutil

The `oadutil` command can be used to register an object implementation from the command line or from within a script. The parameters are either the interface name and object name, the service name, or the POA name, and path name to the executable that starts the implementation. If the activation policy is not specified, the shared server policy will be used by default. You may write an implementation and start it manually during the development and testing phases. When your implementation is ready to be deployed, you can simply use `oadutil` to register your implementation with the OAD.

**Note**

When registering an object implementation, use the same object name that is used when the implementation object is constructed. Only named objects (those with a global scope) may be registered with the OAD.

The `oadutil reg` command has the following syntax:

```
oadutil reg [options]
```

**Note**

An `oad` process must be started on at least one host in your network before you can use the `oadutil reg` command.

The options for the `oadutil reg` command accepts the following command-line arguments:

| Option | Required | Description |
|---|---|---|
| `-i <interface name>` | Yes | Specifies a particular IDL interface name. Only one of the following options may be specified at a particular time: `-i`, `-r`, `-s`, or `-poa`. See "Converting interface names to repository IDs" for details on specifying repository IDs. |
| `-r <repository id>` | Yes | Specifies a particular repository id. Only one of the following options may be specified at a particular time: `-i`, `-r`, `-s`, or `-poa`. |
| `-s <service name>` | Yes | Specifies a particular service name. Only one of the following options may be specified at a particular time: `-i`, `-r`, `-s`, or `-poa`. |
| `-poa <poa_name>` | Yes | Use this option to register the POA instead of an object implementation. Only one of the following options may be specified at a particular time: `-i`, `-r`, `-s`, or `-poa`. |
| `-o <object name>` | Yes | Specifies a particular object. You can use this only if the interface name or repository id is specified in the command statement. This option is not applicable when an `-s` or `-poa` argument is used. |
| `-cpp <file name to execute>` | Yes | Specifies the full path of an executable file that must create and register an object that matches the `-o`/`-r`/`-s`/`-poa` arguments. Applications registered with the -cpp argument must be stand-alone executables. |
| `-java <full class name>` | Yes | Specifies the full name of a Java class containing a main routine. This application must create and register an Object that matches the `-o`/`-r`/`-s`/`-poa` argument. Classes registered with the `-java` argument will be executed with the command `vbj <full_classname>`. |

| Option | Required | Description |
|---|---|---|
| `-host <OAD host name>` | No | Specifies a specific remote host where the OAD is running. |
| `-verbose` | No | Turns verbose mode on, causing messages to be output to stdout. |
| `-version` | No | Prints the version of this tool. |
| `-cos_name <CosName>` | No | Specifies the CosName to bind this registration to NOTE: This does not work with service or POA registrations. |
| `-d <referenceData>` | No | Specifies reference data to be passed to the server upon activation. |
| `-a arg1 -a arg2` | No | Specifies the arguments to be passed to the spawned executable as command-line arguments. Arguments can be passed with multiple `-a` (arg) parameters. They will be propagated in order to create the spawned executable. |
| `-e env1 -e env2` | No | Specifies environment variables to be passed to the spawned executable. Arguments can be passed with multiple `-e` (env) parameters. They will be propagated in order to create the spawned executable. |
| `-p <shared\|unshared>` | No | Specifies the activation policy of the spawned objects. The default policy is `SHARED_SERVER`. **Shared:** Multiple clients of a given object share the same implementation. Only one server is activated by an OAD at a particular time. **Unshared:** Only one client of a given implementation will bind to the activated server. If multiple clients wish to bind to the same object implementation, a separate server is activated for each client application. A server exits when its client application disconnects or exits. |

## Example: Specifying repository ID

The following command will register with the OAD the VisiBroker program `factory`. It will be activated upon request for objects of repository ID `IDL:ehTest/Factory:1.0` (which corresponds to the interface name `ehTest::Factory`). The instance name of the object to be activated is `ReentrantServer`, and that name is also passed to the spawned executable as a command-line argument. This server has the unshared policy, by which it will be terminated when the requesting client breaks its connection to the spawned server.

```
prompt> oadutil reg -r IDL:ehTest/Factory:1.0 -o ReentrantServer \
        -cpp /home/developer/Project1/factory_r -a ReentrantServer \
        -p unshared
```

## Example: Specifying IDL interface name

The following command will register the VisiBroker `Server` class with the OAD. In this example, the specified class must activate an object of repository ID `IDL:Bank/AccountManager:1.0` (corresponding to the interface name IDL name `Bank::AccountManager`) and instance name `CreditUnion`. The server will be started with unshared policy, ensuring that it will terminate when the requesting client breaks its connection. The server is also passed with an environment variable `DEBUG=1` when it is first started by the client.

```
prompt> oadutil reg -i Bank::AccountManager -o CreditUnion \
        -cpp Server -a CreditUnion -p unshared -e DEBUG=1
```

## Remote registration to an OAD

To register an implementation with an OAD on a remote host, use the `-h` argument to `oadutil reg`.

The following is an example of how to perform a remote registration to an OAD on Windows from a UNIX shell. The double backslashes are necessary to avoid having the shell interpret the backslashes before passing them to `oadutil`.

```
prompt> oadutil reg -r IDL:Library:1.0 Harvard \
 -cpp c:\\vbroker\\examples\\library\\libsrv.exe -p shared -h 100.64.15.198
```

## Using the OAD without using the Smart Agent

To access a server using the OAD without involving the Smart Agent, use the property `vbroker.orb.activationIOR` to indicate the OAD's IOR to `oadutil` and to the server.

For example, let us assume that the OAD's IOR is located in the e:/adm dir (on Windows), and you want to run the `bank_portable` example that is included (in the `examples/basic/bank_portable directory`) with with the product. To access this server without using the Smart Agent:

1 **Start the OAD:** the classpath visible to OAD must include the Server's classpath. The command is:

```
prompt>start oad -VBJprop vbroker.agent.enableLocator=false -verbose
```

2 **Register the server using `oadutil`:** the command is:

```
prompt> oadutil -VBJprop vbroker.orb.activationIOR=file:///e:/adm/oadj.ior -
VBJprop
     vbroker.agent.enableLocator=false reg -i Bank::AccountManager
 -o BankManager -cpp Server
```

3 **Generate the Server's IOR:** when the server is started it will write out it's IOR into a file. Terminate the server once it is running, so that the launching of the server by the OAD can be demonstrated. The command is:

```
prompt> Server -Dvbroker.orb.activationIOR=file:///e:/adm/oadj.ior Server
```

4 **Run the Client:** make sure the OAD is running, then use the command:

```
prompt> Client -Dvbroker.agent.enableLocator=false
```

## Using the OAD with the Naming Service

OAD facilitates the use of the Naming Service for bootstrapping. In the above section, the Smart Agent was not used, and the client needed to obtain the server's IOR file. This bootstrapping can be achieved using the Naming Service instead, as illustrated in the following steps.

1 Start the OAD, providing it with a reference to the Naming Service. Assume that the Naming Service runs on port 1111 on host myhost.

```
prompt>oad -verbose -VBJprop
vbroker.orb.initRef=NameService=corbaloc::myhost:1111/NameService
```

2 Register the server with the OAD. Note the use of the `-cos_name` parameter which indicates to the OAD that this server should be automatically bound to the Naming Service.

```
prompt>oadutil -VBJprop vbroker.orb.activationIOR=file:///e:/adm/oadj.ior -
VBJprop
  vbroker.agent.enableLocator=false reg -i Bank::AccountManager -o
BankManager
  -cos_name simple_test -cpp Server/pre>
```

```
prompt>oadutil -VBJprop vbroker.orb.activationIOR=file:///e:/adm/oadj.ior -
VBJprop
  vbroker.agent.enableLocator=false reg -i Bank::AccountManager -o
BankManager
  -cos_name simple_test -java Server
```

3 The client can then use the Naming Service to resolve and obtain the server's reference. A snippet of the client code for a Java client is shown below.

```
prompt>org.omg.CORBA.Object server=
       rootCtx.resolve(new NameComponent[] {new
NameComponent("simple_test","")});
```

Note that the OAD automatically created a binding for the server in the Naming Service because the `-cos_name` parameter was used.

## Distinguishing between multiple instances of an object

Your implementation can use `ReferenceData` to distinguish between multiple instances of the same object. The value of the reference data is chosen by the implementation at object creation time and remains constant during the lifetime of the object. The `ReferenceData` typedef is portable across platforms and VisiBroker ORBs.

VisiBroker does not use the `inf_ptr`, which is defined by the CORBA specification to identify the interface of the object being created. Applications created with VisiBroker should always specify a `NULL` value for this parameter.

## Setting activation properties using the CreationImplDef class

The `CreationImplDef` class contains the properties the OAD requires to activate a VisiBroker ORB object: `path_name`, `activation_policy`, `args`, and `env`. The following sample shows the `CreationImplDef` struct.

The `path_name` property specifies the exact path name of the executable program that implements the object. The `activation_policy` property represents the server's activation policy, which is used in object creation and registration.

```
module extension {
...
   enum Policy {
      SHARED_SERVER,
      UNSHARED_SERVER
   };
   struct CreationImplDef {
      CORBA::RepositoryId repository_id;
      string                 object_name;
      CORBA::ReferenceData    id;
      string                 path_name;
      Policy                 activation_policy;
      CORBA::StringSequence   args;
      CORBA::StringSequence   env;
   };
...
};
```

## Dynamically changing an ORB implementation

The sample below shows the `change_implementation()` method which can be used to dynamically change an object's registration. You can use this method to change the object's activation policy, path name, arguments, and environment variables.

```
module Activation
{
...
   void change_implementation(in extension::CreationImplDef old_info,
         in extension::CreationImplDef new_info)
      raises ( NotRegistered, InvalidPath, IsActive );
...
};
```

**Caution**

Although you can change an object's implementation name and object name with the `change_implementation()` method, you should exercise caution. Doing so will prevent client programs from locating the object with the old name.

## OAD Registration using OAD::reg_implementation

Instead of using the `oadutil reg` command manually or in a script, VisiBroker allows client applications to use the `OAD::reg_implementation` operation to register one or more objects with the activation daemon. Using this operation results in an object implementation being registered with the OAD and the `osagent`. The OAD will store the information in the Implementation Repository, allowing the object implementation to be located and activated when a client attempts to bind to the object.

```
module Activation {
...
   typedef sequence<ObjectStatus> ObjectStatus List;
...
   typedef sequence<ImplementationStatus> ImplStatusList;
...
   interface OAD {
      // Register an implementation.
      Object reg_implementation(in extension::CreationImplDef impl)
         raises (DuplicateEntry, InvalidPath);
   }
}
```

The `CreationImplDef` struct contains the properties the OAD requires. The properties are `repository_id`, `object_name`, `id`, `path_name`, `activation_policy`, `args`, and `env`. Operations for setting and querying their values are also provided. These additional properties are used by the OAD to activate an VisiBroker ORB object.

```
struct CreationImplDef {
   CORBA::RepositoryId repository_id;
   string object_name;
   CORBA::ReferenceData id;
   string path_name;
   Policy activation_policy;
   CORBA::StringSequence args;
   CORBA::StringSequence env;
};
```

The `path_name` property specifies the exact path name of the executable program that implements the object. The `activation_policy` property represents the server's activation policy. The `args` and `env` properties represent optional arguments and environment settings to be passed to the server.

### Arguments passed by the OAD

When the OAD starts an object implementation it passes all of the arguments that were specified when the implementation was registered with the OAD.

## Un-registering objects

When the services offered by an object are no longer available or temporarily suspended, the object should be unregistered with the OAD. When the VisiBroker ORB object is unregistered, it is removed from the Implementation Repository. The object is also removed from the Smart Agent's dictionary. Once an object is unregistered, client programs will no longer be able to locate or use it. In addition, you will be unable to use the `OAD.change_implementation()` method to change the object's implementation. As with the registration process, un-registering may be done either at the command line or programmatically.

### Un-registering objects using the oadutil tool

The `oadutil unreg` command allows you to unregister one or more object implementations registered with the OAD. Once an object is unregistered, it can no longer be automatically activated by the OAD if a client requests the object. Only objects that have been previously registered via the `oadutil reg` command may be unregistered with `oadutil unreg`.

If you specify only an interface name, all VisiBroker ORB objects associated with that interface will be unregistered. Alternatively, you may identify a specific VisiBroker ORB object by its interface name and object name. When you unregister an object, all processes associated with that object will be terminated.

**Note**

An `oad` process must be started on at least one host in your network before you can use the oadutil reg command.

The `oadutil unreg` command has the following syntax:

```
oadutil unreg [options]
```

The options for the `oadutil unreg` command accepts the following command line arguments:

| Option | Required | Description |
|---|---|---|
| `-i <interface name>` | Yes | Specifies a particular IDL interface name. Only one of the following options may be specified at a particular time: `-i`, `-r`, `-s`, or `-poa`. See "Converting interface names to repository IDs" for details on specifying repository IDs. |
| `-r <repository id>` | Yes | Specifies a particular repository id. Only one of the following options may be specified at a particular time: `-i`, `-r`, `-s`, or `-poa`. |
| `-s <service name>` | Yes | Specifies a particular service name. Only one of the following options may be specified at a particular time: `-i`, `-r`, `-s`, or `-poa`. |
| `-o <object name>` | Yes | Specifies a particular object name. You can use this only if the interface name or repository id is included in the command statement. This option is not applicable when a -s or -poa argument is used. |
| `-poa <POA_name>` | Yes | Unregisters the POA registered using `oadutil reg -poa <POA_name>`. |
| `-host <host name>` | No | Specifies the host name where the OAD is running. |
| `-verbose` | No | Enables verbose mode, causing messages to be output to stdout. |
| `-version` | No | Prints the version of this tool. |

### Unregistration example

The `oadutil unreg` utility unregisters one or more VisiBroker ORB objects from these three locations:

- Object Activation Daemon
- Implementation repository
- Smart Agent

The following is an example of how to use the `oadutil unreg` command. It unregisters the implementation of the `Bank::AccountManager` named `MyBank` from the local OAD.

```
oadutil unreg -i Bank::AccountManager -o MyBank
```

## Unregistering with the OAD operations

An object's implementation can use any one of the operations or attributes in the OAD interface to unregister a VisiBroker ORB object.

- `unreg_implementation(in CORBA::RepositoryId repId, in string object_name)`
- `unreg_interface(in CORBA::RepositoryId repId)`
- `unregister_all()`
- `attribute boolean destroy_on_unregister()`

| Operation | Description |
|---|---|
| `unreg_implementation()` | Use this operation when you want to un-registered implementations using a specific repository id and object name. This operation terminates all processes currently implementing the specified repository id and object name. |
| `unreg_interface()` | Use this operation when you want to un-registered implementations by using a specific repository id only. This operation terminates all processes currently implementing the specified repository id. |
| `unregister_all()` | Use this operation to un-registered all implementations. Unless `destroyActive` is set to `true`, all active implementations continue to execute. For backward compatibility, `unregister_all()` is **not** destructive; it is equivalent to invoking `unregister_all_destroy(false)`. |
| `destroy_on_unregister` | Use this attribute to destroy any spawned processes on unregistration of the relevant implementation. The default value is `false`. |

The following is an example of an OAD unregistered operation:

```
module Activation {
...
  interface OAD {
      ...
    void unreg_implementation(in CORBA::RepositoryId repId,
      in string object_name)
      raises(NotRegistered);
      ...
  }
}
```

## Displaying the contents of the Implementation Repository

You can use the `oadutil` tool to list the contents of a particular Implementation Repository. For each implementation in the repository the `oadutil` tool lists all the object instance names, the path name of the executable program, the activation mode and the reference data. Any arguments or environment variables that are to be passed to the executable program are also listed.

## IDL interface to the OAD

The OAD is implemented as a VisiBroker ORB object, allowing you to create a client program that binds to the OAD and uses its interface to query the status of objects that have been registered. The sample below shows the IDL interface specification for the OAD.

```
module Activation
{
   enum state {
      ACTIVE,
      INACTIVE,
      WAITING_FOR_ACTIVATION
   };
   struct ObjectStatus {
      long unique_id;
      State activation_state;
      Object objRef;
   };
   typedef sequence<ObjectStatus> ObjectStatusList;
   struct ImplementationStatus {
      extension::CreationImplDef impl;
      ObjectStatusList status;
   };
   typedef sequence<ImplementationStatus> ImplStatusList;
   exception DuplicateEntry {};
   exception InvalidPath {};
   exception NotRegistered {};
   exception FailedToExecute {};
   exception NotResponding {};
   exception IsActive {};
   exception Busy {};
   interface OAD {
      Object reg_implementation(   in extension::CreationImplDef impl)
               raises (DuplicateEntry, InvalidPath);
      extension::CreationImplDef get_implementation(
               in CORBA::RepositoryId repId,
               in string object_name)
               raises ( NotRegistered);
      void change_implementation(in extension::CreationImplDef old_info,
               in extension::CreationImplDef new_info)
                raises (NotRegistered,InvalidPath,IsActive);
      attribute boolean destroy_on_unregister;
      void unreg_implementation(in CORBA::RepositoryId repId,
               in string object_name)
               raises ( NotRegistered );
      void unreg_interface(in CORBA::RepositoryId repId)
               raises ( NotRegistered );
      void unregister_all();
      ImplementationStatus get_status(in CORBA::RepositoryId repId,
               in string object_name)
               raises ( NotRegistered);
      ImplStatusList get_status_interface(in CORBA::RepositoryId repId)
                                           raises (NotRegistered);
      ImplStatusList get_status_all();
};
```

20

# 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 (DII)).

Reading this section will enable you to create an Interface Repository and access it with VisiBroker utilities or with your own code.

## 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 Location Service, described in “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 `irep` program, which is the IR server (implementation). You can update or populate an Interface Repository with the VisiBroker `idl2ir` 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-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 VisiBroker ORB can look up the interface in the IR. If no Interface Repository is available, or if the IR that the VisiBroker 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 Interface Repository server is called `irep`, and is located in the `<install_dir>/bin` directory. The `irep` program runs as a daemon. You can register `irep` with the Object Activation Daemon (OAD) as you would any object implementation. The `oadutil` tool requires the object ID, for example, `IDL:org.omg/CORBA/Repository:2.3` (as opposed to an interface name such as `CORBA::Repository`).

**Note**

The irep server needs a rollback file to keep its internal data consistent. The file is created if it does not already exist, for example when launching the irep server for the first time. The IRepName specified in the command line is used to make up the name of the rollback file. Make sure that the name contains only valid file system characters based on your platform. If the specified name contains directory locations that do not exist, they will be automatically created.

### 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 <driver_options> <other_options> <IRepName> [file.idl]
```

The syntax for creating an Interface Repository in the `irep` is described in the following table:

| Syntax | Description |
|---|---|
| `IRepName` | Specifies the instance name of the Interface Repository. Clients can bind to this Interface Repository instance by specifying this name. |
| `file.idl` | Specifies the IDL file whose contents `irep` will load into the Interface Repository it creates and will store the IR contents into when it exits. If no file is specified, `irep` creates an empty Interface Repository. |

The `irep` arguments are defined in the following table. You may also use the driver options defined in "General options" .

| Argument | Description |
|---|---|
| `-D, -define foo[=bar]` | Define a preprocessor macro, optionally with value. |
| `-I, -include <dir>` | Specify additional directory for #include searching. |
| `-P, -no_line_directives` | Do not emit #line directives from preprocessor. The default is `off`. |
| `-H, -list_includes` | Display #included file names as they are encountered. The default is `off`. |
| `-C, -retain_comments` | Retain comments in preprocessed output. The default is `off`. |
| `-U, -undefine foo` | Undefine a preprocessor macro. |
| `-[no_]idl_strict` | Strict OMG-standard interpretation of IDL source. The default is `off`. |
| `-[no_]warn_unrecognized_pragmas` | Warn if a #pragma is not recognized. The default is `on`. |
| `-[no_]back_compat_mapping` | Use mapping that is compatible with VisiBroker 3.x. |
| `-h, -help, -usage, -?` | Print this usage information. |
| `-version` | Display software version numbers. |
| `-install <service name>` | Install as a NT service. |
| `-remove <service name>` | Uninstall this NT service. |

The following example shows how an Interface Repository named `TestIR` can be created from a file called `Bank.idl`.

```
irep TestIR Bank.idl
```

## Viewing the contents of the Interface Repository

You can view the contents of the Interface Repository with either the VisiBroker `ir2idl` utility, or the VisiBroker 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 |
|---|---|
| `-irep <IRname>` | Directs the program to bind to the Interface Repository instance named IRname. If the option is not specified, it binds to any Interface Repository returned by the Smart Agent. |

## Updating an Interface Repository with idl2ir

You can update an Interface Repository with the VisiBroker `idl2ir` utility, which is an IR client. The syntax for the `idl2ir` utility is:

```
idl2ir [arguments] <idl_file_list>
```

The following example shows how the `TestIR` Interface Repository would be updated with definitions from the `Bank.idl` file.

```
idl2ir -irep TestIR -replace Bank.idl
```

Entries in an Interface Repository cannot be removed using the `idl2ir` or `irep` utilities. To remove an item:

- Exit or quit the `irep` program.
- Edit the IDL file named in the `irep` command line.
- 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.

**Note**

If you wish to remove all entries in the Interface Repository, you can replace the contents with a new empty IDL file. For example, using an IDL file named `Empty.idl`, you could run the following command:

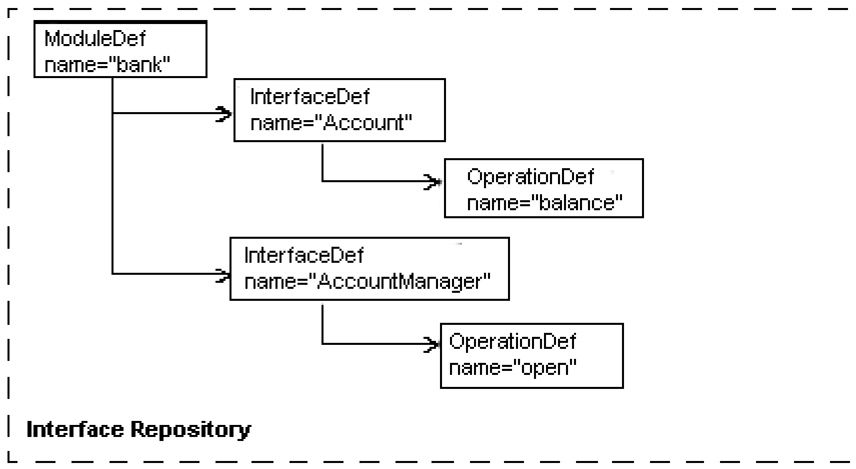
```
idl2ir -irep TestIR -replace Empty.idl
```

## 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 example IDL file shown below would translate to a hierarchy of objects in an Interface Repository.

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

**Figure 20.1** Interface repository object hierarchy for Bank.idl



The `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.

| 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 `IRObject`. A RepositoryID contains three components, separated by colon (`:`) delimiters. The first component is `IDL:` and the last is a version number such as `:1.0`. The second component is a sequence of identifiers separated by slash (`/`) 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

The following table 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.

| Object type | Description |
|---|---|
| `Repository` | Represents the top-level module that contains all other objects. |
| `ModuleDef` | Represents an IDL module declaration that can contain `ModuleDefs`, `InterfaceDefs`, `ConstantDefs`, `AliasDefs`, `ExceptionDefs`, and the IR equivalents of other IDL constructs that can be defined in IDL modules. |
| `InterfaceDef` | Represents an IDL interface declaration and contain `OperationDefs`, `ExceptionDefs`, `AliasDefs`, `ConstantDefs`, and `AttributeDefs`. |
| `AttributeDef` | Represents an IDL attribute declaration. |
| `OperationDef` | 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. |
| `ConstantDef` | Represents an IDL constant declaration. |
| `ExceptionDef` | Represents an IDL exception declaration. |
| `ValueDef` | Represents a valuetype definition containing lists of constants, types, valuemembers, exceptions, operations, and attributes. |
| `ValueBoxDef` | Represents a simple boxed valuetype of another IDL type. |
| `ValueMemberDef` | Represents a member of the valuetype. |
| `NativeDef` | Represents a native definition. Users can not define their own natives. |
| `StructDef` | Represents an IDL structure declaration. |
| `UnionDef` | Represents an IDL union declaration. |
| `EnumDef` | Represents an IDL enumeration declaration. |
| `AliasDef` | Represents an IDL typedef declaration. Note that the IR `TypedefDef` interface is a base interface that defines common operations for `StructDefs`, `UnionDefs`, and others. |
| `StringDef` | Represents an IDL bounded string declaration. |
| `SequenceDef` | Represents an IDL sequence declaration. |
| `ArrayDef` | Represents an IDL array declaration. |
| `PrimitiveDef` | Represents an IDL primitive declaration: `null`, `void`, `long`, `ushort`, `ulong`, `float`, `double`, `boolean`, `char`, `octet`, `any`, `TypeCode`, `Principal`, `string`, `objref`, `longlong`, `ulonglong`, `longdouble`, `wchar`, `wstring`. |

### 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 the table above). The following table summarizes these widely inherited interfaces. For more information on the other methods for these interfaces, see the VisiBroker *Programmer's Reference*.

| Interface | Inherited by | Principal query methods |
|---|---|---|
| `IRObject` | All IR objects including `Repository` | `def_kind()` returns an IR object's definition kind, for example, module or interface `destroy()` destroys an IR object |
| `Container` | IR objects that can contain other IR objects, for example, module or interface | `lookup()` looks up a contained object by name `contents()` lists the objects in a `Container` `describe_contents()` describes the objects in a `Container` |
| `Contained` | IR objects that can be contained in other objects, that is, `Containers` | `name()` name of this object `defined_in()` `Container` that contains an object `describe()` describes an object `move ()` 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 below. A complete description of this interface can be found in the *Programmer's Reference*.

**Note**

A program that uses an Interface Repository must be compiled with the-`D_VIS_INCLUDE_IR` flag.

```
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);
      ...
   };
   ...
};
```

## Interface Repository example program

This section describes a simple Interface Repository example which contains an `AccountManager` interface to create an account and (re)open an account. This example code can be found in the following directory:

```
<install_dir>\Vbroker\examples\ir
```

At the initialization time the `AccountManager` implementation bootstraps the Interface Repository definition for the managed Account interface. This exposes the additional operation that has been already implemented by this particular `Account` implementation to the clients. The clients now can access all known operations (which are described in IDL) and, additionally, they can verify with the Interface Repository support for other operations and invoke them. The example illustrates how we can manage Interface Repository definition objects and how to introspect remote objects using the Interface Repository.

Before this program can be tested, the following conditions should exist:

- OSAgent should be up and running. For more information, see "Using the Smart Agent."
- Interface repository should be started using irep. For more information, see "Creating and viewing an Interface Repository with 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. For more information, see "Updating an Interface Repository with idl2ir".
- Start the client program.

Looking up an interface's operations and attributes in an IR:

```
/* PrintIR.C */
#ifndef _VIS_INCLUDE_IR
#define _VIS_INCLUDE_IR
#endif
#include "corba.h"
#include "strvar.h"
int main(int argc, char *argv[]) {
   try {
      if (argc != 2) {
         cout << "Usage: PrintIR idlName" << endl;
         exit(1);
      }
      CORBA::String_var idlName = (const char *)argv[1];
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      CORBA::Repository_var rep = CORBA::Repository::_bind();
      CORBA::Contained_var contained = rep->lookup(idlName);
      CORBA::InterfaceDef_var intDef = CORBA::InterfaceDef::_narrow(contained);
      if (intDef != CORBA::InterfaceDef::_nil()) {
         CORBA::InterfaceDef::FullInterfaceDescription_var fullDesc = intDef-
>describe_interface();
         cout << "Operations:" << endl;
         for(CORBA::ULong i = 0; i < fullDesc->operations.length(); i++)
            cout << "  " << fullDesc->operations[i].name << endl;
            cout << "Attributes:" << endl;
            for(i = 0; i < fullDesc->attributes.length(); i++)
               cout << "  " << fullDesc->attributes[i].name << endl;
      } else
         cout << "idlName is not an interface: " << idlName << endl;
   } catch (const CORBA::Exception& excep) {
      cerr << "Exception occurred ..." << endl;
      cerr << excep << endl;
      exit(1);
   }
   return 0;
}
```

# 21

# 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.

## 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 does not 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.

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 the *target reference,* an object reference representing the CORBA object. 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`. 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 the VisiBroker ORBs do not contain type information. By specifying a place holder result type in the `Request`, you give the VisiBroker 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.

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. `TypeCode`s for IDL constructed types, such as `structs`, `unions`, and `typedefs`, have to be constructed. Such `TypeCode`s 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 VisiBroker 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 `Request` is represented a single named value.

## Options for sending requests

Once you create and populate 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 VisiBroker 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 VisiBroker 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 `Request`s 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 VisiBroker ORB's `get_next_response` method. To do the same thing without risking blocking, use the VisiBroker 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.
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.

## Example programs for using the DII

Several example programs that illustrate the use of the DII are included in the following directory:

```
<install_dir>/examples/Vbroker/bank_dynamic
```

These example programs are used to illustrate DII concepts in this section.

Compile these example programs with the `VIS_INCLUDE_IR` flag, and add the typecode generation option.

# 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.

The code sample below shows how your client program can use the `bind` method offered by the VisiBroker ORB object to bind to any object by specifying its name. This method returns a generic `CORBA::Object` .

```
...
CORBA::Object_var account;
try {
   // initialize the ORB.
   CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
} catch (const CORBA::Exception& e)
   cout << "Failure during ORB_init" << endl;
   cout << e << endl;
}
...
try {
   // Request ORB to bind to object supporting the account interface.
```

```
      account = orb->bind("IDL:Account:1.0");
   } catch (const CORBA::Exception& excep)
      cout << "Error binding to account" << endl;
      cout << excep << endl;
   }
   cout << "Bound to account object" << endl;
   ...
```

## 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.

```
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 sample shows the methods offered by the `CORBA::Object` class.

```
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

The code sample in "Example of creating a Request object" 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. Its result type is initialized as an Object reference type via a call to set_return_type method. After a call has been made, the return value is extracted with the result's call to the result method. 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. The following code sample is an example of creating a request object.

```
...
CORBA::NamedValue_ptr result;
CORBA::Any_ptr resultAny;
CORBA::Request_var req;
CORBA::Any customer;
...
try {
   req = account->_request("balance");

   // Create argument to request
   customer <<= (const char *) name;
   CORBA::NVList_ptr arguments = req->arguments();
   arguments->add_value("customer", customer, CORBA::ARG_IN);
   // Set result
   result = req->result();
   resultAny = result->value();
   resultAny->replace(CORBA::_tc_float, &result);
} catch(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.

```
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. The following code sample is an example of the 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. The following code sample is an example of the 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.

| Method | Description |
|---|---|
| `name` | Returns a pointer to the name of the item that you can then use to initialize the name. |
| `value` | Returns a pointer to an `Any` object representing the item's value that you can then use to initialize the value. For more information, see the "Passing type safely with the Any class" section. |
| `flags` | 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 VisiBroker ORB should make a copy of the argument and leave the caller's memory intact. Flags are `ARG_IN`, `ARG_OUT`, and `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 methods are provided to read and write the object from and to a stream. The following code sample is an example of defining this 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 wit 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 code sample is an example of the `TypeCode` class.

The following table shows the kinds and parameters for the `TypeCode` objects.

| Kind | Parameter list |
|---|---|
| tk_abstract_interface | repository_id, interface_name |
| tk_alias | repository_id, alias_name, TypeCode |
| tk_any | None |
| tk_array | length, TypeCode |
| tk_boolean | None |
| tk_char | None |
| tk_double | None |
| tk_enum | repository_id, enum-name, enum-id$^1$, enum-id$^2$, ... enum-id$^n$ |
| tk_except | repository_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 | repository_id, interface_id |
| tk_octet | None |
| tk_Principal | None |
| tk_sequence | TypeCode, maxlen |
| tk_short | None |
| tk_string | maxlen-integer |
| tk_struct | repository_id, struct-name, {member$^1$, TypeCode$^1$}, {member$^n$, TypeCode$^n$} |
| tk_TypeCode | None |
| tk_ulong | None |
| tk_ulonglong | None |
| tk_union | repository_id, union-name, switch TypeCode,{label-value$^1$, member-name$^1$, TypeCode$^1$}, {label$^l$-value$^n$, member-name$^n$, TypeCode$^n$} |
| tk_ushort | None |
| tk_value | repository_id, value_name, boxType |
| tk_value_box | repository_id, value_name, typeModifier, concreteBase, members |
| tk_void | None |
| tk_wchar | None |
| tk_wstring | None |

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, as discussed in “Creating and initializing a request”, 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. The following code sample shows how to send a request with invoke.

```
try {
   ...
   // Create request that will be sent to the account object
   request = account->_request("balance");
   // Set the result type
   request->set_return_type(CORBA::_tc_float);
```

```
      // Execute the request to the account object
    request->invoke();
      // Get the return balance
      CORBA::Float balance;
      CORBA::Any& balance_result = request->return_value();
      balance_result >>= balance;
      // Print out the balance
      cout << "The balance in " << name << "'s account is $" << balance << endl;
   } catch(const CORBA::Exception& e) {
      cerr << e << endl;
      return 1;
   }
   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 codes show how these methods are used. The following sample shows you how to use the `send_deferred` and `poll_response` methods to send a deferred DII request.

```
   ...
   try {
      // Create request that will be sent to the manager object
      CORBA::Request_var request = manager->_request("open");
      // 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
      request->set_return_type(CORBA::_tc_Object);
      // Creation of a new account can take some time
      // Execute the deferred request to the manager object
    request->send_deferred();
      VISPortable::vsleep(1);
      while (!request->poll_response()) {
         cout << "  Waiting for response..." << endl;
         VISPortable::vsleep(1);  // Wait one second between polls
      }
    request->get_response();
      // Get the return value
      CORBA::Object_var account;
      CORBA::Any& open_result = request->return_value();
      open_result >>= CORBA::Any::to_object(account.out());
   ...
   }
```

## 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 VisiBroker 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.

The following code sample 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.

```
...
// Create request to balance
try {
   req1 = account->_request("balance");
   // Create argument to request
   customer1 <<= (const char *) "Happy";
   CORBA::NVList_ptr arguments = req1->arguments();
   arguments->add_value("customer", customer1, CORBA::ARG_IN);
   // Set result
   ...
} catch(const CORBA::Exception& excep) {
   cout << "Error while creating request" << endl;
   cout << excep << endl;
}
// Create request2 to slowBalance
try {
   req2 = account->_request("slowBalance");
   // Create argument to request
   customer2 <<= (const char *) "Sleepy";
   CORBA::NVList_ptr arguments = req2->arguments();
   arguments->add_value("customer", customer2, CORBA::ARG_IN);
   // Set result
   ...
} catch(const CORBA::Exception& excep) {
   cout << "Error while creating request" << endl;
   cout << excep << endl;
}
// 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
try {
   orb->send_multiple_requests_deferred(reqseq);
   cout << "Send multiple deferred calls are made..." << endl;
} catch(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 VisiBroker 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 VisiBroker 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. The following code sample shows an example of receiving multiple requests.

VisiBroker 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

One source of the information needed to populate a DII Request object is an interface repository (IR) (see "Using Interface Repositories"). The following example uses an interface repository to get obtain the parameters of an operation. Note that the example, atypical of real DII applications, 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.

- Binds to any Account object.
- Looks up the Account's balance method in the IR and builds an operation list from the IR OperationDef.
- Creates argument and result components and passes these to the _create_request method. Note that the balance method does not return an exception.
- Invokes the Request, extracts and prints the result.

```
// acctdii_ir.C
// This example illustrates IR and DII
#include <iostream.h>
#include "corba.h"
int main(int argc, char* const* argv) {
   CORBA::ORB_ptr orb;
   CORBA::Object_var account;
   CORBA::NamedValue_var result;
   CORBA::Any_ptr resultAny;
   CORBA::Request_var req;
   CORBA::NVList_var operation_list;
   CORBA::Any customer;
   CORBA::Float acct_balance;
try {
    // use argv[1] as the account name, or a default.
    CORBA::String_var name;
    if (argc == 2)
       name = (const char *) argv[1];
    else
       name = (const char *) "Default Name";
   try {
      // Initialize the ORB.
      orb = CORBA::ORB_init(argc, argv);
   } catch(const CORBA::Exception& excep) {
      cout << "Failure during ORB_init" << endl;
      cout << excep << endl;
      exit(1);
   }
   cout << "ORB_init succeeded" << endl;
   // Unlike traditional binds, this bind is called off of "orb"
   // and returns a generic object pointer based on the interface name
   try {
 account = orb->bind("IDL:Account:1.0");
   } catch(const CORBA::Exception& excep) {
      cout << "Error binding to account" << endl;
      cout << excep << endl;
      exit(2);
   }
   cout << "Bound to account object" << endl;
   // Obtain Operation Description for the "balance" method of
   // the Account
   try {
      CORBA::InterfaceDef_var intf = account->_get_interface();
      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;
         exit(3);
      }
 CORBA::Contained_var oper_container = intf->lookup("balance");
 CORBA::OperationDef_var oper_def =
      CORBA::OperationDef::_narrow(oper_container);
      orb->create_operation_list(oper_def, operation_list.out());

   } catch(const CORBA::Exception& excep) {
      cout << "Error while obtaining operation list" << endl;
      cout << excep << endl;
      exit(4);
   }
   // Create request that will be sent to the account object
   try {
      // Create placeholder for result
      orb->create_named_value(result.out());
      resultAny = result->value();
      resultAny->replace( CORBA::_tc_float, &result);
      // Set the argument value within the operation_list
 CORBA::NamedValue_ptr arg = operation_list->item(0);
 CORBA::Any_ptr anyArg = arg->value();
      *anyArg <<= (const char *) name;

      // Create the request
  account->_create_request(CORBA::Context::_nil(),

            "balance",
            operation_list,
            result,
            req.out(),
            0);

   } catch(const CORBA::Exception& excep) {
      cout << "Error while creating request" << endl;
      cout << excep << endl;
      exit(5);
   }
   // Execute the request
   try {
      req->invoke();
      CORBA::Environment_ptr env = req->env();
      if ( env->exception() ) {
         cout << "Exception occurred" << endl;
         cout << *(env->exception()) << endl;
         acct_balance = 0;
      } else  {
         // Get the return value;
 acct_balance = *(CORBA::Float *)resultAny->value();
      }
   } catch(const CORBA::Exception& excep) {
      cout << "Error while invoking request" << endl;
      cout << excep << endl;
      exit(6);
   }
   // Print out the results
   cout << "The balance in " << name << "'s account is $";
   cout << acct_balance << "." << endl;
} catch ( const CORBA::Exception& excep ) {
   cout << "Error occurred" << endl;
   cout << excep << endl;
}
```

# 22

# Using the Dynamic Skeleton Interface

This section describes how object servers can dynamically create object implementations at run time to service client requests.

## 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 VisiBroker 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 VisiBroker ORB object. Clients do not need to provide any special handling to communicate with an object implementation that uses the DSI.

The VisiBroker 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. However, 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:

1. When compiling your IDL use the `-type_code_inf` flag.
2. 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.
3. Declare and implement the `invoke` method, which the VisiBroker ORB will use to dispatch client requests to your object.

4 Register your object implementation (POA servant) with the POA manager as the default servant.

### Example program for using the DSI

An example program that illustrates the use of the DSI is located in the following directory:

```
<install_dir>/examples/Vbroker/basic/bank_dynamic
```

This example is used to illustrate DSI concepts in this section. The `Bank.idl` file, shown below, illustrates the interfaces implemented in this 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.

```
class PortableServer::DynamicImplementation : public virtual
PortableServer::ServantBase {
   public:
      virtual void invoke(PortableServer::ServerRequest_ptr request) = 0;
...
};
```

### Example of designing objects for dynamic requests

The code sample below 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 VisiBroker ORB will call the `invoke` method to pass client operation requests to the implementation in the form of `ServerRequest` objects.

The code sample below shows the `Account` class constructor and `_primary_interface` function.

```
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 dollars
            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;
         try {
            name = PortableServer::ObjectId_to_string(oid);
         } catch (const CORBA::Exception& e) {
            throw CORBA::OBJECT_NOT_EXIST();
         }
         // Ensure that the operation name is correct
         if (strcmp(request->operation(), "balance") != 0) {
            throw 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))
            throw CORBA::OBJECT_NOT_EXIST();
         CORBA::Any result;
         result <<= balance;
         request->set_result(result);
         cout << "Checked " << name << "'s balance: " << balance << endl;
      }
};
```

The following code sample shows the implementation of the `AccountManagerImpl` class that need to be implemented with the DSI. It is also derived from the `DynamicImplementation` class, which declares the `invoke` method. The VisiBroker ORB will call the `invoke` method to pass client operation requests to the implementation in the form of `ServerRequest` objects.

```
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)
              throw CORBA::BAD_OPERATION();
           // Fetch the input parameter
           char *name = NULL;
           try {
              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;
           } catch (const CORBA::Exception& e) {
              throw CORBA::BAD_PARAM();
           }
           // 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

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:

1 Replace all non-leading instances of the delimiter scope resolution operator (::) with a slash (/).

2 Add "IDL:" to the beginning of the string.

3 Add ":1.0" to the end of the string.

For example, this code sample shows an IDL interface name:

```
Bank::AccountManager
```

The resulting repository identifier looks like this:

```
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.

```
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 VisiBroker ORB. The memory for these arguments will be released by the VisiBroker 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 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.

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

The following 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 the "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'sarguments 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 VisiBroker ORB. This means that if an object implementation modifies an ARG_INOUT parameter in the NVList, the change will automatically be apparent to the VisiBroker ORB. This NVList should not be released by the caller.

An alternative to constructing the NVList for the input arguments is to use the VisiBroker ORB object'screate_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 the "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 in the following code sample, is almost identical to the original example in "Developing an example application with VisiBroker."

```
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
      // Get a reference to the root POA
      CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
      PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);
      // Get the POA Manager
      PortableServer::POAManager_var poaManager = rootPOA->the_POAManager();
      // Create the account POA with the right policies
      CORBA::PolicyList accountPolicies;
      accountPolicies.length(3);
      accountPolicies[(CORBA::ULong)0] =
      rootPOA->create_servant_retention_policy(PortableServer::NON_RETAIN);
      accountPolicies[(CORBA::ULong)1] =
      rootPOA->create_request_processing_policy(
      PortableServer::USE_DEFAULT_SERVANT);
      accountPolicies[(CORBA::ULong)2] =
      rootPOA->create_id_uniqueness_policy(
      PortableServer::MULTIPLE_ID);
      PortableServer::POA_var accountPOA =
            rootPOA->create_POA("bank_account_poa",
            poaManager,
            accountPolicies);
      // Create the account default servant
      PortableServer::Current_var current =
PortableServer::Current::_instance();
      AccountImpl accountServant(current, accountPOA);
      accountPOA->set_servant(&accountServant);
      // Create the manager POA with the right policies
      CORBA::PolicyList managerPolicies;
      managerPolicies.length(3);
      managerPolicies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy(
      PortableServer::PERSISTENT);
      managerPolicies[(CORBA::ULong)1] =
      rootPOA->create_request_processing_policy(
      PortableServer::USE_DEFAULT_SERVANT);
      managerPolicies[(CORBA::ULong)2] =
      rootPOA->create_id_uniqueness_policy(
      PortableServer::MULTIPLE_ID);
      PortableServer::POA_var managerPOA = rootPOA-
>create_POA("bank_agent_poa",
      poaManager,
      managerPolicies);
      // Create the manager default servant
      AccountManagerImpl managerServant(&accountServant);
      managerPOA->set_servant(&managerServant);
      // Activate the POA Manager
      poaManager->activate();
      cout << "AccountManager is ready" << endl;
      // Wait for incoming requests
      orb->run();
   } catch(const CORBA::Exception& e) {
      cerr << e << endl;
      return 1;
   }
   return 0;
}
```

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.”

23

# Using Portable Interceptors

This section provides an overview of Portable Interceptors. Several Portable Interceptor examples are discussed as well as the advanced features of Portable Interceptor factories.

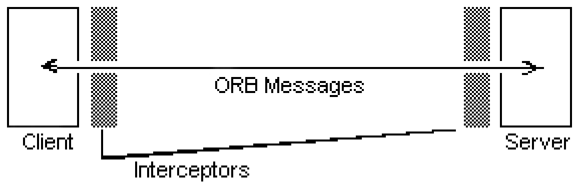
For a complete description of Portable Interceptors, refer to the OMG Final Adopted Specification, ptc/2001–04–03, Portable Interceptors.

## Portable Interceptors overview

The VisiBroker 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.

**Figure 23.1** How Interceptors work



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 can develop an application that monitors the activity of various servers and performs load balancing.

## Types of interceptors

There are two types of interceptors supported by the VisiBroker ORB.

| Portable Interceptors | VisiBroker Interceptors |
|---|---|
| An OMG standardized feature that allows writing of portable code as interceptors, which can be used with different ORB vendors. | VisiBroker-specific interceptors. For more information, go to the "Using VisiBroker Interceptors." |

## Types of Portable Interceptors

The two kinds of Portable Interceptors defined by the OMG specification are: *Request Interceptors* and *IOR interceptors*.

| Request Interceptors | IOR interceptor |
|---|---|
| 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*. | 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 additional information on using both Portable Interceptors and VisiBroker Interceptors, see the "Using VisiBroker Interceptors."

See also *VisiBroker for Java APIs*, and the "Portable Interceptor interfaces and classes for C++" chapter of the *VisiBroker for C++ API Reference*.

## 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`

### Interceptor class

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;
}
```

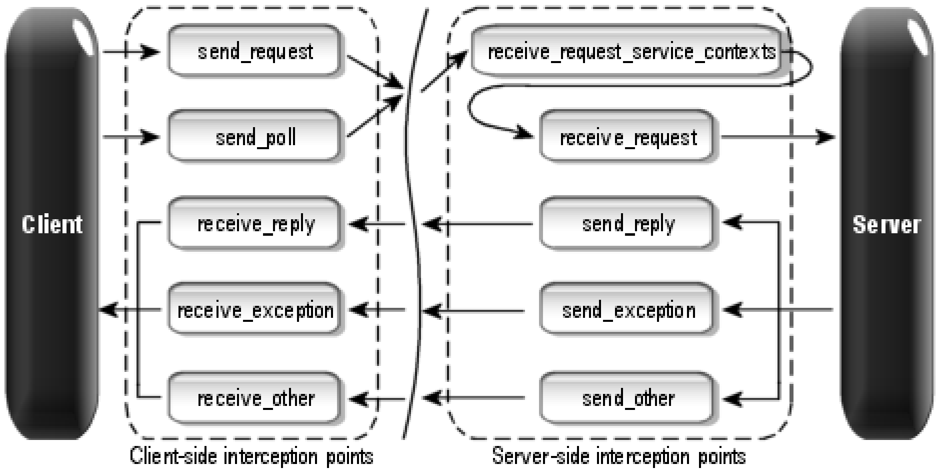
### Request Interceptor

A *request interceptor* is used to intercept the 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 23.2** Request Interception points



For more detail information on Request Interceptors, see *VisiBroker for Java APIs* and "Portable Interceptor interfaces and classes for C++" chapter of the *VisiBroker for C++ API Reference*.

## ClientRequestInterceptor

`ClientRequestInterceptor` has its interception points implemented on the client-side. There are five interception points defined in `ClientRequestInterceptor` by the OMG as shown in the following table:

| Interception Points | Description |
|---|---|
| `send_request` | Lets a client-side Interceptor query a request and modify the service context before the request is sent to the server. |
| `send_poll` | Lets a client-side Interceptor query a request during a Time-Independent Invocation (TII)[1] polling get reply sequence. |
| `receive_reply` | Lets a client-side Interceptor query the reply information after it is returned from the server and before the client gains control. |
| `receive_exception` | Lets a client-side Interceptor query the exception's information, when an exception occurs, before the exception is sent to the client. |
| `receive_other` | Lets a client-side Interceptor query the information which is available when a request result other than normal reply or an exception is received. |

[1] TII is not implemented in the VisiBroker ORB. As a result, the `send_poll( )` interception point will never be invoked.

For more information on each interception point, see *VisiBroker for Java APIs* and "Portable Interceptor interfaces and classes for C++" chapter of the *VisiBroker for C++ API Reference*.

```
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;
};
```

### Client-side rules

The following are the client-side rules:

- 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 | `send_request` is followed by `receive_reply`; a start point is followed by an end point |
| Retries | `send_request` is followed by `receive_other`; 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 following table shows the `ServerRequestInterceptor` Interception points.

| Interception Points | Description |
|---|---|
| `receive_request_service_contexts` | Lets a server-side Interceptor get its service context information from the incoming request and transfer it to `PortableInterceptor::Current`'s slot. |
| `receive_request` | Lets a server-side Interceptor query request information after all information, including operation parameters, is available. |
| `send_reply` | 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. |
| `send_exception` | 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. |
| `send_other` | 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 *VisiBroker for Java APIs* and “Portable Interceptor interfaces and classes for C++” chapter of the *VisiBroker for C++ API 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;
};
```

### Server-side rules

The following are the server-side rules:

- 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` 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 *VisiBroker for Java APIs* and “Portable Interceptor interfaces and classes for C++” chapter of the *VisiBroker for C++ API Reference*.

```
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 (PI) 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 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:

```
ORB->resolve_initial_references("PICurrent");
```

`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

`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`. For more information, see "CodecFactory".

The `Codec` class:

```
class _VISEXPORT Codec
{
   public:
      virtual CORBA::OctetSequence* encode(const CORBA::Any& _data) = 0;
      virtual CORBA::Any* decode(const CORBA::OctetSequence& _data) = 0;
      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 with a call to:

```
ORB->resolve_initial_references("CodecFactory")
```

The `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:

1 The Interceptor must be inherited from one of the following Interceptor interfaces:
   - `ClientRequestInterceptor`
   - `ServerRequestInterceptor`
   - `IORInterceptor`

1 The Interceptor implements one or more interception points that are available to the Interceptor.

2 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: Creating a PortableInterceptor

```
#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.

The `ORBInitializer` class:

```
class _VISEXPORT ORBInitializer
{
   public:

      virtual void pre_init(ORBInitInfo_ptr _info) = 0;
      virtual void post_init(ORBInitInfo_ptr _info) = 0;
};
```

The `ORBInitInfo` class:

```
class _VISEXPORT ORBInitInfo
{
   public:
      virtual CORBA::StringSequence* arguments() = 0;
      virtual char* orb_id() = 0;
      virtual IOP::CodecFactory_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 an `ORBInitializer`, the 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
2. makes an instantiating `ORB_Init( )` call with a new ORB identifier to produce a new ORB.

During `ORB.init( )`:

1. these ORB properties which begin with `org.omg.PortableInterceptor.ORBInitializerClass` are collected.
2. the `<Service>` portion of each property is collected.
3. an object is instantiated with the `<Service>` string as its class name.
4. the `pre_init( )` and `post_init( )` methods are called on that object.
5. if there is any exception, the ORB ignores them and proceeds.

**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: Registering ORBInitializer

A client-side monitoring tool written by company ABC may have the following `ORBInitializer` implementation:

```
#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_service);
```

The following command may be used to run a program called `MyApp` using this monitoring service:

```
java -
Dorg.omg.PortableInterceptor.ORBInitializerClass.com.abc.Monitoring.MonitoringS
ervice MyApp
```

## VisiBroker extensions to Portable Interceptors

### POA scoped Server Request Interceptors

Portable Interceptors specified by OMG are scoped globally. VisiBroker 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.

The 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 Portable Interceptors implementation

The following are limitations of the Portable Interceptor implementation in VisiBroker.

### ClientRequestInfo limitations

- `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` (for example, IDL is compiled with `-typecode_info` and linked into program), otherwise `CORBA::UNKNOWN` is always returned.

### ServerRequestInfo limitations

- `arguments(), result()`, are only available for DSI invocations. For more information, see "Using the Dynamic Skeleton Interface."
- `exceptions(), contexts(), operation_context()`: not available, `CORBA::NO_RESOURCES` thrown.
- `sending_exception()`: available only if typecode info is available (for example, IDL is compiled with `-typecode_info` and linked into program), otherwise `CORBA::UNKNOWN` is always returned.

## Portable Interceptors 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 Java and C++. For more information on the definition of each interface, see *VisiBroker for Java APIs* and "Portable Interceptor interfaces and classes for C++" chapter of the *VisiBroker for C++ API Reference*. We also recommend that developers who want to make use of Portable Interceptors read the chapter on Portable Interceptors in the most recent CORBA specification.

The Portable Interceptors examples are located in the following directory:

```
<install_dir>/examples/Vbroker/pi
```

Each example is associated with one of the following directory names to better illustrate the objective of that example.

- `client_server`
- `chaining`

## Example: client_server

This section provides a detailed description, explanation, the compilation procedure, and the execution or deployment of the examples in `client_server`.

### Objective of example

This example demonstrates how easily 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>/examples/Vbroker/basic/bank_agent
```

Portable Interceptors have been added to the entire example during runtime configuration. This provides developers, who are familiar with VisiBroker Interceptors, a fast way of coding between VisiBroker Interceptors and OMG specific Portable Interceptors.

### Importing required packages

To use Portable Interceptor interfaces, inclusion of the related packages or header files is required.

**Note**

If you are using any Portable Interceptors exceptions, such as, `DuplicateName` or `InvalidName`, the `ORBInitInfoPackage` is optional.

Required header files for using Portable Interceptor are:

```
#include "PortableInterceptor_c.hh"
#include "IOP_c.hh"
```

To load a client-side request interceptor, a class that uses the `ORBInitializer` interface 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
```

**Note**

Each 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 “Understanding 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 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

```
public: void pre_init(PortableInterceptor::ORBInitInfo_ptr _info) {
            SampleClientInterceptor *interceptor = new SampleClientInterceptor;
               try {
                  _info->add_client_request_interceptor(interceptor);
                  ...
```

According to the OMG specification, the required application registers the respective interceptors through the method `register_orb_initializer`. For more information, see “Developing the Client and Server Application”.

VisiBroker provides an optional way of registering these interceptors through a dynamic link library (DLL). The advantage of using this method of registering is that the applications do not require changing any code, only changing the way they are executed. With an additional option during execution, the interceptors are registered and executed. The option is similar to 4.x Interceptors:

```
vbroker.orb.dynamicLibs=<DLL filename>
```

where `<DLL filename>`is the filename of the dynamic link library (extension `.SO` for UNIX or `.DLL` for Windows). To load more than one DLL file, separate each filename with a comma (`","`), for example:

**Windows**

```
vbroker.orb.dynamicLibs=a.dll,b.dll,c.dll
```

**UNIX**

```
vbroker.orb.dynamicLibs=a.so,b.so,c.so
```

In order to load the interceptor dynamically, the `VISInit` interface is used. This is similar to the one used in VisiBroker Interceptors. For more information, see “Using VisiBroker Interceptors.” The registration of each interceptor loader is similar within the `ORB_init` implementation.

Registration of client-side `ORBInitializer` through DLL 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;

         try {
               _info->add_client_request_interceptor(interceptor);

               _interceptors_installed = 1;
                 cout << "=====>SampleClientLoader: Interceptors loaded."
                      << endl;
         }
         catch(PortableInterceptor::ORBInitInfo::DuplicateName &e) {
            cout << "=====>SampleClientLoader: "
                << e.name << " already installed!" << endl;
         }
         catch(...) {
            cout << "=====>SampleClientLoader: other exception occurred!"
                << endl;
             }
         }

         void post_init(PortableInterceptor::ORBInitInfo_ptr _info) {
   }
};

#if defined( DLL_COMPILE )

class VisiClientLoader : VISInit
{
private:
      static VisiClientLoader _instance;
      short int _bind_interceptors_installed;

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;

         try {
               SampleClientLoader *client = new SampleClientLoader();
               PortableInterceptor::register_orb_initializer(client);
```

```
              _bind_interceptors_installed = 1;
          }
          catch(const CORBA::Exception& e)
          {
             cerr << e << endl;
          }
    }
};
// 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:

```
public void pre_init(PortableInterceptor::ORBInitInfo_ptr _info) {
             SampleServerInterceptor *interceptor = new
SampleServerInterceptor;
             try {
                 _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 loading through DLL:

```
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:

```
// 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();

   try {
         _info->add_server_request_interceptor(interceptor);

 _interceptors_installed = 1;

         cout << "=====>SampleServerLoader: Interceptors loaded."
             << endl;
   }
   catch(PortableInterceptor::ORBInitInfo::DuplicateName &e) {
      cout << "=====>SampleServerLoader: "
          << e.name << " already installed!" << endl;
   }
   catch(...) {
      cout << "=====>SampleServerLoader: other exception occurred!"
          << endl;
      }
   }
 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;
 try {
           SampleServerLoader *server = new SampleServerLoader();
           PortableInterceptor::register_orb_initializer(server);
 _poa_interceptors_installed = 1;
      }
   catch(const 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 `ClientRequestInterceptor` interface for the request interceptor to work properly.

When the class implements the interface, the following five request interceptor methods are implemented regardless of any implementation:

- `send_request()`
- `send_poll()`
- `receive_reply()`
- `receive_exception()`
- `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.

```
// 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. For more information, go to the Using POAs, "Using servants and servant managers".

### 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:

```
// 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::ServerRequestInfo_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.

The client and server register the respective `ORBInitializer` classes through the `PortableInterceptor::register_orb_initalizer(<class_name>)` method, where `<class_name>` is the name of the class to be registered.

In the example, we also demonstrate another way of registering the interceptor classes as a dynamic link library (DLL). The advantage of registering it this way is that while changes in the execution are required, the application does not require any code changes.

**Note**

This is a VisiBroker proprietary way of registration. If you wish to have full compliance with OMG, do not use the following style.

If you choose to load the interceptor classes as a DLL (using a proprietary method of VisiBroker), no additional code is required for the client and server applications. Notice that in the example a portion of the code is conditionalized out through a macro if DLL compilation and linking is not specified.

### Implementation of the client application

```
// Client.C

#include "Bank_c.hh"

// USE_STD_NS is a define setup by VisiBroker to use the std namespace
USE_STD_NS

#if !defined( DLL_COMPILE )
#include "SampleClientLoader.C"
#endif

int main(int argc, char* const* argv)
{
    try {
        // Instantiate the Loader *before* the orb initialization
        // if chose not to use DLL method of loading
        #if !defined( DLL_COMPILE )
        SampleClientLoader* loader = new SampleClientLoader;
        PortableInterceptor::register_orb_initializer(loader);
        #endif

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

        // 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);

        // use argv[1] as the account name, or a default.
        const char* name = argc > 1 ? argv[1] : "Jack B. Quick";

        // 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;
    }
    catch(const CORBA::Exception& e) {
        cerr << e << endl;
        return 1;
    }

    return 0;
}
```

## Implementation of the server application

```
// Server.C

#include "BankImpl.h"

// USE_STD_NS is a define setup by VisiBroker to use the std namespace
USE_STD_NS

#if !defined( DLL_COMPILE )
#include "SampleServerLoader.C"
#endif

int main(int argc, char* const* argv)
{
    try {
        // Instantiate an interceptor loader before initializing the orb:
        #if !defined( DLL_COMPILE )
        SampleServerLoader* loader = new SampleServerLoader();
        PortableInterceptor::register_orb_initializer(loader);
        #endif

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

        // 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
        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
        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;

        // Wait for incoming requests
        orb->run();
    }
    catch(const CORBA::Exception& e) {
        cerr << e << endl;
        return 1;
    }
    return 0;
}
```

## Compilation procedure

To compile the C++ example without using the VisiBroker propriety loading method, simply execute the following commands:

**Windows**

```
<install_dir>\examples\Vbroker\pi\client_server> nmake -f Makefile.cpp
```

**UNIX**

```
<install_dir>/examples/Vbroker/pi/client_server> make -f Makefile.cpp
```

To compile the C++ example using the VisiBroker propriety loading method, execute the following commands:

**Windows**

```
<install_dir>\examples\Vbroker\pi\client_server>
 nmake -f Makefile.cpp dll
```

**UNIX**

```
<install_dir>/examples/Vbroker/pi/client_server>
 make -f Makefile.cpp dll
```

## Execution or deployment of Client and Server Applications

To run the C++ example without using the VisiBroker proprietary loading method, start the server and client as follows:

#### Windows

```
<install_dir>\examples\Vbroker\pi\client_server> start Server (running under a
new command prompt window)
<install_dir>\examples\Vbroker\pi\client_server> Client John (using a given
name)
```

or

```
<install_dir>\examples\Vbroker\pi\client_server> Client (using a default name)
```

#### UNIX

Open two console shells:

```
<install_dir>/examples/Vbroker/pi/client_server> Server(in the first window)
<install_dir>/examples/Vbroker/pi/client_server> Client John (in the second
window, using a given name)
```

or

```
<install_dir>/examples/Vbroker/pi/client_server> Client (in the second window,
using the default name)
```

To run the C++ example using the VisiBroker proprietary loading method, start the server and client as follows:

#### Windows

```
<install_dir>\examples\Vbroker\pi\client_server> start Server -
Dvbroker.orb.dynamicLibs= SampleServerLoader.dll(running under a new command
prompt window)
<install_dir>\examples\Vbroker\pi\client_server> Client John -
Dvbroker.orb.dynamicLibs= SampleClientLoader.dll(using a given name)
```

or

```
<install_dir>\examples\Vbroker\pi\client_server> Client -
Dvbroker.orb.dynamicLibs= SampleClientLoader.dll (using a default name)
```

#### UNIX

Open two console shells:

```
<install_dir>/examples/Vbroker/pi/client_server> Server
-Dvbroker.orb.dynamicLibs=./SampleServerLoader.so (in the first window)
<install_dir>/examples/Vbroker/pi/client_server> Client
-Dvbroker.orb.dynamicLibs=./SampleClientLoader.so John (in the second window,
using a given name)
```

24

# Using VisiBroker Interceptors

This section provides an overview of the VisiBroker Interceptors (Interceptors) framework, walks through a Interceptor example, and describes some advanced features such as Interceptor factories and chaining Interceptors. This section also covers the expected behaviors when both Portable Interceptors and VisiBroker Interceptors are used in the same service.

## Interceptors overview

Similar to Portable Interceptors, VisiBroker Interceptors offers VisiBroker ORB services a mechanism to intercept normal flow of execution of the ORB. There are two kinds of VisiBroker Interceptors:

- *Client Interceptors* are system-level Interceptors which are called when a method is invoked on a client object.
- *Server Interceptors* are system-level Interceptors which are called when a method is invoked on a server object.

To use VisiBroker 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 is then notified by its manager when, for example, an object has had one of its methods invoked or its parameters marshalled or demarshalled.

An important difference between VisiBroker interceptors and Portable interceptors is that VisiBroker interceptors do not get invoked for co-located calls. Therefore, users have to make a cautious decision when choosing which interceptor to use.

**Note**

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 before it is processed on the server side, use object wrappers, described in “Using object wrappers.”

## 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.

- Client Interceptors:

  - BindInterceptor
  - ClientRequestInterceptor
- Server Interceptors:
  - POALifeCycleInterceptor
  - ActiveObjectLifeCycleInterceptor
  - ServerRequestInterceptor
  - IORCreationInterceptor
- ServiceResolver Interceptor

## Client Interceptors

There are currently two kinds of client Interceptor and their respective managers:

- BindInterceptor and BindInterceptorManager
- ClientRequestInterceptor and ClientRequestInterceptorManager

For more details about client Interceptors, see "Using Portable Interceptors."

### BindInterceptor

A BindInterceptor object is a global Interceptor which is called on the client side before and after binds.

```
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.

```
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 the following kinds of server Interceptors:

– `POALifeCycleInterceptor` and `POALifeCycleInterceptorManager`

– `ActiveObjectLifeCycleInterceptor` and `ActiveObjectLifeCycleInterceptorManager`

– `ServerRequestInterceptor` and `ServerRequestInterceptorManager`

– `IORCreationInterceptor` and `IORCreationInterceptorManager`

For more details about server Interceptors see "Using Portable Interceptors."

### POALifeCycleInterceptor

A `POALifeCycleInterceptor` object is a global Interceptor which is called every time a POA is created (using the `create` method) or destroyed (using the `destroy` method).

```
class _VISEXPORT POALifeCycleInterceptor : public virtual VISPseudoInterface {
   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 (using the `create` method) or after an object has been deactivated and etherealized (using 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.

```
class _VISEXPORT ActiveObjectLifeCycleInterceptor : public virtual
VISPseudoInterface {
   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 before the invocation (using the preinvoke method) and after the invocation both before and after the marshalling of the reply (using the `postinvoke_premarshal` and `postinvoke_premarshal` methods respectively). This Interceptor may be registered by a `POALifeCycleInterceptor` object at POA creation time on a per-POA basis.

```
class _VISEXPORT ServerRequestInterceptor : public virtual VISPseudoInterface {
   public:
      virtual void preinvoke(CORBA::Object_ptr _target, const char* _operation,
            const IOP::ServiceContextList& _service_contexts,
            CORBA_MarshalInBuffer& _payload,
            VISClosure& _closure) = 0;
      virtual void postinvoke_premarshal(CORBA::Object_ptr _target,
            IOP::ServiceContextList& _service_contexts,
            CORBA::Environment_ptr _env,
            VISClosure& _closure) = 0;
      virtual void postinvoke_postmarshal(CORBA::Object_ptr _target,
            CORBA_MarshalOutBuffer& _payload,
            VISClosure& _closure) = 0;
      virtual void exception_occurred(CORBA::Object_ptr _target,
            CORBA::Environment_ptr _env,
            VISClosure& _closure) = 0;
};
```

**Note**

If a `CORBA::SystemException` or any sub-classes (for example `CORBA::NO_PERMISSION`) is raised on the server side, the exception should not be encrypted. This is because the ORB uses some of these exceptions internally (for example `TRANSIENT` for doing automatic rebind).

### IORCreationInterceptor

An `IORCreationInterceptor` object is called whenever a POA creates an object reference (using the `create` method). This Interceptor may be registered by a `POALifeCycleInterceptor` at POA creation time on a per-POA basis.

```
class _VISEXPORT IORCreationInterceptor : public virtual VISPseudoInterface {
   public:
      virtual void create(PortableServer::POA_ptr _poa,
            IOP::IORValue*& _ior) = 0;
};
```

## Service Resolver Interceptor

This Interceptor is used to install a user service that you can then dynamically load.

```
class _VISEXPORT interceptor::ServiceResolverInterceptor :public virtual
VISPsuedoInterface {
   public:
      virtual CORBA::Object_ptr resolve(const char* _name) = 0;
};
class _VISEXPORT ServiceResolverInterceptorManager :public virtual
InterceptorManager,
   public virtual VISPseudoInterface {
      public:
         virtual void add(const char* _name, ServiceResolverInterceptor_ptr
_interceptor) =
               0;
         virtual void remove(const char* _name) = 0;
};
```

When you call `resolve_initial_references`, the resolve on all installed services gets called. The resolve then can return the appropriate object.

To write service initializers, you must obtain a `ServiceResolver` after getting an `InterceptorManagerControl` to be able to add your services.

## Registering Interceptors with the VisiBroker ORB

Each Interceptor interface has a corresponding Interceptor manager interface which is used to register your Interceptor objects with the VisiBroker ORB. The following steps are 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 the following table which shows the String values to pass to the `get_manager` method of the `InterceptorManagerControl` object. (Be sure to cast the object reference to its corresponding Interceptor manager interface.)

**Table 24.1** String values of the InterceptorManagerControl object

| Value | Corresponding Interceptor interface |
|---|---|
| `ClientRequest` | `ClientRequestInterceptor` |
| `Bind` | `BindInterceptor` |
| `POALifeCycle` | `POALifeCycleInterceptor` |
| `ActiveObjectLifeCycle` | `ActiveObjectLifeCycleInterceptor` |
| `ServerRequest` | `ServerRequestInterceptor` |
| `IORCreation` | `IORCreationInterceptor` |
| `ServiceResolver` | `ServiceResolverInterceptor` |

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 Interceptors and registers them with the VisiBroker ORB. Your factory class must derive from the `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 can also create new instances of your Interceptors and register them with the VisiBroker ORB from within other Interceptors as in the examples in "Example Interceptors".

## 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 in this section uses all of the Interceptor API methods (listed in "Using Portable Interceptors") 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 is simply implemented to print informational messages to the standard output.

The following example applications are located in the directory:

```
<install_dir>\examples\Vbroker\interceptors\
```

- active_object_lifecycle
- client_server
- ior_creation

#### Client-server Interceptors example

To run the example, compile the files as you normally would. Then start up the server and the client as follows:

```
prompt>Server
prompt>Client John
```

**Note**

The `ServiceInit` class used in VisiBroker 3.x is replaced by implementing two interfaces: `ServiceLoader` and `ServiceResolverInterceptor`.

The results of executing the example Interceptor are shown in the following table. The execution by the client and server is listed in sequence.

| Client | Server |
|---|---|
| | `============>SampleServerLoader: Interceptors loaded============> In POA /. Nothing to do.============> In POA bank_agent_poa, 1 ServerRequest interceptor installedStub [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` | |

| Client | Server |
|---|---|
| | `============> SampleServerInterceptor id MyServerInterceptor preinvoke => openCreated 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` | |
| | `============> SampleServerInterceptor id MyServerInterceptor postinvoke_premarshal============> SampleServerInterceptor id MyServerInterceptor postinvoke_postmarshal` |
| `============> SampleClientInterceptor id MyClientInterceptor postinvoke The balance in john's account is $245.64` | |

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 demonstrated by the example code and results, the Interceptors for both the client and server are installed when the respective process starts. Information about registering an interceptor is covered in "Registering Interceptors with the VisiBroker ORB".

## Code listings

### SampleServerLoader

The `SampleServerLoader` object is responsible for loading the `POALifeCycleInterceptor` class and instantiating an object. This class is linked to the VisiBroker ORB dynamically by `vbroker.orb.dynamicLibs`. The `SampleServerLoader` class contains the `init` method which is called by the VisiBroker ORB during initialization. Its sole purpose is to install a `POALifeCycleInterceptor` object by creating it and registering it with the `InterceptorManager`.

```
#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(manager);
         // Add POA interceptor to the list
         poa_mgr->add(
(PortableServerExt::POALifeCycleInterceptor*)interceptor);
         cout << "POA interceptors installed" << endl;
         _poa_interceptors_installed = 1;
      }
};
```

## SamplePOALifeCycleInterceptor

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`.

```
#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;
      }
};
```

## SampleServerInterceptor

The `SampleServerInterceptor` object is invoked every time a request is received at or a reply is made by the server.

```
#include <iostream.h>
#include "vclosure.h"
#include "interceptor_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 : 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;
      }
};
```

## SampleClientInterceptor

The `SampleClientInterceptor` is invoked every time a request is made by or a reply is received at the client.

```
#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_MarshalInBuffer& 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;
      }
};
```

### SampleClientLoader

The `SampleClientLoader` is responsible for loading `BindInterceptor` objects. This class is linked to the VisiBroker ORB dynamically by `vbroker.orb.dynamicLibs`. The `SampleClientLoader` class contains the `bind` and `bind_succeeded` 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 the `InterceptorManager`.

```
#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;
      }
};
```

## SampleBindInterceptor

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.

```
#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
         interceptor::ClientRequestInterceptor_var 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

`Closure` objects are created by the ORB at the beginning of certain sequences of Interceptor calls. The same `Closure` object is used for all calls in that particular sequence. The `Closure` object contains a single public data field `object` of type `java.lang.Object` 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 `Closure` is created before calling `preinvoke_premarshal` and the same `Closure` is used for that request until the request completes, successfully or not. Likewise, in the `ServerInterceptor`, a new `Closure` is created before calling `preinvoke`, and that `Closure` is used for all Interceptor calls related to processing that particular request.

For an example of how `Closure` is used, see the examples in the following directory:

```
<install_dir>/examples/Vbroker/interceptors/client_server
```

The `Closure` object can be cast to `ExtendedClosure` to obtain `response_expected` and `request_id` as follows:

```
CORBA::Boolean my_response_expected =
    ((ExtendedClosure)closure).reqInfo.response_expected;
CORBA::ULong my_request_id =
    ((ExtendedClosure)closure).reqInfo.request_id;
```

## Using both Portable Interceptors and VisiBroker Interceptors simultaneously

Both Portable Interceptors and VisiBroker Interceptors can be installed simultaneously with the VisiBroker ORB. However, as they have different implementations, there are several rules of flow and constrains that developers need to understand when using both Interceptors, as described in the following.

## 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 Interceptors 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 Interceptors 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. An 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, such as `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()`

# 25

# Using object wrappers

This section describes the object wrapper feature of VisiBroker, which allows your applications to be notified or to trap an operation request for an object.

## Object wrappers overview

The VisiBroker 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 which is invoked at the VisiBroker 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. For more information about VisiBroker Interceptors, see "Using VisiBroker Interceptors."

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.

The following 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 VisiBroker `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 offers two kinds of object wrappers: *typed* and *untyped*. You can mix the use of both of these object wrappers within a single application. For information on typed wrappers, see "Typed object wrappers". For information on untyped wrappers, see "Untyped object wrappers". The following table summarizes the important distinctions between these two kinds of object wrappers.

| Features | Typed | Untyped |
|---|---|---|
| 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 untyped 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 an Object wrapper base class for each of your interfaces.

### Object wrapper example applications

The sample client and server applications used to illustrate both the typed and untyped object wrapper concepts in this section are located in the following directory:

```
<install_dir>\examples\Vbroker\interceptors\objectWrappers\
```

## Untyped object wrappers

Untyped 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. Untyped 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 untyped object wrappers within the same client or server application.

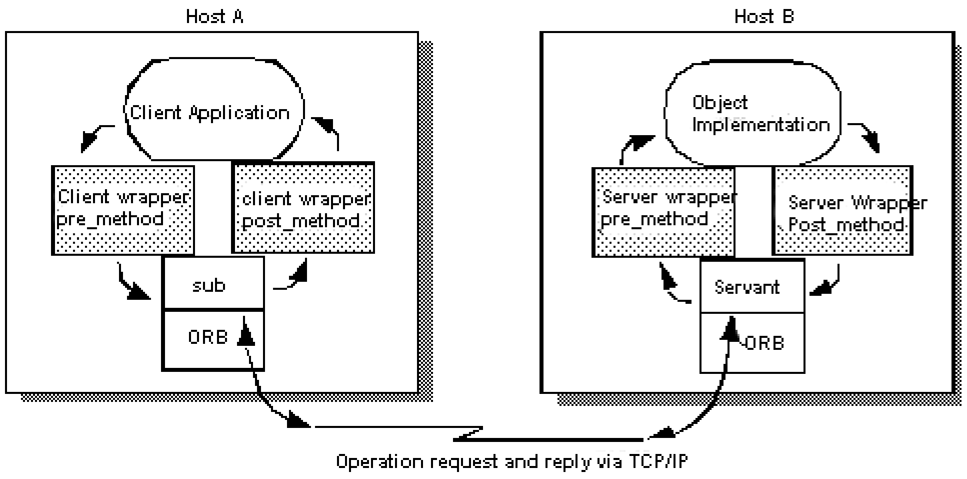
By default, untyped object wrappers have a global scope and will be invoked for any operation request. You can design untyped wrappers so that they have no effect for operation requests on object types in which you are not interested.

**Note**

Unlike typed object wrappers, untyped 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.

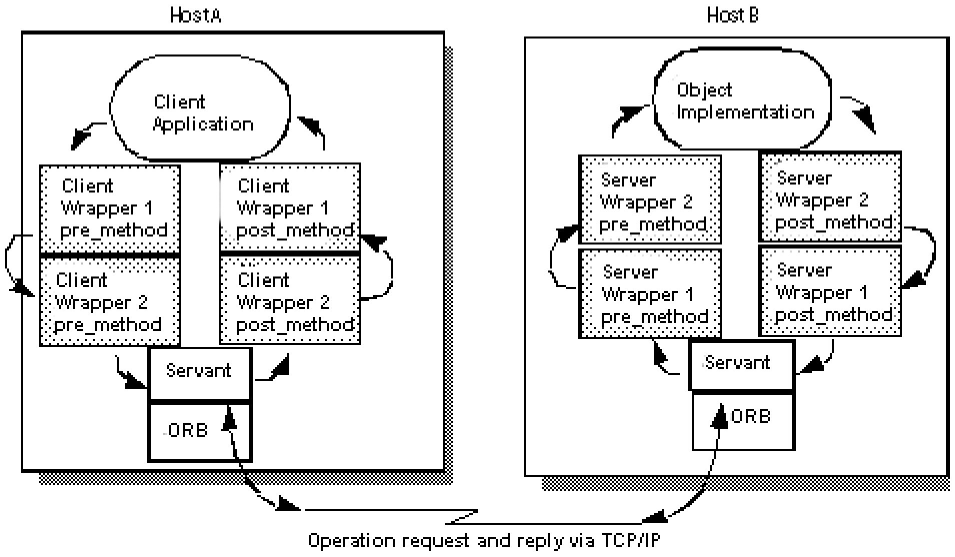
The following figure shows how an untyped object wrapper's `pre_method` is invoked before the client stub method and how the `post_method` is invoked afterward. It also shows the calling sequence on the server-side with respect to the object implementation.

**Figure 25.1** Single untyped object wrapper


Host A
Host B
Client Application
Client wrapper pre_method
client wrapper post_method
sub
ORB
Object Implementation
Server wrapper pre_method
Server Wrapper Post_method
Servant
ORB
Operation request and reply via TCP/IP

## Using multiple, untyped object wrappers

**Figure 25.2** Multiple untyped object wrappers


HostA
Host B
Client Application
Client Wrapper 1 pre_method
Client Wrapper 1 post_method
Client Wrapper 2 pre_method
Client Wrapper 2 post_method
Servant
ORB
Object Implementation
Server Wrapper 2 pre_method
Server Wrapper 2 post_method
Server Wrapper 1 pre_method
Server Wrapper 1 post_method
Servant
ORB
Operation request and reply via TCP/IP

## Order of pre_method invocation

When a client invokes a method on a bound object, each untyped object wrapper `pre_method` will receive control before the client's stub routine is invoked. When a server receives an operation request, each untyped 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 untyped object wrappers, see "Combined use of untyped and typed object wrappers" for information on the invocation order.

## Using untyped object wrappers

The following are the required steps for using untyped object wrappers. Each step is discussed in further detail in the following sections.

1. Identify the interface, or interfaces, for which you want to create a untyped object wrapper.
2. Generate the code from your IDL specification using the `idl2cpp` compiler with the `-obj_wrapper` option.
3. Create an implementation for your untyped object wrapper factory, derived from the `VISObjectWrapper::UntypedObjectWrapperFactory` class.
4. Create an implementation for your untyped object wrapper, derived from the `VISObjectWrapper::UntypedObjectWrapper` class.
5. Modify your application to create your untyped object wrapper factory.

### Implementing an untyped object wrapper factory

The `TimeWrap.h` file, part of the `ObjectWrappers` sample applications, illustrates how to define an untyped object wrapper factory that is derived from the `VISObjectWrapper::UntypedObjectWrapperFactory`.

Your factory's `create` method will be invoked to create an untyped 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 untyped 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 untyped 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.

```
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 untyped object wrapper

The following code sample shows the implementation of the TimingObjectWrapper, also defined in the TimeWrap.h file. Your untyped 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 untyped 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 untyped 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.

```
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 the parameters shown in the following table.

| Parameter | Description |
|---|---|
| operation | Name of the operation that was requested on the target object. |
| target | Target object. |
| closure | Area where data can be saved across method invocations for this wrapper. |
| environment | post_method only parameter used to inform the user of any exceptions that might have occurred during the previous steps of the method invocation. |

## Creating and registering untyped object wrapper factories

An untyped object wrapper factory is automatically added to the chain of untyped 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 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 untyped object wrapper factories for a client. The factories are created after the VisiBroker ORB has been initialized, but before the client binds to any objects.

```
int main(int argc, char* const* argv) {
   try {
         // Initialize the ORB.
         CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
         // Install untyped object wrappers
         TimingObjectWrapperFactory timingfact(VISObjectWrapper::Client,
                    "timeclient");
         TraceObjectWrapperFactory tracingfact(VISObjectWrapper::Client,
                  "traceclient");
         // Now locate an account manager.
   ...]
```

The following code sample illustrates the sample file `UntypedServer.C`, which shows the creation and registration of untyped object wrapper factories for a server. The factories are created after the VisiBroker ORB is initialized, but before any object implementations are created.

```
// UntypedServer.C
#include "Bank_s.hh"
#include "BankImpl.h"
#include "TimeWrap.h"
#include "TraceWrap.h"
USE_STD_NS
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB.
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
      // Initialize the 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_ow_poa",
            poa_manager,
            policies);
      // Install Untyped Object Wrappers for Account Manager.
 TimingObjectWrapperFactory timingfact(VISObjectWrapper::Server,
                  "timingserver");
      TraceObjectWrapperFactory tracingfact(VISObjectWrapper::Server,
                  "traceserver");
      // Create the Account Manager Servant.
      AccountManagerImpl managerServant;
      // Decide on ID for 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.
      rootPoa->the_POAManager()->activate();
      cout << "Manager is ready." << endl;
      // Wait for Incoming Requests.
      orb->run();
   } catch(const CORBA::Exception& e) {
      cerr << e << endl;
      return 1;
   }
   return 0;
}
```

### Removing untyped object wrappers

The `VISObjectWrapper::ChainUntypedObjectWrapperFactory` class `remove` method can be used to remove an untyped 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

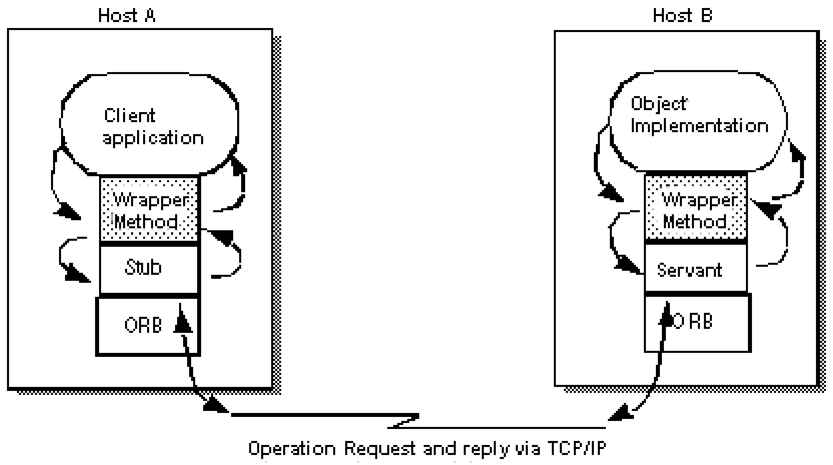
When you implement a typed object wrapper for a particular class, you define the processing that is to take place when a method is invoked on a bound object. The following figure shows how an object wrapper method on the client is invoked before the client stub class method and how an object wrapper 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 untyped object wrappers within the same client or server application. For more information, see “Combined use of untyped and typed object wrappers”.

**Figure 25.3** Single typed object wrapper registered



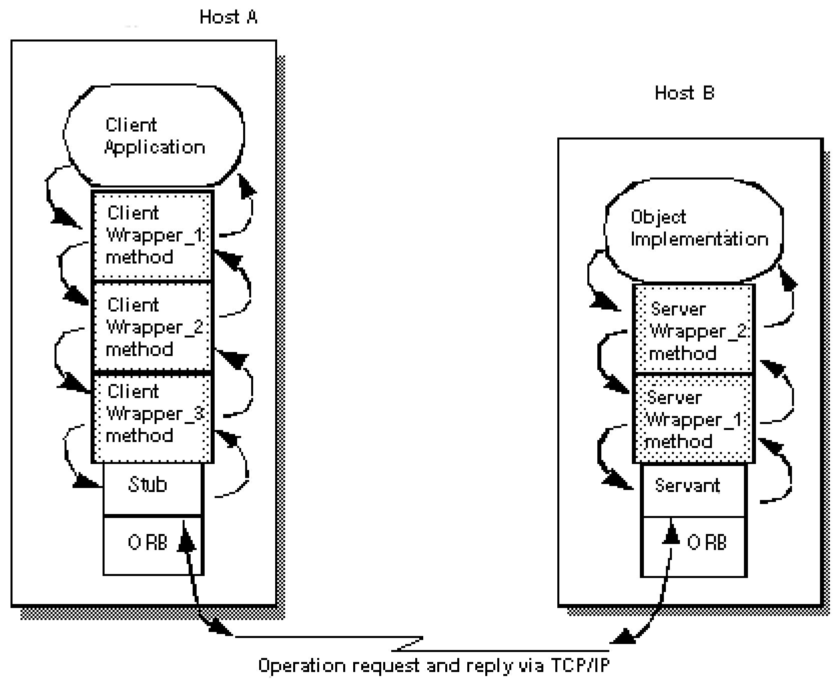
## Using multiple, typed object wrappers

You can implement and register more than one typed object wrapper for a particular class of object, as shown in the following figure.

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 25.4** Multiple, typed object wrappers registered



## Order of invocation

The methods for a typed object wrapper that are registered for a particular class will receive all of the arguments that are normally passed to the stub method on the client side or to the 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.

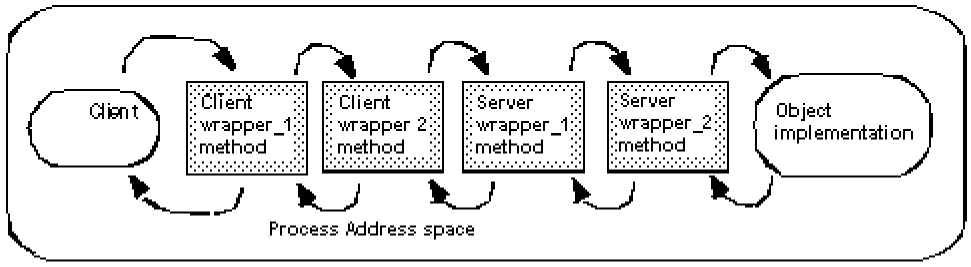
A typed object wrapper method's ability to return control to the previous method in the chain allows you to create a wrapper method that never invokes a client stub or object implementation. For example, you can create an object wrapper method that caches the results of a frequently requested operation. 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 untyped object wrappers, see "Combined use of untyped 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 process, the first object wrapper method to receive control will belong to the first client-side object wrapper that was installed. The following figure illustrates the invocation order.

**Figure 25.5** Typed object wrapper invocation order



## Using typed object wrappers

The following are the required steps for using typed object wrappers. Each step is discussed in further detail in the following sections.

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 with the `-obj_wrapper` option.

3 Derive your typed object wrapper class from the `<interface_name>ObjectWrapper` class generated by the 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 sample 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`, the `_inited` flag is set to `true`, and the value is returned.

3 If this method has been invoked before, simply return the cached value.

```
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 sample shows a portion of the `TypedClient.java` file that creates and registers a typed object wrapper.

```
...
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB.
 CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

      // 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 VisiBroker 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 sample shows a portion of the `TypedServer.C` file that installs a typed object wrapper.

```
// TypedServer.C
#include "Bank_s.hh"
#include "BankImpl.h"
#include "BankWrap.h"
USE_STD_NS
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB.
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
      // Initialize the 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_ow_poa",
            poa_manager,
            policies);
```

```
         // 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 Account Manager Servant.
         AccountManagerImpl managerServant;
         // Decide on ID for 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.
         rootPoa->the_POAManager()->activate();
         cout << "Manager is ready." << endl;
         // Wait for Incoming Requests.
         Orb>run();
      } catch(const CORBA::Exception& e) {
         cerr << e << endl;
         return 1;
      }
      return 0;
   }
```

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 untyped and typed object wrappers

If you choose to use both typed and untyped object wrappers in your application, all `pre_method` methods defined for the untyped 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 untyped 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 untyped object wrappers are to be used.

## Command-line arguments for typed wrappers

The following table shows the command-line arguments you can use to enable the use of typed object wrappers for the sample bank applications implemented in `Client.C` and `Server.C`.

| **Bank wrappers properties** | **Description** |
|---|---|
| `-BANKaccountCacheClnt <0\|1>` | Enables or disables a typed object wrapper that caches the results of the `balance` method for a client application. |
| `-BANKaccountCacheSrvr <0\|1>` | Enables or disables a typed object wrapper that caches the results of the `balance` method for a server application. |
| `-BANKmanagerCacheClnt <0\|1>` | Enables or disables a typed object wrapper that caches the results of the `open` method for a client application. |
| `-BANKmanagerCacheSrvr <0\|1>` | Enables or disables a typed object wrapper that caches the results of the `open` method for a server application. |
| `-BANKmanagerSecurityClnt <0\|1>` | Enables or disables a typed object wrapper that detects unauthorized users passed on the `open` method for a client application. |
| `-BANKmanagerSecuritySrvr <0\|1>` | Enables or disables a typed object wrapper that detects unauthorized users passed on the `open` method for a server application. |

## Initializer for typed wrappers

The typed wrappers are created in the `BankInit::update` initializer, defined in `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 following code sample 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.

```
...
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 untyped wrappers

The following table shows the command-line arguments you can use to enable the use of untyped object wrappers for the sample bank applications implemented in `Client.C` and `Server.C`.

| Bank wrappers properties | Description |
|---|---|
| `-TRACEWRAPclient <numwraps>` | Instantiates the specified number of untyped object wrapper factories for tracing wrappers for a client application. |
| `-TRACEWRAPserver <numwraps>` | Instantiate the specified number of untyped object wrapper factories for tracing on a server application. |
| `-TRACEWRAPboth <numwraps>` | Instantiate the specified number of untyped object wrapper factories for tracing for both a client and server application. |
| `-TIMINGWRAPclient <numwraps>` | Instantiate the specified number of untyped object wrapper factories for timing on a client application. |
| `-TIMINGWRAPserver <numwraps>` | Instantiate the specified number of untyped object wrapper factories for timing on a server application |
| `-TIMINGWRAPboth <numwraps>` | Instantiate the specified number of untyped object wrapper factories for timing on both a client and a server application. |

## Initializers for untyped wrappers

The untyped wrappers are created and registered in the `TraceWrapInit::update` and`TimingWrapInit::update` methods, defined in `BankWrappers/TraceWrap.C` and `TimeWrap.C`. These initializers will be invoked when the `ORB_init` method is invoked and will handle the installation of various untyped object wrappers.

The following code sample shows a portion of the `TraceWrap.C` file, which will install the appropriate untyped object wrapper factories, based on the command-line properties you specify.

```
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. For more information, see "Starting a Smart Agent (osagent)". You can then execute the server application without any tracing or timing object wrappers by using the following command:

```
prompt> Server
```

**Note**

The server is designed as a co-located application. It implements both the server and a client.

From another window, you can execute the client application without any tracing or timing object wrappers to query the balance in a user's account using the following command:

```
prompt> Client John
```

You can also execute the following command if you want a default name to be used:

```
prompt> Client
```

### Turning on timing and tracing object wrappers

To execute the client with untyped timing and tracing object wrappers enabled, use the following command:

```
prompt> Client -TRACEWRAPclient 1 -TIMINGWRAPclient 1
```

To execute the server with untyped wrappers for timing and tracing enabled, use the following command:

```
prompt> 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:

```
prompt> Client -BANKaccountCacheClnt 1 _BANKmanagerCacheClnt 1 \
         -BANKmanagerSecurityClnt 1
```

To execute the server with typed wrappers for caching and security enabled, use the followiong command:

```
prompt> Server -BANKaccountCacheSrvr 1 -BANKmanagerCacheSrvr 1 \
         -BANKmanagerSecuritySrvr 1
```

## Turning on typesd and untyped wrappers

To execute the client with all typed and untyped wrappers enabled, use the following command:

```
prompt> Client -BANKaccountCacheClnt 1 -BANKmanagerCacheClnt 1 \
      -BANKmanagerSecurityClnt 1 \
      -TRACEWRAPclient 1 -TIMINGWRAPclient 1
```

To execute the server with all typed and untyped wrappers enabled, use the following command:

```
prompt> 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 untyped wrapper enabled for just the client, and the untyped tracing wrapper for just the server:

```
prompt> Server -BANKaccountCacheClnt 1 -BANKaccountCacheSrvr 1 \
         -BANKmanagerCacheClnt 1 -BANKmanagerCacheSrvr 1 \
         -BANKmanagerSecurityClnt 1 \
         -BANKmanagerSecuritySrvr 1 \
         -TRACEWRAPboth 1 \
         -TIMINGWRAPboth 1
```

# 26

# Event Queue

This section provides information about the Event Queue feature. This feature is provided for the server-side only.

A server can register listeners to the event queue based on event types that the server is interested and therefore can process those events when the server needs to do so.

## Event types

Currently, connection event type is the only event type generated.

### Connection events

There are two connection events that the VisiBroker ORB will generate and push to the registered connection event, as follows:

- *Connection established:* Indicates that a new client is connected to the server successfully.

- *Connection closed:* Indicates that an existing client is disconnected from the server.

### Event listeners

A server implements and registers listeners with the VisiBroker ORB based on event types the server needs to process. The connection event listener is the only event lis tener supported.

## IDL definition

The interface definitions are as follows:

```
module EventQueue {
// Connection event types
enum EventType {UNDEFINED, CONN_EVENT_TYPE};
// Peer (Client) connection info
struct ConnInfo {
string ipaddress; // in %d.%d.%d.%d format
long port;
long connID;
};
```

// Marker interface for all types of event listeners

```
local interface EventListener {};
typedef sequence<EventListener> EventListeners;
// connection event listener interface
local interface ConnEventListener : EventListener{
void conn_established(in ConnInfo info);
void conn_closed(in ConnInfo info);
};
```

// The EventQueue manager

```
local interface EventQueueManager : interceptor::InterceptorManager {
void register_listener(in EventListener listener, in EventType type);
void unregister_listener(in EventListener listener, in EventType type);
EventListeners get_listeners(in EventType type);
};
};
```

The details of the interface definitions are described in the following sections.

## ConnInfo structure

The ConnInfo structure contains the following client connection information.

**Table 26.1** ConnInfo structure client connection information

| Paramters | Description |
|---|---|
| `ipaddress` | stores the client ip address |
| `port` | stores the client port number |
| `connID` | stores the per server unique identification for this client connection |

### EventListener interface

The `EventListener` interface section is the marker interface for all types of event listeners.

### ConnEventListeners interface

The `ConnEventListeners` interface defines the following operations.

**Table 26.2** ConnEventListeners interface operations

| Operation | Description |
|---|---|
| `void conn_established` (in ConnInfo info) | This operation is called back by the VisiBroker ORB to push the connection established event. The VisiBroker ORB fills in the client connection information into the in `ConnInfo` info parameter and passes this value into the callback operation. |
| `void conn_closed` (in ConnInfo info) | This operation is called back by the VisiBroker ORB to push the connection closed event. The VisiBroker ORB fills in the client connection information into the in `ConnInfo` info parameter and passes this value into the callback operation. |

The server-side application is responsible for the implementation of the ConnEventListener interface as well as the processing of the events being pushed into the listener.

## EventQueueManager interface

The `EventQueueManager` interface is used as a handle by the server-side implementation for the registration of event listeners. This interface defines the the following operations.first.

| Operation | Description |
|---|---|
| `void register_listener` (in EventListener listener, in EventType type) | This operation is provided for the registration of an event listener with the specified event type. |
| `EventListeners get_listeners` (in EventType type) | This operation returns the list of registered event listeners for the specified type. |
| `void unregister_listener` (in EventListener listener, in EventType type) | This operation removes a pre-registered listener of the specified type. |

## How to return the EventQueueManager

An `EventQueueManager` object is created upon ORB initialization. Server-side implementation returns the `EventQueueManager` object reference using the following code:

```
CORBA::Object *object =
orb->resolve_initial_references("VisiBrokerInterceptorControl");
interceptor::InterceptorManagerControl_var control =
interceptor::InterceptorManagerControl::_narrow(object);
interceptor::InterceptorManager_var manager =
control->get_manager("EventQueueManager");
EventQueue::EventQueueManager_var eq_mgr =
EventQueue::EventQueueManager::_narrow(manager);Note
```

## Event Queue code samples

This section contains some code samples for registering EventListeners and implementing a connection EventListener.

### Registering EventListeners

The `SampleServerLoader` class contains the `init()` method which is called by the ORB during initialization. The purpose of the ServerLoader is to register an `EventListener` by creating and registering it to the `EventQueueManager`.

```
#ifdef _VIS_STD
#include <iostream>
#else
#include <iostream.h>
#endif
#include "vinit.h"
#include "ConnEventListenerImpl.h"

USE_STD_NS
```

```
class SampleServerLoader : VISInit {
private:
   short int _conn_event_interceptors_installed;
public:
   SampleServerLoader(){
      _conn_event_interceptors_installed = 0;
}
void ORB_init(int& argc, char* const* argv, CORBA::ORB_ptr orb) {
   if( _conn_event_interceptors_installed) return;
cout << "Installing Connection event interceptors" << endl;
ConnEventListenerImpl *interceptor =
   new ConnEventListenerImpl("ConnEventListener");
// 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("EventQueueManager");
EventQueue::EventQueueManager_var eq_mgr =
   EventQueue::EventQueueManager::_narrow(manager);
// Add POA interceptor to the list
eq_mgr->register_listener(
   (EventQueue::ConnEventListener *)interceptor,
EventQueue::CONN_EVENT_TYPE);
   cout << "Event queue interceptors installed" << endl;
   _conn_event_interceptors_installed = 1;
   }
};
```

## Implementing EventListeners

The ConnEventListenerImpl contains a connection event listener implementation sample. The ConnEventListener interface implements the conn_established and conn_closed operations at the server-side application. For more information, see "ConnEventListeners interface". The implementation enables the connection to idle for 30000 milliseconds while waiting for a request at the server-side.

These operations are called when the connection is established by the client and when the connection is dropped, respectively.

```
#ifdef _VIS_STD
#include <iostream>
#else
#include <iostream.h>
#endif
#include "vextclosure.h"
#include "interceptor_c.hh"
#include "IOP_c.hh"
#include "EventQueue_c.hh"
#include "vutil.h"

// USE_STD_NS is a define setup by VisiBroker to use the std namespace
USE_STD_NS

//-------------------------------------------------------------------
// defines the server interceptor functionality
//-------------------------------------------------------------------
```

```
class ConnEventListenerImpl : EventQueue::ConnEventListener
{
private:
char * _id;
public:
ConnEventListenerImpl( const char* id) {
_id = new char[ strlen(id) + 1];
strcpy( _id,id);
}
~ConnEventListenerImpl() {
delete[] _id;
_id = NULL;
}

//----------------------------------------------------------------------
// This method gets called when a request arrives at the server end.
//----------------------------------------------------------------------

void conn_established(const EventQueue::ConnInfo& connInfo){
      cout <<"Processing connection established from" <<endl;
      cout << connInfo;
      cout <<endl;
      VISUtil::sleep(30000);
   }
void conn_closed(const EventQueue::ConnInfo & connInfo) {
      cout <<"Processing connection closed from " <<endl ;
      cout <<connInfo ;
      cout << endl;
      VISUtil::sleep(30000);
   }
};
```

27

# Using the dynamically managed types

This section describes the `DynAny` feature of VisiBroker, which allows you to construct and interpret data types at runtime.

## DynAny interface 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. Using the `DynAny` interface, you can build powerful client and server applications that create and interpret data types at runtime.

### DynAny examples

Example client and server applications that illustrate the use of `DynAny` are included as part of the VisiBroker distribution. The examples are located in the following directory:

```
<install_dir>\examples\Vbroker\dynany\
```

These example programs are used to illustrate `DynAny` concepts throughout this section.

## 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, its methods and classes are also documented in the VisiBroker API References. “Programmer tools for C++,” 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.

| Interface | TypeCode | Description |
|---|---|---|
| DynArray | _tk_array | An array of values with the same data type that has a fixed number of elements. |
| DynEnum | _tk_enum | A single enumeration value. |
| DynFixed | _tk_fixed | Not supported. |
| DynSequence | _tk_sequence | A sequence of values with the same data type. The number of elements may be increased or decreased. |
| DynStruct | _tk_struct | A structure. |
| DynUnion | _tk_union | A union. |
| DynValue | _tk_value | Not supported. |

## DynAny usage restrictions

A DynAny object may only be used locally by the process 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 3.0.

## 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 in C++ or the DynAny.get_<type> methods in Java 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 provides methods for copying, assigning, and converting to or from an Any object. The sample programs, described in “DynAny example client application” and “DynAny example server application”, provide examples of how to use some of these methods.

# Constructed data types

The following types are derived from the `DynAny` interface and are used to represent constructed data types.

## 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 |
|---|---|
| `rewind` | Resets the current component pointer to the first component. Has no effect if the object contains only one component. |
| `next` | Advances the pointer to the next component. If there are no more components or if the object contains only one component, `false` is returned. |
| `current_component` | Returns a `DynAny` object, which may be narrowed to the appropriate type, based on the component's `TypeCode`. |
| `seek` | Sets the current component pointer to the component with the specified, zero-based index. Returns `false` if there is no component at the specified index. Sets the current component pointer to `-1` (no component) if specified with a negative index. |

## DynEnum

The `DynEnum` interface represents a single enumeration constant. Methods are provided for setting and obtaining the value as a string or as an integral value.

## DynStruct

The `DynStruct` 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

The `DynUnion` 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 can use the `rewind`, `next`, `current_component`, and `seek` methods to traverse the members in a `DynArray` or `DynSequence`.

## DynAny 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.

```
// 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();
   };
};
```

## DynAny example client application

The following code sample shows a client application that can be found in the following VisiBroker distribution directory:

```
<install_dir>\examples\Vbroker\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 VisiBroker ORB in the usual manner and binding to a `PrintManager` object, the client performs the following steps:

1. Creates an empty `DynStruct` with the appropriate type.
2. Creates a sequence of `NameValuePair` objects that will contain the structure members.
3. Creates and initializes `Any` objects for each of the structure member's values.
4. Initializes each `NameValuePair` with the appropriate member name and value.
5. Initializes the `DynStruct` object with the `NameValuePair` sequence.
6. Invokes 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.

The following code sample is an example of a client application that uses DynStruct:

```
// Client.C
#include "Printer_c.hh"
#include "dynany.h"
int main(int argc, char* const* argv) {
  try {
    // Initialize the ORB.
    CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
    DynamicAny::DynAnyFactory_var factory =
        DynamicAny::DynAnyFactory::_narrow(
            orb->resolve_initial_references("DynAnyFactory"));
    // Get the manager Id
    PortableServer::ObjectId_var managerId =
        PortableServer::string_to_ObjectId("PrinterManager");
    // Locate an account manager. Give the full POA name and the servant ID.
    Printer::PrinterManager_ptr manager =
        Printer::PrinterManager::_bind("/serverPoa", managerId);
    DynamicAny::NameValuePairSeq seq(3);
    seq.length(3);
    CORBA::Any strAny,enumAny,floatAny;
    strAny <<= "String";
    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::DynStruct_var info =
        DynamicAny::DynStruct::_narrow(
            factory->create_dyn_any_from_type_code(
                          Printer::_tc_StructType));
    info->set_members(seq);
    manager->printAny(*(info->to_any()));
    manager->shutdown();
  }
  catch(const CORBA::Exception& e) {
      cerr << "Caught " << e << "Exception" << endl;
  }
}
```

## DynAny example server application

The following code sample shows a server application that can be found in the following VisiBroker distribution directory:

```
<install_dir>\examples\Vbroker\dynany\
```

The server application performs the following steps.

1. Initializes the VisiBroker `ORB`.
2. Creates the policies for the POA.
3. Creates a `PrintManager` object.
4. Exports the `PrintManager` object.
5. Prints a message and waits for incoming operation requests.

```
...
int main(int argc, char* const* argv) {
  try {
    // Initialize the ORB.
     CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
```

```
      int Verbose = 0;
      // get a reference to the root POA
       PortableServer::POA_var rootPOA =
           PortableServer::POA::_narrow(
               orb->resolve_initial_references("RootPOA"));
       CORBA::PolicyList policies;
       policies.length(1);
       policies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy(
               PortableServer::PERSISTENT);
       // Create serverPOA with the right policies
       PortableServer::POA_var serverPOA = rootPOA->create_POA( "serverPoa",
               rootPOA->the_POAManager(),
               policies );
       // Resolve Dynamic Any Factory
       DynamicAny::DynAnyFactory_var factory =
               orb->resolve_initial_references("DynAnyFactory");
       PortableServer::ObjectId_var managerId =
               PortableServer::string_to_ObjectId("PrinterManager");
       // Create the printer manager object.
       PrinterManagerImpl manager( orb, factory, serverPOA, managerId);
       // Export the newly create object.
       serverPOA->activate_object_with_id(managerId,&manager);
       // Activate the POA Manager
       rootPOA->the_POAManager()->activate();
       cout << serverPOA->servant_to_reference(&manager)
               << " is ready" << endl;
       // Wait for incoming requests
       orb->run();
    }
    catch(const CORBA::Exception& e) {
      cerr << e << endl;
    }
  }
```

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 Creates a DynAny object, initializing it with the received Any.

2 Performs a switch on the DynAny object's type.

3 If the DynAny contains a basic data type, simply prints out the value.

4 If the DynAny contains an Any type, creates a DynAny for it, determines it's contents, and then prints out the value.

5 If the DynAny contains an enum, creates a DynEnum for it and then prints out the string value.

6 If the DynAny contains a union, creates a DynUnion for it and then prints out the union's discriminator and the member.

7 If the DynAny contains a struct, array, or sequence, traverses through the contained components and prints out each value.

```
// PrinterManager Implementation
class PrinterManagerImpl : public POA_Printer::PrinterManager
{
   CORBA::ORB_var                 _orb;
   DynamicAny::DynAnyFactory_var  _factory;
   PortableServer::POA_var        _poa;
   PortableServer::ObjectId_var   _oid;

public:
```

```
PrinterManagerImpl(CORBA::ORB_ptr orb,
   DynamicAny::DynAnyFactory_ptr DynAnyFactory,
   PortableServer::POA_ptr poa,
   PortableServer::ObjectId_ptr oid
   ) : _orb(orb), _factory(DynAnyFactory),
       _poa(poa), _oid(oid) {}

void printAny(const CORBA::Any& info) {
 try {
     // Create a DynAny object
   DynamicAny::DynAny_var dynAny = _factory->create_dyn_any(info);
     display(dynAny);
  }
  catch (CORBA::Exception& e) {
     cout << "Unable to create Dynamic Any from factory" << endl;
  }
}
 void shutdown() {
 try {
    _poa->deactivate_object(_oid);
     cout << "Server shutting down..." << endl;
    _orb->shutdown(0UL);
  }
catch (const CORBA::Exception& e) {
     cout << e << endl;
  }
}

void display(DynamicAny::DynAny_var value)  {
  switch(value->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;
        }
        case CORBA::tk_octet: {
          cout << value->get_octet() << endl;
          break;
        }
        case CORBA::tk_string: {
          cout << value->get_string() << endl;
          break;
        }
        case CORBA::tk_any: {
          DynamicAny::DynAny_var dynAny = _factory->create_dyn_any(*(
                          value->get_any()));
          display(dynAny);
          break;
        }
        case CORBA::tk_TypeCode: {
          cout << value->get_typecode() << endl;
          break;
        }
        case CORBA::tk_objref: {
          cout << value->get_reference() << endl;
          break;
        }
        case CORBA::tk_enum: {
          DynamicAny::DynEnum_var dynEnum = DynamicAny::DynEnum::_narrow(value);
          cout << dynEnum->get_as_string() << endl;
          break;
        }
        case CORBA::tk_union: {
         DynamicAny::DynUnion_var dynUnion =
DynamicAny::DynUnion::_narrow(value);
          display(dynUnion->get_discriminator());
          display(dynUnion->member());
          break;
        }
        case CORBA::tk_struct:
        case CORBA::tk_array:
        case CORBA::tk_sequence: {
          value->rewind();
        CORBA::Boolean next = 1UL;
          while(next) {
          DynamicAny::DynAny_var d = value->current_component();
            display(d);
            next = value->next();
          }
          break;
        }
        case CORBA::tk_longlong: {
          cout << value->get_longlong() << endl;
          break;
        }
        case CORBA::tk_ulonglong: {
          cout << value->get_ulonglong() << endl;
          break;
        }
```

```
      case CORBA::tk_wstring: {
        cout << value->get_wstring() << endl;
        break;
      }
      case CORBA::tk_wchar: {
        cout << value->get_wchar() << endl;
        break;
      }
      default:
        cout << "Invalid Type" << endl;
     }
   }
};
```

# 28

# Using valuetypes

This section explains how to use the `valuetype` IDL type in VisiBroker.

## Understanding valuetypes

The `valuetype` IDL type 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.

### Valuetype IDL code sample

The following IDL code declares a simple `valuetype`:

```
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 VisiBroker 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 one other 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 `struct`, `union`, or `sequence` cannot 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 VisiBroker ORB.

7 Either implement the `_add_ref`, `_remove_ref`, and `_ref_countvalue` methods or derive from `CORBA::DefaultValueRefCountBase`.

## Defining your valuetypes

In the IDL sample (for more information, see “Valuetype IDL code sample”), 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

When you have defined your IDL, compile it using `idl2cpp` to create source files. You then modify the source files to implement your valuetypes.

If you compile the IDL shown in “Valuetype IDL code sample”, your output consists 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.

In the `obv\PointImpl.java`, the `PointImpl` class extends the `Point` class, which is generated from the IDL.

Inheriting the valuetype base class:

```
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

When you have created an implementation class, implement the Factory for your valuetype.

In the following example, the generated `Point_init` class contains the `create` method declared in your IDL. This class extends `CORBA::ValueFactoryBase` . The `PointDefaultFactory` class implements `PointValueFactory` as shown in the following example.

```
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 does not create a skeleton class for this.

### Registering your Factory with the VisiBroker ORB

To register your Factory with the VisiBroker ORB, call `ORB::register_value_factory`. For more information on registering Factories, see "Registering valuetypes".

## Implementing factories

When the VisiBroker ORB receives a valuetype, it must first be demarshaled, 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 unmarshaled 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.

VisiBroker version 4.5 or later version will generate the correct signatures for either the JDK 1.3 or JDK 1.4 default value factory method. Existing (version 4.0) generated code is not designed to run under JDK 1.3, unless you modify the default value factory method signature as shown below. If you use your existing code with JDK 1.3 and do not modify default value factory, the code will not compile or will throw a `NO_IMPLEMENT` exception. Consequently, we recommend that you regenerate your code to generate the correct signatures.

The following code sample shows how you should modify the default value factory method signature to make sure that it compiles under JDK 1.3:

```
class PointFactory: public CORBA::ValueFactoryBase
{
public:
   PointFactory(){}
   virtual  ~PointFactory(){}
   CORBA::ValueBase*  create_for_unmarshal() {
      return new PointImpl();
   }
};
```

### Factories and valuetypes

When the VisiBroker ORB receives a valuetype, it will look for that type's factory. It will look for a factory named `<valuetype>DefaultFactory`. For instance, the Point valuetype's factory is called `PointDefaultFactory`. If the correct factory doesn't conform to this naming schema (`<valuetype>DefaultFactory`), you must register the correct factory so the VisiBroker ORB can create an instance of the valuetype.

If the VisiBroker ORB cannot find the correct factory for a given valuetype, a `MARSHAL` exception is raised, with an identified minor code.

### Registering valuetypes

Each language mapping specifies how and when registration occurs. If you created a factory with the `<valuetype>DefaultFactory` naming convention, this is considered implicitly registering that factory, and you do not need to explicitly register your factory with the VisiBroker ORB.

To register a factory that does not conform to the `<valuetype>DefaultFactory` naming convention, call `register_value_factory`. To unregister a factory, call `unregister_value_factory` on the VisiBroker ORB. You can also lookup a registered valuetype factory by calling `lookup_value_factory` on the VisiBroker 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.

Abstract interfaces 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 VisiBroker 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.

```
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.

```
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.

## 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 in 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.

# 29

# Bidirectional Communication

This section explains how to establish bidirectional connections in VisiBroker without using the GateKeeper. For information about bidirectional communications when using GateKeeper, see "Introduction to GateKeeper."

**Note**

Before enabling bidirectional IIOP, please read about "Security considerations".

## Using bidirectional IIOP

Most clients and servers that exchange information by way of 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 IIOP 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 VisiBroker ORB properties

The following 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
```

### 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 following table describes the `vbroker.orb.enableBiDir` property value options:

| Value | Description |
|---|---|
| `client` | Enables bidirectional IIOP for all POAs and for all outgoing connections. This setting is equivalent to creating all POAs with a setting of the BiDirectional policy to `both` and setting the policy override for the BiDirectional policy to `both` on the VisiBroker ORB level. Furthermore, all created SCMs will permit bidirectional connections, as if the `exportBiDir` property had been set to `true` for every SCM. |
| `server` | Causes the server to accept and use connections that are bidirectional. This is equivalent to setting the `importBiDir` property on all SCMs to `true`. |
| `both` | Sets the property to both `client` and `server`. |
| `none` | Disables bidirectional IIOP altogether. This is the default value. |

### 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.

### importBiDir property

The `vbroker.se.<se-name>.scm.<scm-name>.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 take effect. This allows you to set the `enableBiDir` property globally and specifically turn off BiDir in individual SCMs.

## About the BiDirectional examples

Examples demonstrating use of this feature are installed as part of your VisiBroker distribution in subdirectories in the following location:

```
<install_dir>\examples\Vbroker\bidir-iiop
```

All the examples are based on a simple stock quote callback application:

1. The client creates a CORBA object that processes stock quote updates.
2. The client sends the object reference of this CORBA object to the server.
3. 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 for Java and C++ applications without modifying any source code. A simple callback application that does not use bidirectional IIOP at all is located in the following directory:

```
<install_dir>\examples\Vbroker\bidir-iiop\basic
```

To enable bidirectional IIOP for the callback example, you set the `vbroker.orb.enableBiDir` property as follows:

1 Make sure the osagent is running.

2 Start the server.

**UNIX**

```
prompt> -Dvbroker.orb.enableBiDir=server Server &
```

**Windows**

```
prompt> start -Dvbroker.orb.enableBiDir=server Server
```

3 Start the client.

```
prompt> -Dvbroker.orb.enableBiDir=client RegularClient
```

The existing callback application now uses bidirectional IIOP and works through a client-side firewall.

## Explicitly enabling bidirectional IIOP

The client in directory `<install_dir>\examples\Vbroker\bidir-iiop\basic` is derived from the `RegularClient` described in “Enabling bidirectional IIOP for existing applications”, except that this client enables bidirectional IIOP programmatically.

The changes required are in the client code only. To convert the unidirectional client into a bidirectional client, all you need to do is:

1 Include the BiDirectional policy in the list of policies for the callback POA.

2 Add the BiDirectional policy to the list of overrides for the object reference that refers to the server for which we want to enable bidirectional IIOP.

3 Set the `exportBiDir` property to `true` in the client.

In the following code sample, the code that implements bidirectional IIOP is displayed in bold:

```
try {

   CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

   // Get the manager Id
   PortableServer::ObjectId_var managerId =
       PortableServer::string_to_ObjectId("BankManager");
   PortableServer::ObjectId_var oid =
       PortableServer::string_to_ObjectId("QuoteServer");
   Quote::QuoteServer_var quoter =
       Quote::QuoteServer::_bind("/QuoteServer_poa", oid);

   // set up the callback object... first get the RootPOA
    CORBA::Object_var obj =
       orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var rootPOA =
       PortableServer::POA::_narrow(obj);
    PortableServer::POAManager_var the_manager =
       rootPOA->the_POAManager();
    PortableServer::POA_var consumer_poa;

   // Set up a policy.
   CORBA::Any policy_value;
   policy_value <<= BiDirPolicy::BOTH;
   CORBA::Policy_var policy =
       orb->create_policy(
           BiDirPolicy::BIDIRECTIONAL_POLICY_TYPE,
           policy_value);
   CORBA::PolicyList policies;
   policies.length(1);
   policies[0] = CORBA::Policy::_duplicate(policy);
   consumer_poa = rootPOA->create_POA(
       "QuoteConsumer_poa", the_manager, policies );
   QuoteConsumerImpl* consumer = new QuoteConsumerImpl;
   oid = PortableServer::string_to_ObjectId("consumer");
   consumer_poa->activate_object_with_id(oid, consumer);
   the_manager->activate();
   CORBA::Object_var obj =
       quoter->set_policy_overrides(policies, CORBA::ADD_OVERRIDE);
   quoter = Quote::QuoteServer::_narrow(obj);
   obj = consumer_poa->id_to_reference(oid);
   Quote::QuoteConsumer_var quote_consumer =
       Quote::QuoteConsumer::_narrow(obj);
   quoter->registerConsumer(quote_consumer.in());
   cout << "implementation is running" << endl;
   orb->run();
}
catch(const CORBA::Exception& e) {
   cout << e << endl;
}
```

## Unidirectional or bidirectional connections

A client connection can be either unidirectional or bidirectional. A server can use a bidirectional connection to call back the client without opening a new connection. Otherwise, the connection is considered unidirectional.

### Enabling bidirectional IIOP for POAs

The POA on which the callback object is hosted must enable bidirectional IIOP by setting the BiDirectional policy to `BOTH`. This POA must be created on an SCM which has been enabled for bidirectional support by setting the `vbroker.<sename>.scm.<scmname>.manager.exportBiDir` property on the SCM manager. Otherwise, the POA will not be able to receive requests from the server over a client-initiated connection.

If a POA does not specify the BiDirectional policy, it must not be exposed in outgoing connections. To satisfy this requirement, a POA which does not have the BiDirectional policy set cannot be created on a server engine which has even one SCM whose `exportBiDir` property is set. If an attempt is made to create a POA on a unidirectional SE, an `InvalidPolicy` exception is raised, with the `ServerEnginePolicy` in error.

**Note**

Different objects using the same client connection may set conflicting overrides for the BiDirectional policy. Nevertheless, once a connection is made bidirectional, it always remains bidirectional, regardless of the policy effective at a later time.

Once you have full control over the bidirectional configuration, you enable bidirectional IIOP on the `iiop_tp SCM` only:

```
prompt> -Dvbroker.se.iiop_tp.scm.iiop_tp.manager.exportBiDir=
true Client
```

## 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 VisiBroker ORB properties" for more information.) 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).

# 30

# Using the BOA with VisiBroker

This section describes how to use the BOA with VisiBroker.

**Note**

BOA support is provided as backward compatibility for VisiBroker version 4.0 (CORBA spec. 2.1) and 3.x versions. For current CORBA specification support, see “Using POAs.”

## Compiling your BOA code with VisiBroker

If you have existing BOA code that you developed with a previous version of VisiBroker, you can continue to use it with the current version.

**Note**

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 “IDL to C++ mapping.”

## Supporting BOA options

All BOA command line options supported by VisiBroker 4.x are still supported.

## Using object activators

BOA object activators are supported by VisiBroker. However, these activators can be used only with BOA, not POA. The POA uses servant activators and servant locators in place of object activators.

In this release of VisiBroker, the Portable Object Adaptor (POA) supports the features that were provided by the BOA in VisiBroker 3.x releases. For backward compatibility reasons, you may still use the object activators with your code.

## Naming objects under the BOA

Though the BOA is deprecated in VisiBroker, 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 server process that provided different interfaces, all of which had the same object name, but in the current version of VisiBroker, different interfaces may not have string-equivalent names.

# 31

# Using object activators

This section describes how to use the VisiBroker object activators.

In this release, as well as the VisiBroker 4.1 release and later, the Portable Object Adaptor (POA) supports the features that were provided by the BOA in the VisiBroker 3.x and 4.0 releases. For backward compatibility reasons, you may still use the object activators as described in this section with your code. For more information on how to use the BOA activators with this release, see "Using the BOA with VisiBroker."

## Deferring object activation

You can defer activation of multiple object implementations using service activation with a single `Activator` when a server needs to provide implementations for a large number of objects.

## Activator interface

You can derive your own interface from the `Activator` class. This allows you to implement the pure virtual `activate` and `deactivate` methods that the VisiBroker ORB will use for the `AccountImpl` object. You can then delay the instantiation of the `AccountImpl` object until the BOA receives a request for that object. It also allows you to provide clean-up processing when the BOA deactivates the object.

The following code sample shows the `Activator` class.

```
class Activator {
   public:
      virtual CORBA::Object_ptr activate(
         extension::ImplementationDef impl)=0;
      virtual void deactivate(
         Object_ptr, extension::ImplementationDef_ptr impl)=0;
};
```

The following code sample shows you how to create an `Activator` for the `AccountImpl` interface.

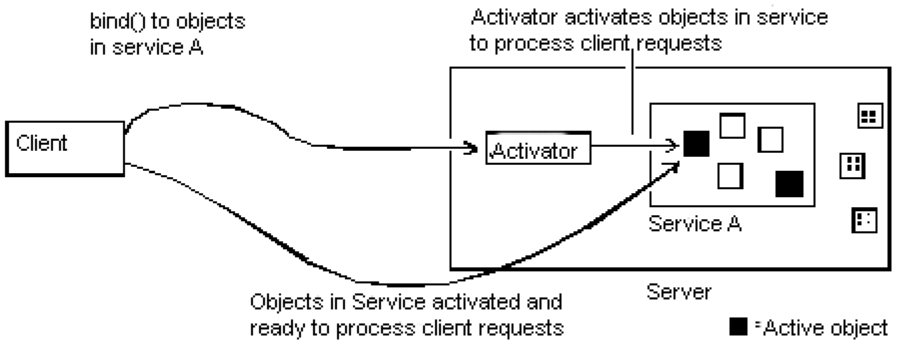
```
class extension {
   ...
   class AccountImplActivator : public extension::Activator {
      public:
         virtual CORBA::Object_ptr activate(
               CORBA::ImplementationDef_ptr impl);
         virtual void deactivate(CORBA::Object_ptr,
               CORBA::ImplementationDef_ptr impl);
   };
   CORBA::Object_ptr AccountImplActivator::activate(
         CORBA::ImplementationDef_ptr impl) {
      // When the BOA needs to activate us, instantiate the AccountImpl object.
      extension::ActivationImplDef* actImplDef =
               extension::ActivationImplDef::_downcast(impl);
      CORBA::Object_var obj = new AccountImpl(actImplDef->object_name());
      return CORBA::_duplicate(obj);
   }
   void AccountImplActivator::deactivate(CORBA::Object_ptr obj,
         CORBA::ImplementationDef_ptr impl) {
      // When the BOA deactivates us, release the Account object.
      obj->_release;
   }
}
```

## Using the service activation approach

Service activation can be used when a server needs to provide implementations for a large number of objects (commonly thousands of objects, possibly millions) but only a small number of implementations need to be active at any specific time. The server can supply a single `Activator` which is notified whenever any of these subsidiary objects are needed. The server can also deactivate these objects when they are not in use.

For example, you might use service activation for a server that loads object implementations whose states are stored in a database. The `Activator` is responsible for loading all objects of a given type or logical distinction. When VisiBroker ORB requests are made on the references to these objects, the `Activator` is notified and creates a new implementation whose state is loaded from the database. When the `Activator` determines that the object should no longer be in memory and, if the object had been modified, it writes the object's state to the database and releases the implementation.

**Figure 31.1** Process of Deferring Activation for a Service


bind() to objects in service A
Activator activates objects in service to process client requests
Client
Activator
Service A
Server
Objects in Service activated and ready to process client requests
■ =Active object

### Deferring object activation using service activators

Assuming the objects that will make up the service have already been created, the following steps are required to implement a server that uses service activation:

**1** Define a service name that describes all objects activated and deactivated by the `Activator`.

2 Provide implementations for the interface which are service objects, rather than persistent objects. This is done when the object constructs itself as an activatable part of a service.

3 Implement the `Activator` which creates the object implementations on demand. In the implementation, you derive an Activator interface from `extension::Activator`, overriding the `activate` and `deactivate` methods.

4 Register the service name and the Activator interface with the BOA.

## Example of deferred object activation for a service

The following sections describe the `odb` example for service activation which is located in the following VisiBroker directory:

```
<install_dir>/examples/Vbroker/boa/odb
```

This directory contains the following files:

| Name | Description |
|---|---|
| `odb.idl` | IDL for DB and DBObject interfaces. |
| `Server.C` | Creates objects using service activators, returns IORs for the objects, and deactivates the objects. |
| `Creator.C` | Calls the DB interface to create 100 objects and stores the resulting stringified object references in a file (`objref.out`). |
| `Client.C` | Reads the stringified object references to the objects from a file and makes calls on them, causing the activators in the server to create the objects. |
| `Makefile` | When `make` or `nmake` (on Windows) is invoked in the `odb` subdirectory, builds the following client and server programs: `Server.exe`, `Creator.exe`, and `Client.exe`. |

The odb example shows how an arbitrary number of objects can be created by a single service. The service alone is registered with the BOA, instead of each individual object, with the reference data for each object stored as part of the IOR. This facilitates object-oriented database (OODB) integration, since you can store object keys as part of an object reference. When a client calls for an object that has not yet been created, the BOA calls a user-defined `Activator`. The application can then load the appropriate object from persistent storage.

In this example, an `Activator` is created that is responsible for activating and deactivating objects for the service named "DBService." References to objects created by this `Activator` contain enough information for the VisiBroker ORB to relocate the `Activator` for the `DBService` service, and for the `Activator` to recreate these objects on demand.

The `DBService` service is responsible for objects that implement the `DBObject` interface. An interface (contained in `odb.idl`) is provided to enable manual creation of these objects.

### odb.idl interface

The `odb.idl` interface enables manual creation of objects that implement the `DBObject` odb interface.

```
interface DBObject {
   string get_name();
};
typedef sequence<DBObject> DBObjectSequence;
interface DB {
   DBObject create_object(in string name);
};
```

The `DBObject` interface represents an object created by the `DB` interface, and can be treated as a service object.

`DBObjectSequence` is a sequence of `DBObject`s. The server uses this sequence to keep track of currently active objects.

The DB interface creates one or more DBObjects using the create_object operation. The objects created by the DB interface can be grouped together as a service.

### Implementing a service-activated object

The idl2cpp compiler generates two kinds of constructors for the skeleton class _sk_DBObject from boa/odb/odb.idl. The first constructor is for use by manually-instantiated objects; the second constructor enables an object to become part of a service. As shown below, the implementation of DBObject constructs its base _sk_DBObject method using the service constructor, rather than the object_name constructor typically used for manually-instantiated objects. By invoking this type of constructor, the DBObject constructs itself as a part of a service called DBService.

```
class DBObjectImpl: public _sk_DBObject {
   private:
      CORBA::String_var   _name;
   public:
      DBObjectImpl(const char *nm, const CORBA::ReferenceData& data)
      : _sk_DBObject("DBService", data), _name(nm) {}
   ...
};
```

The base constructor requires a service name as well as an opaque CORBA::ReferenceData value. The Activator uses these parameters to uniquely identify this object when it must be activated due to client requests. The reference data used to distinguish among multiple instances in this example consists of the range of numbers from 0 to 99.

### Implementing a service activator

Normally, an object is activated when a server instantiates the classes implementing the object, and then calls BOA::obj_is_ready followed by BOA::impl_is_ready . To defer activation of objects, it is necessary to gain control of the activate method that the BOA invokes during object activation. You obtain this control by deriving a new class from extension::Activator and overriding the activate method, using the overridden activate method to instantiate classes specific to the object.

In the odb example, the DBActivator class derives from extension::Activator , and overrides the activate and deactivate methods. The DBObject is constructed in the activate method.

Th following code sample is an example of overriding activate and deactivate.

```
class DBActivator: public extension::Activator {
      virtual CORBA::Object_ptr activate(CORBA::ImplementationDef_ptr impl);
      virtual void deactivate(CORBA::Object_ptr,
            CORBA::ImplementationDef_ptr impl );
   public:
      DBActivator(CORBA::BOA_ptr boa) : _boa(boa) {}
   private:
      CORBA::BOA_ptr _boa;
};
```

When the BOA receives a client request for an object under the responsibility of the Activator, the BOA invokes the activate method on the Activator. When calling this method, the BOA uniquely identifies the activated object implementation by passing the Activator an ImplementationDef parameter, from which the implementation can obtain the CORBA::ReferenceData , the requested object's unique identifier.

As shown in the following code sample, the DBActivator class creates an object based on its CORBA::ReferenceData parameter.

```
CORBA::Object_ptr DBActivator::activate(CORBA::ImplementationDef_ptr impl) {
   extension::ActivationImplDef* actImplDef =
         extension::ActivationImplDef::_downcast(impl);
   CORBA::ReferenceData_var id(actImplDef->id());
   cout << "Activate called for object=[" << (char*) id->data()
```

```
         << "]" << endl;
      DBObjectImpl *obj = new DBObjectImpl((char *)id->data(), id);
      _impls.length(_impls.length() + 1);
      _impls[_impls.length()-1] =  DBObject::_duplicate(obj);
      _boa->obj_is_ready(obj);
      return obj;
   }
```

## Instantiating the service activator

The `DBActivator` service activator is responsible for all objects that belong to the `DBService` service. All requests for objects of the `DBService` service are directed through the `DBActivator` service activator. All objects activated by this service activator have references that inform the VisiBroker ORB that they belong to the `DBService` service.

As shown in the following code sample, the `DBActivator` service activator is created and registered with the BOA using the `BOA::impl_is_ready` call in the main server program.

```
int main(int argc, char **argv) {
   CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
   CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);
   MyDB db("Database Manager");
   boa->obj_is_ready(&db);
   DBObjectImplReaper reaper;
   reaper.start();
   cout << "Server is ready to receive requests" << endl;
   boa->impl_is_ready("DBService", new DBActivator(boa));
   return(0);
}
```

**Note**

The call to `BOA::impl_is ready` is a variation on the usual call to `BOA::impl_is_ready` . It takes two arguments:

- Service name.
- Instance of an Activator interface that will be used by the BOA to activate objects belonging to the service.

## Using a service activator to activate an object

Whenever an object is constructed, `BOA::obj_is_ready` must be explicitly invoked in `DBActivator::activate`. There are two calls to `BOA::obj_is_ready` in the server program. One call occurs when the server creates a service object and returns an IOR to the creator program.

```
DBObject_ptr create_object(const char *name) {
   char ref_data[100];
   memset(ref_data,'\0',100);
   sprintf(ref_data, "%s", name);
   CORBA::ReferenceData id(100, 100, (CORBA::Octet *)ref_data);
   DBObjectImpl *obj = new DBObjectImpl(name, id);
   _boa()->obj_is_ready(obj);
   _impls.length(_impls.length() + 1);
   _impls[_impls.length()-1] =  DBObject::_duplicate(obj);
   return obj;
}
```

The second occurrence of `BOA::obj_is_ready` is in `DBActivator::activate`, and this needs to be explicitly called.

## Deactivating service-activated object implementations

The main use for service activation is to provide the illusion that a large number of objects are active within a server, but to have only a small number of these objects actually active at any given moment. To support this model, the server must be able to temporarily remove objects from use. The multithreaded `DBActivator` example program contains a reaper thread that deactivates all `DBObjectImpls` every 30 seconds. The `DBActivator` simply releases the object reference when the `deactivate` method is invoked. If a new client request arrives for a deactivated object, the VisiBroker ORB informs the `Activator` that the object should be reactivated.

```
// static sequence of currently active Implementations
static VISMutex         _implMtx;
static DBObjectSequence _impls;
// updated DBActivator to store activated implementations
// in the global sequence.
class DBActivator: public extension::Activator {
   virtual CORBA::Object_ptr activate(CORBA::ImplementationDef_ptr impl) {
      extension::ActivationImplDef* actImplDef =
            extension::ActivationImplDef::_downcast(impl);
      CORBA::ReferenceData_var id(actImplDef->id());
      DBObjectImpl *obj = new DBObjectImpl((char *)id->data(), id);
      VISMutex_var lock(_implMtx);
      _impls.length(_impls.length() + 1);
      _impls[_impls.length()-1] =  DBObject::_duplicate(obj);
      return obj;
   }
   virtual void deactivate(CORBA::Object_ptr,
      CORBA::ImplementationDef_ptr impl) {
      obj->_release();
   }
};
// Multi-threaded Reaper for destroying all activated
// objects every 30 seconds.
class DBObjectImplReaper : public VISThread {
   public:
      // Reaper methods
      virtual void start() {
         run();
      }
      virtual CORBA::Boolean startTimer() {
         vsleep(30);
         return 1;
      virtual void begin() {
         while (startTimer()) {
            doOneReaping();
         }
      }
   protected:
      virtual void doOneReaping() {
         VISMutex_var lock(_implMtx);
         for (CORBA::ULong i=0; i < _impls.length(); i++) {
            // assigning nil into each element will release
            // the reference stored in the _var
            DBObject_var obj = DBObject::_duplicate(_impls[i-1];
            _impls[i] = DBObject::_nil();
            CORBA::BOA_var boa = obj->_boa();
            boa->deactivate_obj(obj);
         }
         _impls.length(0);
      }
};
```

# 32

# Real-Time CORBA Extensions

This section describes the Real-Time CORBA extensions, as defined in the Real-Time CORBA 1.0 Specification, supported by VisiBroker for C++. It also explains how to apply these Real-Time CORBA extensions in application code.

**Note**

Real-Time CORBA Extensions support is available only on the following platforms:

- Solaris
- HP–UX
- AIX
- Linux

## Overview

VisiBroker for C++ provides the following Real-Time CORBA extensions:

- **Real-Time ORB**
  Used to manage the creation and destruction of other Real-Time CORBA entities, such as Threadpools and Mutexes.
- **Real-Time Object Adapters**
  Enhanced Portable Object Adapters (POAs), which work with Threadpools and have a number of configurable Real-Time CORBA properties.
- **Real-Time CORBA Priority**
  A platform-independent priority scheme that is used to control the priority of threads related to the VisiBroker application. Specifying priorities in terms of the Real-Time CORBA priority scheme, instead of the priority scheme of a particular OS, allows applications to be developed that schedule real-time activities consistently across machines running different Operating Systems, both Real-Time and non-Real-Time. 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 OS. You can install a Priority Mapping to control the way the priorities are mapped or you can 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 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, which 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.
- **Control of Internal ORB Thread Priorities**
  Mechanisms to allow range limitation and explicit control of the priorities of additional threads used internally within the ORB.

**Note**

Depending on the underlying Operating System, the control of Thread Priorities might require superuser (*root*) privileges.

## 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:

```
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 in "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 fails, and a system exception is raised.

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.

**Note**

To support Real-Time CORBA Extensions, the VisiBroker for C++ ORB has to operate in a special 'real-time compatible' mode, the behavior and semantics of which differ from the regular mode of operation. Since obtaining an `"RTORB"` reference automatically puts the ORB in this special mode, you should **obtain an `"RTORB"` reference as early as possible** in your application code to avoid any possible inconsistency in behavior.

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: `priority_model`, `threadpool`, `visthread` and `rtmutex`.

```
#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 it 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:

```
// 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 "Threadpools" and "Real-Time CORBA Priority Models".

If any of these Real-Time properties is not configured by the application at the time of POA creation, the ORB initializes 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 "Real-Time CORBA 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, improves application portability, and simplifies future extension to a mixed OS environment.

For consistency and portability, **Real-Time CORBA applications must 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**

As per the Real-Time CORBA Specification, numerically higher `RTCORBA::Priority` values are defined to be of higher priority.

In practice, an application does not need to use the entire range of valid `RTCORBA::Priority` values (0 to 32767.) A smaller range, which 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 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. For example, the Pthreads priority scheme used by some Operating Systems is POSIX compliant with priorities in the range 0 to 31. In Real-Time CORBA, this is referred to as the *Native Thread Priority Scheme* and the priority values are referred to as *Native Priority* values.

As the Real-Time CORBA application describes 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 can also be used directly by the application, but this should only occur in special circumstances. This is discussed further in "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 might 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 for C++, it might be appropriate to skip the rest of this section, and learn about the rest of the Real-Time CORBA features (beginning with "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, depending on the OS, the valid range is a subset of this range.

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:

```
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`, the `to_CORBA` 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, which 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 might 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 might fail to map values within that range as well. (The default Priority Mapping chooses to map all values in the 0 to 31 range.)
- Lower `RTCORBA::Priority` values should always map to lower importance Native Priority values, and higher to higher. Note that in the case of a Pthreads 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. The convention maintains 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.
- `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` usually return the same value (or fail to map) every time they are called with the same input value.
- `to_native` and `to_CORBA` are usually reverse mappings of one another.
- The ranges of `RTCORBA::Priority` and Native Priority values that are mapped are usually each a single contiguous range of priority values.

## Default Priority Mapping

VisiBroker 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 in “Replacing the Default Priority Mapping”.

**Note**

Only one Priority Mapping can be installed at any one time on a given VisiBroker application. The act of installing one Priority Mapping automatically un-installs the previously installed Priority Mapping (usually 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 is raised 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 native operating system—the "Native Priority range".
- 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. The following table shows the `RTCORBA::Priority` default mappings for the supported operating systems.

| Operating System | RTCORBA::Priority range | Native Priority range |
|---|---|---|
| HPUX | 0–31 | 0–31 |
| Solaris | 0-31 | 10-41 |
| AIX | 0-31 | 28-59 |
| Linux | 0-31 | 30-61 |

The default Priority Mapping is defined within the ORB, and hence the source code for it is not included in the VisiBroker release. The source code for the mapping is shown here, however, to show exactly how this mapping behaves:

```
VisiBroker for C++ Default 'to_corba' Priority Mapping CORBA::Boolean
VISDefaultPriorityMapping::to_CORBA( RTCORBA::NativePriority native_priority,

        RTCORBA::Priority &amp;corba_priority )
#if defined(SOLARIS)
        native_priority = 10 + corba_priority;  // 0 -> 10, 31 -> 41
#if defined(AIX)
        native_priority = 28 + corba_priority;  // 0 -> 28, 31 -> 59
#elif defined(HPUX_11)
        native_priority = corba_priority;       // 0 ->  0, 31 -> 31
#elif defined(__linux)
        native_priority = 30 + corba_priority;  // 0 -> 30, 31 -> 61
#else
#       error Supported OS not detected
#endif
        return (CORBA::Boolean) 1;
    }
}
```

```
// VisiBroker for C++ Default 'to_corba' Priority Mapping
CORBA::Boolean
VISDefaultPriorityMapping::to_native( RTCORBA::Priority corba_priority,
        RTCORBA::NativePriority &amp;native_priority )
#if defined(SOLARIS)
    if ((native_priority < 10) || (native_priority > 41))
#elif defined(AIX)
    if ((native_priority <  28) || (native_priority > 59))
#elif defined(HPUX_11)
    if ((native_priority <  0) || (native_priority > 31))
#elif defined(__linux)
    if ((native_priority < 30) || (native_priority > 61))
#else
#   error Supported OS not detected
#endif
    {
        return (CORBA::Boolean) 0;
    }
    else
    {
#if defined(SOLARIS)
        corba_priority = native_priority - 10;    // 10 -> 0, 41 -> 31
#elif defined(AIX)
        corba_priority = native_priority - 28; // 28 -> 0, 59 -> 31

#elif defined(HPUX_11)
        corba_priority = native_priority;        //  0 ->  0, 31 -> 31
#elif defined(__linux)
        corba_priority = native_priority - 30;    // 30 -> 0, 61 -> 31
#else
#       error Supported OS not detected
#endif
        return (CORBA::Boolean) 1;
    }
}

// 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 can be installed at any one time on a particular system. The act of installing one Priority Mapping automatically uninstalls the previously installed Priority Mapping (usually the default Priority Mapping.)

The application might 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 10 to 41, and replace it with the range 50 to 81 (higher importance) or 200 to 231 (even higher importance.)

- To have more or fewer `RTCORBA::Priority` values in the range of valid (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 fewer Native Priority values in the range of valid (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 can 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. This means ensuring that `RTCORBA::Priority` 0 maps to the numerically smallest Native Priority value (of the sub-range that is being mapped to.) This maintains consistency with Real-Time CORBA applications developed across OSs. 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 `RTCORBA::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 writing and installing a new Priority Mapping, look at the files `mapping.h` and `mapping.C` in the `threadpool` example included in 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 built with a `VisiBroker` application and static constructor initialization takes place during the execution of the application, it is the initialization of this instance that installs the mapping.

## Using Native Priorities in VisiBroker Application Code

Although applications must use Real-Time CORBA Priority to discern 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 needs to discern Native Priority. Examples include configuring a sub-system outside of the CORBA application, which only knows about the Native Priority scheme, or using an OS call directly, which takes a Native Priority value as a parameter. In these cases, it might be necessary to translate between Real-Time CORBA and Native Priority in the application. To allow this, VisiBroker for C++ provides the static `instance` method on the class `RTCORBA::PriorityMapping`. This method returns a pointer to the currently installed Priority Mapping.

Using this method, it is guaranteed to the application code that any priority mapping method calls it makes are executed on the currently installed mapping, regardless of the internal implementation details of the mapping. This allows the code to continue to work even if the installed mapping is changed. The following example uses 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 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 might use. In systems where the total number of threads that can 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 can be created dynamically, to execute CORBA invocations received when all the static threads are currently in use. The number can be zero, in which case no threads can 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 is executed depends on the Real-Time CORBA Priority Model in use. See "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, which 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 for C++, and the value of these arguments is ignored.

If `dynamic_threads` is greater than zero, so that threads can 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 might 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 can 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.

**Note**

The semantics of `dynamic_threads` value here in the Real-Time context differs from that of the *Dispatcher Thread pool* `threadMax` value. (See "Thread pool dispatch policy") A `dynamic_threads` value of zero means that, even if needed, no additional threads are created for the RTCORBA Threadpool. In contrast, a `threadMax` value of zero for the dispatcher threadpool means that the VisiBroker ORB has the freedom to create additional threads, as and when required.

## Association of an Object Adapter with a Threadpool

Every POA created using VisiBroker for C++ is associated with a Threadpool. Each Threadpool, on the other hand, can 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. The following example associates 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
```

`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 is raised.

## 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.

The General Threadpool is 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 can 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 (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 raised.

## Real-Time CORBA Current

Real-Time CORBA defines a Real-Time CORBA Current interface to provide access to the CORBA priority of a thread. The following sample shows the RTCORBA::Current interface.

```
module RTCORBA {
    interface Current : CORBA::Current {
        attribute Priority base_priority;
    };
};
```

A Real-Time CORBA Priority can 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 is immediately 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 configured to support the ‘Client Priority Propagation’ Priority Model. (See “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 is raised 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 is raised 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 is raised 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 shown in the following example:

```
// Obtain the RTCORBA::Current reference
CORBA::Object_var ref;

VISTRY
{
    ref = orb->resolve_initial_references("RTCurrent");
}
VISCATCH
{
    // handle error here
}
VISEND_CATCH

// Narrow the RTCORBA::Current reference
RTCORBA::Current_ptr rtcurrent;
VISTRY
{
    rtcurrent = RTCORBA::Current::_narrow(ref);
}
VISCATCH(CORBA::Exception, e)
{
    // handle error here
}
VISEND_CATCH
```

Note that the `RTCORBA::Current` reference only needs to be obtained once. The same variable can 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 of the following values:

- `RTCORBA::CLIENT_PROPAGATED`

  To select the Client Propagated Priority Model.

- `RTCORBA::SERVER_DECLARED`

  To select the Server Declared Priority 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 Propagated Priority 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 Priority 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 Priority 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 Priority 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 "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 Propagated Priority Model is selected, with a default priority of 7 (for invocations from non-Real-Time Clients).

```
// Create Real-Time CORBA Priority Model Policy

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_model` example included with the VisiBroker installation for further examples of configuring the two different Priority Models.

## Setting Priority at the Object Level

When the Server Declared Priority 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 can 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.

The Real-Time CORBA Specification defines these methods in the `RTPortableServer` module. However, VisiBroker for C++ defines these operations 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_srvr.C` in the `priority_model` example included with VisiBroker.

## Real-Time CORBA Mutex API

VisiBroker 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 and the calling thread will be assigned ownership of the Mutex object.

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.

## Control of Internal ORB Thread Priorities

VisiBroker for C++ allows the application to control the priority of the threads that the ORB creates for internal use.

The internal ORB threads are:

- DSUser thread

  A single DSUser thread is created the first time the ORB attempts to communicate with the VisiBroker Smart Agent (`osagent`). 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 Smart Agent. The thread name is 'VISDSUser'.

- Listener threads

  Listener Threads are 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 thread 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 is created so that it can be used.) Garbage Collection Threads have thread 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.

There are two ways of configuring the priority of the different types of internal ORB threads:

- On a per-type basis (and in some cases a per-instance basis), through VisiBroker properties.
- 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.

### Configuring Individual Internal ORB Thread Priorities

The priority of different types (and in one case, different instances) of internal ORB threads can 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.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 Manager. 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.agent.threadPriority`

  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 a VisiBroker Smart Agent (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.

**Note**

In earlier versions of the VisiBroker-RT product, the `vbroker.agent.threadPriority` property was called `vbroker.dsuser.thread.priority`. The name has been changed to keep it in sync with other VisiBroker Smart Agent property names. However, the old property name is still supported and you can continue to use it in existing deployments.

## 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, as shown in the following 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 defaults to running at the maximum priority that is specified.

If the range is invalid, the `ORB_init` call fails and raises 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 raised. 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 raised.

# 33

# CORBA exceptions

This section provides information about CORBA exceptions that can be thrown by the VisiBroker ORB, and explains possible causes for VisiBroker throwing them.

## CORBA exception descriptions

The following table lists CORBA exceptions, and explains reasons why the VisiBroker ORB might throw them.

| Exception | Explanation | Possible causes |
|---|---|---|
| `CORBA::BAD_CONTEXT` | 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. |
| `CORBA::BAD_INV_ORDER` | The necessary prerequisite operations have not been called prior to the offending operation request. | An attempt to call the`CORBA::Request::get_response()` or`CORBA::Request::poll_response()` methods may have occurred prior to actually sending the request. An attempt to call the `exception::get_client_info()` 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. An operation was called on the VisiBroker ORB that was already shut down. |
| `CORBA::BAD_OPERATION` | An invalid operation has been performed. | 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. The `CORBA::Request::return_value()` 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 `CORBA::Request::set_return_type()` method. |

(continued)

| Exception | Explanation | Possible causes |
|---|---|---|
| `CORBA::BAD_PARAM` | A parameter passed to the VisiBroker ORB is invalid. | Sequences throw `CORBA::BAD_PARAM` if an access is attempted to an invalid index. Make sure you use the `length()` method to set the length of the sequence before storing or retrieving elements of the sequence.<br>The VisiBroker ORB throws this exception if an invalid `Object_ptr` is passed as an `in` argument; for example, if a `nil` reference is passed.<br>An attempt may have been made to send a `NULL` pointer where the IDL to C++ language mapping requires an initialized C++ object to be sent. For example, attempting to return `NULL` as a return value or `out` parameter from a method that should be returning a `sequence` will throw this exception. In this case a new `sequence` (probably of length 0) should be returned instead. The types which cannot be sent with the C++ `NULL` value include `Any`, `Context`, `struct`, or `sequence`.<br>An attempt may have been made to insert a `nil` object reference into an `Any`.<br>An attempt was made to send a value that is out of range for an enumerated data type.<br>An attempt may have been made to construct a `TypeCode` with an invalid `kind` value.<br>Using the DII and one way method invocations, an `OUT` argument may have been specified. An interface repository throws this exception if an argument passed into an IR object's operation conflicts with its existing settings. See the compiler errors for more information. |
| `CORBA::BAD_QOS` | Quality of service cannot be supported. | Can be raised whenever an object cannot support the quality of service required by an invocation parameter that has a quality of service semantics associated with it. |
| `CORBA::BAD_TYPECODE` | The ORB has encountered a malformed type code. | |
| `CORBA::CODESET_INCOMPATIBLE` | 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. |
| `CORBA::COMM_FAILURE` | Communication is lost while an operation is in progress, after the request was sent by the client, but before the reply has been returned. | An existing connection may have closed due to failure at the other end of the connection.<br>Potentially, the operation was invoked.<br>When `COMM_FAILURE` occurs due to system exceptions, the system error number is set in the minor code of the `COMM_FAILURE`. Check the minor code against the system-specific error numbers (for example, in the `include/sys/errno.h` or `msdev\include\winerror.h` files). |
| `CORBA::DATA_CONVERSION` | The VisiBroker ORB cannot convert the representation of marshaled data into its native representation or vice-versa. | An attempt to marshal Unicode characters with`Output.write_char()` or `Output.write_string`fails. |
| `CORBA::FREE_MEM` | The VisiBroker ORB failed to free dynamic memory. | The memory segments that the VisiBroker ORB is trying to free may be locked. The heap could be corrupt. |
| `CORBA::IMP_LIMIT` | An implementation limit was exceeded in the VisiBroker ORB run time. | The VisiBroker 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. |

(continued)

| Exception | Explanation | Possible causes |
|---|---|---|
| `CORBA::INITIALIZE` | A necessary initialization has not been performed. | The `ORB_init()` method may not have been called. All clients must call the `ORB_init()` method prior to performing any VisiBroker ORB-related operations. This call is typically made immediately upon program startup at the top of the main routine. |
| `CORBA::INTERNAL` | An internal VisiBroker ORB error has occurred. | An internal VisiBroker ORB error may have occurred. For instance, the internal data structures of the VisiBroker 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 `get_interface()` 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 VisiBroker ORB will throw this exception if an object reference is obtained that contains no usable profiles. The `ORB::string_to_object()` method will throw this exception if the stringified object reference does not begin with the characters "`IOR:`". |
| `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. | This exception could be raised if an error occurred while trying to register a Resource. |
| `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 VisiBroker ORB raises this exception. A `MARSHAL` 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. | Indicates that even though the operation that was invoked exists (it has an IDL definition), no implementation for that operation exists. For example, a NO_IMPLEMENTATION is raised when a server doesn't exist or is not running when a client initiates a bind. |
| `CORBA::NO_MEMORY` | The VisiBroker ORB runtime has run out of memory. | |
| `CORBA::NO_PERMISSION` | The caller has insufficient privileges to complete an invocation. | The `Object::get_implementation()` and `BOA::dispose()` methods throw this exception if they are called on the client side. It is only valid to call these methods within the server that activated the object implementation.<br>An object other than the transaction originator has attempted `Current::commit()` or `Current::rollback()`. |

(continued)

| Exception | Explanation | Possible causes |
|---|---|---|
| `CORBA::NO_RESOURCES` | A necessary resource could not be acquired. | If a new thread cannot be created, this exception will be thrown. 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 `::socket()` or `::accept()` call. A client will similarly throw this exception if a `::connect()` call fails due to running out of file descriptors. Running out of memory may also throw this exception. |
| `CORBA::NO_RESPONSE` | A client attempts to retrieve the result of a deferred synchronous call, but the response for the request is not yet available. | If BindOptions are used to set timeouts, this exception is raised when send and receive calls do not occur within the specified time. |
| `CORBA::OBJ_ADAPTER` | An administrative mismatch has occurred. | A server has attempted to register itself with an implementation repository under a name that is already in use, or is unknown to the repository. The POA has raised an `OBJ_ADAPTER` error due to problems with the application's servant managers. |
| `CORBA::OBJECT_NOT_EXIST` | 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. For instance, if an attempt to bind to an object fails, or an auto-rebind fails, `OBJECT_NOT_EXIST` will be raised |
| `CORBA::PERSIST_STORE` | A persistent storage failure has occurred. | Attempts to establish a connection to a database has failed, or the database is corrupt. |
| `CORBA::REBIND` | 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 Rebind Policy has a value of `NO_REBIND`,`NO_CONNECT`, or `VB_NOTIFY_REBIND` and an invocation on a bound object reference results in an object forward or a location forward message. |
| `CORBA::TIMEOUT` | The VisiBroker ORB timed out an operation | When attempting to establish a connection or waiting for a request/reply, if the operation does not complete before the specified time, a TIMEOUT exception is thrown. `CORBA::TIMEOUT` has the following minor codes:<br>■ 0x56420001: connection timed out (could not connect within the connection timeout)<br>■ 0x56420002: request timed out (could not send the request within the timeout specified)<br>■ 0x56420003: Reply timed out (the reply was not received within the round trip timeout specified) |
| `CORBA::TRANSACTION_REQUIRED` | The request has a null transaction context, and an active transaction is required. | A method was invoked that must execute as part of a transaction, but no transaction was active on the client thread. |
| `CORBA::TRANSACTION_ROLLEDBACK` | The transaction associated with a request has already been rolled back, or marked for roll back. | A requested operation could not be performed because the transaction has already been marked for rollback. |
| `CORBA::TRANSACTION_MODE` | Raised by the VisiBroker ORB, when it detects a mismatch between the `TransactionPolicy` in the IOR and the current transaction mode. | |

(continued)

| Exception | Explanation | Possible causes |
|---|---|---|
| `CORBA::TRANSACTION_UNAVAILABLE` | Raised by the VisiBroker ORB, when it cannot process a transaction service context because its connection to the Transaction Service has been abnormally terminated. | |
| `CORBA::TRANSIENT` | An error has occurred, but the VisiBroker ORB believes it is possible to retry the operation. | A communications failure may have occurred and the VisiBroker 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 `BindOptions` are set to false with the `enable_rebind()` method, or the RebindPolicy is properly set. 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). When `TRANSIENT` exceptions occur due to system exceptions, the system error number is set in the minor code of the `COMM_FAILURE`. Check the minor code against the system-specific error numbers (for example, in the `include/sys/errno.h` or `msdev\include\winerror.h files`). |
| `CORBA::UNKNOWN` | The VisiBroker ORB could not determine the thrown exception. | The server throws something other than a correct exception, such as a Java runtime exception. There is an IDL mismatch between the server and the client, and the exception is not defined in the client program. 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 `CORBA::Request`. Set the property `vbroker.orb.warn=2` on the server to see which runtime exception caused the problem. |
| `CORBA::UnknownUser` | A user exception has been received, but the client has no compile-time knowledge of that exception. | 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 VisiBroker ORB has no way to unmarshal the exception on its own. |

**Table 33.1**

| System exception | Minor code | Explanation |
|---|---|---|
| `BAD_PARAM` | 1 | Failure to register, unregister, or lookup the value factory |
| `BAD_PARAM` | 2 | RID already defined in the interface repository |
| `BAD_PARAM` | 3 | Name already used in the context in the interface repository |
| `BAD_PARAM` | 4 | Target is not a valid container |
| `BAD_PARAM` | 5 | Name clash in inherited context |
| `BAD_PARAM` | 6 | Incorrect type for abstract interface |
| `MARSHAL` | 1 | Unable to locate value factory |
| `NO_IMPLEMENT` | 1 | Missing local value implementation |
| `NO_IMPLEMENT` | 2 | Incompatible value implementation version |
| `BAD_INV_ORDER` | 1 | Dependency exists in the interface repository preventing the destruction of the object |
| `BAD_INV_ORDER` | 2 | Attempt to destroy indestructible objects in the interface repository |
| `BAD_INV_ORDER` | 3 | Operation would deadlock |
| `BAD_INV_ORDER` | 4 | VisiBroker ORB has shut down |
| `OBJECT_NOT_EXIST` | 1 | Attempt to pass a deactivated (unregistered) value as an object reference |

## Heuristic OMG-specified exceptions

A *heuristic* decision is a unilateral decision made by a participant in a transaction to commit or rollback updates without first obtaining the consensus outcome determined by the VisiTransact Transaction Service. See "Transaction completion" for more information about heuristics.

The following table lists heuristic exceptions as defined by the OMG CORBAservices specification, and explains reasons why they might be thrown.

| Exception | Description | Possible causes |
|---|---|---|
| `CosTransactions::HeuristicCommit` | A heuristic decision was made and all relevant updates have been committed by the Resource. | The VisiTransact Transaction Service invoked `rollback()` on a Resource object that already made a heuristic decision to commit its work. The Resource raises the `HeuristicCommit` exception to indicate its state to the VisiTransact Transaction Service. |
| `CosTransactions::HeuristicHazard` | A Resource may or may not have made a heuristic decision, and does not know if all relevant updates have been made. For updates that are known, all have been committed or rolled back. This exception takes priority over `HeuristicMixed`. | The VisiTransact Transaction Service invokes `commit()` or `rollback()` on a Resource object that may or may not have made a heuristic decision.<br>The Resource raises this exception to indicate to the VisiTransact Transaction Service that its own state is not entirely known. The VisiTransact Transaction Service returns this exception to the application if it does not know if all Resources have made updates. |
| `CosTransactions::HeuristicMixed` | A heuristic decision was made, and some relevant updates have been committed, and others have been rolled back. | The VisiTransact Transaction Service invokes `commit()` or `rollback()` on a Resource object that has made a heuristic decision, but not made all the relevant updates.<br>The Resource raises this exception to indicate to the VisiTransact Transaction Service that its state is not entirely consistent. The VisiTransact Transaction Service returns this exception to the application if it receives mixed responses from Resources. |
| `CosTransactions::HeuristicRollback` | A heuristic decision was made and all relevant updates have been rolled back by the Resource. | The VisiTransact Transaction Service invokes `commit()` on a Resource object that has made a heuristic decision to rollback its work. The Resource raises the `HeuristicRollback` exception to indicate its state to the VisiTransact Transaction Service. |

## Other OMG-specified exceptions

The following table lists other exceptions as defined by the OMG CORBAservices specification, and explains reasons why the VisiTransact Transaction Service might throw them. For more information about, see "Overview of transaction processing."

| Exception | Description | Possible causes |
|---|---|---|
| `CosTransactions::Inactive` | The transaction has already been prepared or terminated. | This exception could be raised if `register_synchronization()` is invoked after the transaction has already been prepared. |
| `CosTransactions::InvalidControl` | An invalid Control has been passed. | This exception is raised when `resume()` is invoked and the parameter is not a null object reference, and is also not valid in the current execution environment. |

| Exception | Description | Possible causes |
|---|---|---|
| `CosTransactions::NotPrepared` | A Resource has not been prepared. | An invocation of `replay_completion()` or `commit()` on a Resource that has not yet prepared will result in this exception. |
| `CosTransactions::NoTransaction` | No transaction is associated with the client thread. | The `commit()`, `rollback()`, or `rollback_only()` methods may raise this exception if there is no transaction associated with the client thread at invocation. |
| `CosTransactions::NotSubtransaction` | The current transaction is not a subtransaction. | This exception is not raised by VisiTransact Transaction Manager since nested transactions are not supported. The `NoTransaction` exception is raised instead. |
| `CosTransactions::SubtransactionsUnavailable` | The client thread already has an associated transaction. The VisiTransact Transaction Service does not support nested transactions. | A subsequent `begin()` invocation was performed after a transaction was already begun. If your transactional object needs to operate within a transaction, it must first check to see if a transaction has already begun before invoking begin().<br>The `create_subtransaction()` method was invoked, but VisiTransact Transaction Manager does not support subtransactions. |
| `CosTransactions::SynchronizationUnavailable` | The Coordinator does not support Synchronization objects. | This exception is not raised by VisiTransact Transaction Manager since Synchronization objects are supported. |
| `CosTransactions::Unavailable` | The requested object cannot be provided. | The Control object cannot provide the Terminator or Coordinator objects when `Control::get_terminator()` or `Control::get_coordinator()` are invoked.<br>The VisiTransact Transaction Service restricts the availability of the `PropagationContext`, and will not return it upon an invocation of `Coordinator::get_txcontext()`. |
| `CORBA::WrongTransaction` | Raised by the ORB when returning the response to a deferred synchronous request. This exception can only be raised if the request is implicitly associated with the current transaction at the time the request was issued. | The `get_response()` and `get_next_response()` methods may raise this exception if the transaction associated with the request is not the same as the transaction associated with the invoking thread. |

34

# VisiBroker Pluggable Transport Interface

VisiBroker for C++ provides a Pluggable Transport Interface, to support the use of transport protocols besides the ones inbuilt in the ORB 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.

## Pluggable Transport Interface Files

The VisiBroker Pluggable Transport Interface is delivered as a library and a supporting header file:

– binpluggable<bitmode>_<p>r_<version>.dll on Windows

– lib/libpluggable<bitmode>_<p>r.so.<version> on Solaris and Linux

– lib/libpluggable<bitmode>_<p>r.sl.<version> on HP–UX

– lib/libpluggable<bitmode>_<p>r.a.<version> on AIX

– The header file vptrans.h can be found in the include directory

where bitmode is 64 on 64 bit platforms, “p” refers to the standard C++ version, and “version” refers to the version of VisiBroker. The APIs of the library are exposed through the `vptrans.h` header file.

## 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 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. Calling a static function of the Pluggable Protocol Interface during the system initialization stage, before starting any CORBA server or client code, performs the registration.

## 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. Borland has a range of Profile ID tags registered with the OMG, and four of these tags are available for use by protocol plugins:

1 0x48454901 (“HEI\001”)

2 0x48454902 (“HEI\002”)

3 0x48454903 (“HEI\003”)

4 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 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.

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 ftp://ftp.omg.org/pub/docs/ptc/99–02–01.txt, 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

Two examples are provided in the examples/pluggable directory that illustrates how a plug-in transport could be implemented and how it could be used by a CORBA application. The example makes use of TCP/IP as transport to lay emphasis on the interface itself rather than to explain the intricacies of a transport layer.

## Implementing a New Transport

The following interfaces, exposed in the `vptrans.h header` file, need to be implemented.

### VISPTransConnection and VISPTransConnectionFactory

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.

The Factory class is used to create instances of the plug-in connection and needs to be registered with a registrar using the static *VISPTransRegistrar::addTransport* API.

```
class _VBPTEXPORT VISPTransConnection {
public:
...

// send data to remote peer
virtual void write(CORBA::Boolean _isFirst, const char* _data,
CORBA::ULong_offset,
CORBA::ULong _length, CORBA::ULongLong _timeout) = 0;

// read data sent from the connection from remote peer
virtual void read(CORBA::Boolean _isFirst,
char* _data, CORBA::ULong _offset,
CORBA::ULong _length,
CORBA::ULongLong _timeout) = 0;

// helpful for buffering transport to flush data immediately
virtual void flush() = 0;

// orderly close of the connection
virtual void close() = 0;

// communicate with remote peer listener to set up a new connection
virtual void connect(CORBA::ULongLong _timeout) = 0;

// should return unique Id (for this transport) for each connection instance
virtual CORBA::Long id() = 0;

// should return true if remote peer is still connected
virtual CORBA::Boolean isConnected() = 0;

// should return true if data is ready to be read
virtual CORBA::Boolean isDataAvailable() = 0;

// should return true if the transport can be used on reverse client-server
      setup
virtual CORBA::Boolean no_callback() = 0;

// should return true if transport cannot wait for next message. This
      makes the ORB
// use the bridge for timing out while waiting for next message
virtual CORBA::Boolean isBridgeSignalling() = 0;

// blocks till data arrives or timeout. Should return true if data is available
virtual CORBA::Boolean waitNextMessage(CORBA::ULong _timeout) = 0;

// Should return a copy of the profile describing the peer endpoint
virtual IOP::ProfileValue_ptr getPeerProfile() = 0;

// input peer profile telling the connection regarding its peer.
      Used while connecting
virtual void setupProfile(const char* prefix, VISPTransProfileBase_ptr peer)
      = 0;

};

class _VBPTEXPORT VISPTransConnectionFactory {
public:
...

// should return a new connection instance and return pointer to it
virtual VISPTransConnection_ptr create(const char* prefix) = 0;

};
```

## VISPTransListener and VISPTransListenerFactory

This class is used on 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.

The factory instance as with the connection should return instances of the implemented plug-in listener and should be registered using *VISPTransRegistrar::addTransportAPI.*

```
class _VBPTEXPORT VISPTransListener {
public:
...
// Called by ORB to establish link to the bridge, so that listener-ORB
// communication can occur
virtual void setBridge(VISPTransBridge* up) = 0;

// Should return a profile describing the listener endpoint
virtual IOP::ProfileValue_ptr getListenerProfile() = 0;

// Called when the ORB has completed reading a request for the given id
// and wants the listener to once again signal via the bridge on any
     new requests.
virtual void completedData(CORBA::Long id) = 0;

// Should return true if connection with given id has data ready to be read

virtual CORBA::Boolean isDataAvailable(CORBA::Long id) = 0;

// Called when the listener needs to tear down the endpoint and close
     all related
// active connections.
  virtual void destroy() = 0;
};

class _VBPTEXPORT VISPTransListenerFactory {
public:
...
// Makes an new instance of the listener and should return pointer to it
virtual VISPTransListener_ptr create(const char* propPrefix) = 0;

};
```

## VISPTransProfileBase and VISPTransProfileFactory

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 down casting with any C++ compiler, a ‘_downcast’ function must be provided that can test if the cast is legal or not.

## Additional classes—VISPTransBridge and VISPTransRegistrar

Two additional classes are provided by the Pluggable Transport Interface, that the user-provided transport plugin code will make calls to.

VISPTransBridge is a generic interface between the ORB and the transport plug-in to communicate various events. Some of the communications are:

1 Communicate to the ORB about a new connection request

2 Communicate to the ORB about new input data

**3** Communicate to the ORB about peer connection closure

```
Class _VBPTEXPORT VISPTransBridge {
public:

// Tell ORB about a new connection request passing the connection pointer
// Returns true if the ORB has accepted the connection, else false
virtual CORBA::Boolean addInput(VISPTransConnection_ptr con);

// Tell ORB of a new request on a connection. Typically this will start off the
// dispatch cycle
virtual void signalDataAvailable(CORBA::Long conId);

// Tell the ORB that the connection was closed by the remote peer.
virtual void closedByPeer(CORBA::Long conId)=0;
```

VISPTransRegistrar is the class that 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.

```
class _VBPTEXPORT VISPTransRegistrar {
public:
// register the transport identifier string and the three factories used
     to specific instances
// of this new transport
static void addTransport(const char* protocolName,
                          VISPTransConnectionFactory* connFac,
                          VISPTransListenerFactory*   listFac,
                           VISPTransProfileFactory*     profFac);

};
```

# 35

# VisiBroker Logging

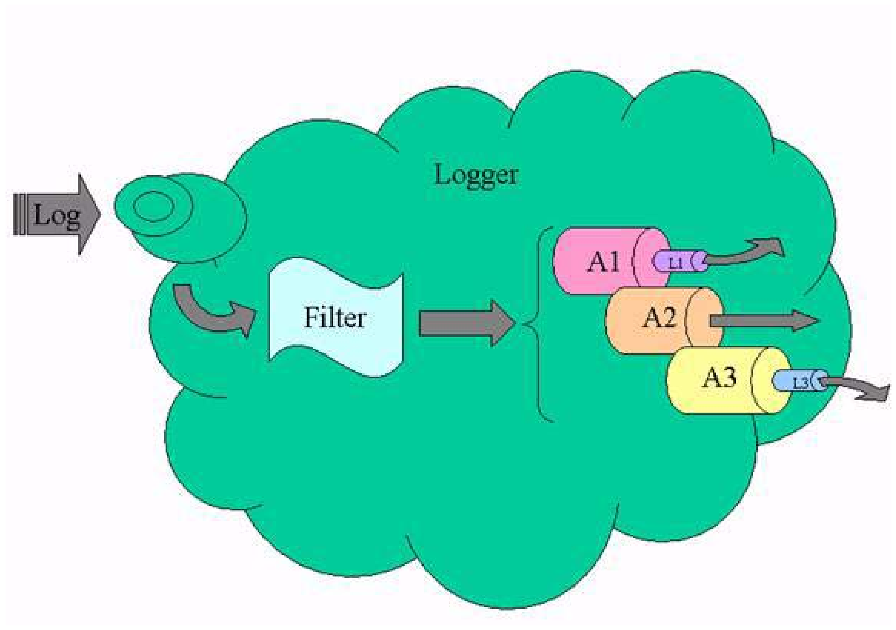
VisiBroker for C++ provides a logging mechanism, which allows applications to log messages and have them directed, via configurable logging forwarders called appenders, to appropriate destination or destinations. The ORB and all its services themselves use this mechanism for the output of any error, warning or informational messages. By using this feature, 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. Filters and layouts give additional capability for filtering and formatting the log messages.

Please note that this logging mechanism is different from the OMG's Telecom Logging Service (VisiTelcoLog). This is a lightweight C++ only logging library, which the ORB and its services including VisiTelcoLog use to log internal messages. The entire logging framework and the inbuilt appenders and layouts are in a shared library named vdlog<bitmode>_<p>r_<version>.dll on Windows, libvdlog<bitmode>_r.so.<version> on Solaris and Linux, libvdlog<bitmode>_<p>r.sl.<version> on HPUX and libvdlog<bitmode>_<p>r.a.<version> on AIX. Here "p" is for Standard C++ libraries, bitmode is "64" for 64 bit platforms, and version is the version of VisiBroker. The APIs of the library are exposed in vdlog.h header file.

## Logging Overview

Logging in VisiBroker employs one or more Logger objects, that applications (including the ORB) may log messages to. ORB and all its C++ services use a special Logger instance (the 'Default Logger' with the name "default"), which is created automatically the first time the ORB logs a message. 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 as said earlier.

A Logger in the framework can have one or more appenders associated with them to which all the log messages are sequentially forwarded and each appender inturn is responsible to output to desired destinations such as standard error, a file, over a network, an OMG Telecom Logging Service log etc. The figure below explains how a logger is associated with three appenders named A1, A2 and A3 where each appender is of a different type forwarding the logs to different types of destinations.



Along with forwarding the message, an appender may optionally choose to use a configured layout to format the log before outputting. Here, appenders A1 and A3 are using layouts L1 and L3. With no explicit configuration, by default, a logger logs to an appender of type "stdout" using a layout instance of type "simple".

The logger has another feature to allow it to filter the log messages. Prior to sending the log message to the list of appenders, the logger processes the message for its source name and log level using a filter and based on the outcome, does the logger decide whether to forward to the appenders or discard the message. By changing the settings of the allowable source names and their log levels, high filtering fidelity can be achieved.

In this chapter, the following topics will be covered:

- Logger manager
- Logging
- Filtering
- Custom appenders and layouts
- Configuration

## Logger Manager

Logger Manager is the starting point for using the functionality provided by the logger framework and one of the main functionality of the Logger Manager is to manage the lifecycle of Loggers. The Logger Manager is a singleton object and a reference to it is obtained by calling its static instance method. No reference counting is performed upon the Logger Manager. The code snippet below explains how the static instance function can be used to access the singleton logger manager object.

```
// Use static instance function to obtain Logger Manager reference
VISLoggerManager_ptr logger_manager = VISLoggerManager::instance();
VISLogger_ptr logger = logger_manager->get_default_logger();
...
// Alternatively, the Logger Manager reference may be obtained each time it
// is used. Here, for example, when calling its get_default_logger method
VISLogger_ptr logger = VISLoggerManager::instance()->get_default_logger();
```

Apart from giving access to the loggers, this singleton is also responsible for being a registrar for the custom appender and layout factories. It is also responsible for providing and configuring the global enabling switch and the verbosity level.

## Logging

As mentioned earlier applications can make use of the logging interface to log messages either using the default logger or a separate logger. All log messages to a single logger are bound to a common set of destinations and by using multiple loggers for logging, messages from different components could be output to various independent end points.

In the code snippet below, the server application is using the default logger to log its application specific messages. It is using the source name “bankagentserver” to identify its log messages from server code and “bankagentimpl” from the implementation code. Source names can be a very helpful tool for modularization and proves its worth during the filter configuration stages.

```
#define MYLOG(LVL, COMP, MESG) \
    if (VISDLoggerMgr::instance()->global_log_enabled()) { \
       VISDLoggerMgr::instance()->get_default_logger()->log( \
            LVL, COMP, MESG \
            , __FILE__, __LINE__ \
            ); \
    }

#define MYLOGDBG(COMP, MESG) \
  MYLOG(VISDLogLevel::DEBUG_, COMP, MESG)
#define MYLOGINF(COMP, MESG) \
  MYLOG(VISDLogLevel::INFO_, COMP, MESG)

#define BAS "bankagentserver"
#define BAS_LOGDBG(MESG) MYLOGDBG(BAS,MESG)
#define BAS_LOGINF(MESG) MYLOGINF(BAS,MESG)

#define BAI "bankagentimpl"
#define BAI_LOGDBG(MESG) MYLOGDBG(BAI,MESG)
#define BAI_LOGINF(MESG) MYLOGINF(BAI,MESG)

int main(int argc, char* const* argv)
{
  BAS_LOGINF("Bank agent server start");
   try {
    BAS_LOGINF("Initializing ORB");
    // Initialize the ORB.
```

```
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

       BAS_LOGINF("Resolving Initial reference to Root POA");
        // get a reference to the root POA
        CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
     ...
   }
```

Alternatively, the application could have chosen to use a logger specific for it's logging by changing the macro MYLOG as follows:

```
#define MYLOG(LVL, COMP, MESG) \
    if (VISDLoggerMgr::instance()->global_log_enabled()) { \
       VISDLoggerMgr::instance()->get_logger("mylogger")->log( \
            LVL, COMP, MESG \
            , __FILE__, __LINE__ \
            ); \
    }
```

All the log messages from the source "bankagentserver" and "bankagentimpl" will then be directed to "mylogger" logger and its configured appenders. The log messages coming from the ORB will go to the appenders configured on the default logger.

## Filtering

During trouble shooting exercises sometimes it is very useful to filter out log messages from particular modules or source names or change their verbosity. The logger framework provides a powerful yet simple means to have a sophisticated filtering mechanism using the logger filters. The filter in each logger considers two components of a logged message based on which a message is forwarded to the appenders. Each log message has information regarding its source of logging and its log level. The filter uses this information and following the logic below, recommends forwarding:

```
If source name is registered with the filter
If source name is enabled
If log's log level is greater than or equal to source's log level
Forward
Else
Reject
Else
Reject
Else If the source name "all" is enabled
If the log's log level is greater than or equal to the global log level
(on the log manager) setting
Forward
Else
Reject
Else
Reject
```

"all" is a special source name indicating all the sources that have not been registered with the filter.

The ORB has modularized itself using the following source names:

- connection – logs from the connection related source areas such as server side connection, client side connection, connection pool etc.
- client – logs from client side invocation path
- agent – logs for Osagent communication
- cdr – logs for GIOP areas

- se – logs from the server engine, such as dispatcher, listener etc
- server – logs from server side invocation path
- orb – log outputs from the ORB.

Each of the services have also modularized their components and used appropriate source names.

Some examples given below illustrate how some filtering objectives can be achieved. For more information about the properties, see the configuration section below.

The following properties enable logging and set the global verbosity level to info. Any log messages with lower level are filtered out.

```
vbroker.log.enable=true
vbroker.log.logLevel=info
```

The following properties enable logging and only turns off the log messages from the component that performs osagent communication. All other messages are logged.

```
vbroker.log.enable=true
vbroker.log.default.filter.register=agent
vroker.log.default.filter.agent.enable=false
```

The following set of properties enable logging and allow all log messages but for messages from the component that performs osagent communication with verbosity lower than info.

```
vbroker.log.enable=true
vbroker.log.default.filter.register=agent
vbroker.log.default.filter.agent.enable=true
vbroker.log.default.filter.agent.logLevel=info
```

The following set of properties enable logging and allow only the logs from osagent communication component whose verbosity is either greater or equal to info. All other log messages are filtered out.

```
vbroker.log.enable=true
vbroker.log.default.filter.register=agent
vbroker.log.default.filter.agent.enable=true
vbroker.log.default.filter.agent.logLevel=info
vbroker.log.default.filter.all.enable=false
```

## Reserved names

The following names are reserved and cannot be used for naming loggers, appender and layout types, appender instances or source names – “default”, “appender”, “appenders”, “layout”, filter”, “simple”, “full”, “xml”, “stdout”, “rolling”, “all” and any name starting with “v”. Behavior is indeterminate if such strings are used.

## Customization

If the built in appenders and layouts are not sufficient, then custom objects can be implemented and provided in shared library and the logger framework made to load them at runtime.

To do this, the following steps need to be performed.

1. VISDAppenderFactory and VISDAppender interfaces implemented in a shared library or DLL.
2. A global instance of the implemented factory should register with the logger manager using register_app_factory in its constructor
3. Using the property configuration as described below, the logger framework can be then made to load the library and use this custom factory and its appenders.

Similar steps could be also performed for custom layouts.

For example, if an application wanted to use its own appender that logged to the console, using a custom layout that just printed the log message and omitted all other details, first the appender and layout interface need to be implemented as described below.

The following code snippet shows the classes that implement the appender factory and the appender.

```
class StdOutAppFactory : public VISDAppenderFactory {
public:
  // Constructor
  StdOutAppFactory() {
  // register when the global instance object is created
  VISDLoggerMgr::instance()->register_app_factory(this);
  }
  ...
  // unique appender type name for this custom appender
  virtual const char* type_name() { return "mystdout"; }
  // API that the framework will call when it needs an appender
  // instance of this type
  virtual VISDAppender_ptr create(const char* logger_name,
  VISDConfig::LogAppenderConfig_ptr p);
  // API that the framework will call when it needs to destroy
  // an appender instance created by this factory
  virtual void destroy(VISDAppender_ptr app);
  // global instance object that gets created when the library
  // or DLL gets loaded
  static StdOutAppFactory _instance;
};
class StdOutApp : public VISDAppender {
public:
  ...
  // should return TRUE if the appender is using ORB features else FALSE.
  // Since this appender type does not need any ORB feature
  // it returns FALSE
  virtual CORBA::Boolean ORB_initialized(void* orb_ptr);
  // After shutdown notification, ORB features should not be used
  virtual void ORB_shutdown();
  // actual append implementation. Should return TRUE is append
  // operation is successful
  virtual CORBA::Boolean append(const VISDLogRecord& record);
  ...
};
```

The following code snippet similarly explains an implementation for a custom layout.

```
class SimpleLayoutFactory : public VISDLayoutFactory {
public:
  // Constructor
  SimpleLayoutFactory() {
    // register when the global instance object is created
    VISDLoggerMgr::instance()->register_lyt_factory(this);
  }
  // unique type name for this layout
  virtual const char* type_name() { return "mysimple"; }
  // logger framework will call this API when it desires a layout instance
  // of this type
  virtual VISDLayout_ptr create(const char* logger_name,
  VISDConfig::LogAppenderConfig_ptrp);
  // calls this API when a layout instance created by this factory
  // is to be destroyed
  virtual void destroy(VISDLayout_ptr layout);
  // global instance object that gets created when the library
```

```
    // or DLL gets loaded
    static SimpleLayoutFactory _instance;
    ...
  };
  class SimpleLayout : public VISDLayout {
  public:
    ...
    // API that will be called by an appender using this layout to format
    // the message
    virtual void format(const VISDLogRecord& record,
                        char* buf,
                        CORBA::ULong buf_size,
                        CORBA::String_var& other_buf);
    ...
  };
```

If the above was built into a library say Custom.dll (or libCutom.so), then by using the following properties, the framework could be made to use this:

```
vbroker.log.enable=true

# Define the appender and layout types for the framework to use
vbroker.log.appender.register="mystdout"
vbroker.log.appender.mystdout.sharedLib=Custom.dll (or libCustom.so)
vbroker.log.layout.register="mysimple"
vbroker.log.layout.mysimple.sharedLib=Custom.dll (or libCustom.so)

# Attach an instance of the above custom types on the default logger
vbroker.log.default.appenders=app1
vbroker.log.default.appender.app1.appenderType=mystdout
vbroker.log.default.appender.app1.layoutType=mysimple
```

At runtime, when the default logger is first accessed, the framework will read the configuration information, will identify that the appenders needed for the default logger are in a shared library, will try to load the shared library assuming that the custom objects contained within would have registered themselves with the logger manager using the register_<>_factory APIs and assemble the logger.

## Configuration

All the composition of the logger framework setup is done through configuration and using a runtime property based mechanism; the following aspects can be configured –

1. Global switch on the logger manager indicating whether the logger framework is enabled and global log message verbosity
2. Custom appender and layout factory registration
3. Appender settings on loggers and the individual appender instance configuration on each logger
4. Filter settings on the logger for filtering and to have finer control on the verbosity

### Log manager configuration

```
vbroker.log.enable={true|false}
```

Setting the above property enables or disables the logger manager. The values input are “true” or “false” and by default the value is “false”

```
vbroker.log.logLevel={emerg|alert|crit|err|warning|notice|info|debug}
```

Setting the above property sets the global coarse-grained verbosity setting for the logger framework. This setting however can be refined for further control by configuring the filter on the logger. By default, the value chosen is debug.

Though there are totally 8 log levels, the ORB and all its services use only the following four:

- debug—Lowest level; Specifies fine-grained informational events that are most useful to debug an application for the developers; Similar to an offline debugger, For example, parameter or argument values, contents of a complex data structure like the marshalling buffer, peek on a certain memory contents like the message on the connection wire, etc.
- info—Specifies informational messages that highlight the progress of the application at coarse-grained level; These are general tracing statements. It gives a linear view of how objects are created/destroyed, the flow of various calling and called functions, how certain actions are carried out, and how different components interact together.
- err—Specifies error events that might still allow the application to continue running.; These are scenarios where an error condition was detected, but corrective action could be taken and progress continued.; Log statements in exception handlers can have this log level.
- emerg—Highest level; Designates very severe error events that will presumably lead the application to abort.; These are scenarios where ORB cannot proceed with the functional requirements and no corrective actions can be taken which lead to undefined behavior.

## Appender and layout registration configuration

```
vbroker.log.appender.register=<comma separated list of appender type names>
vbroker.log.appender.<at>.sharedLib=< shared library file >
vbroker.log.layout.register=<comma separated list of layout type names>
vbroker.log.layout.<lt>.sharedLib=<shared library file>
```

Using the above properties, custom appender and layout type names and their implementation location in shared libraries and DLLs can be made known to the logger framework. Here, “at” and “lt” are names of appender and layout types respectively which are among the comma separated type names being introduced.

## Setting appenders and layouts on loggers

```
vbroker.log.<ln>.appenders=<comma separated list of app instance names>
vbroker.log.<ln>.appender.<an>.appenderType=<at>
vbroker.log.<ln>.appender.<an>.layoutType=<lt>
```

To configure appender instances on the loggers, the above set of properties can be used. “ln”, here denotes the logger name. Using the first two properties, the logger framework is instructed on all the appender instance names associated with the logger and their types. If an appender type is not inbuilt, then the shared libraries as explained in the previous set of properties are loaded and appenders obtained. Please note that the logger framework assumes that the shared library on being loaded will automatically register all the appender and layout factories implemented within with the logger manager.

The third property instructs the appender instance of the desired layout. If the appender does not use any layouts, this information is ignored. Otherwise, an instance of the layout type is obtained.

Apart from providing a means to use custom appenders and layouts, the framework also comes inbuilt with some appender and layout types. “stdout” outputs all its messages to console, while “rolling” performs the log append operation on a rolling file based data store. Both these appenders use layouts and can be set with “simple”, “full” or “xml” layouts or any custom layouts. “xml” formats the messages in Log4J xml format.

## Filter configuration

```
vbroker.log.<ln>.filter.register=<Comma separated source names>
vbroker.log.<ln>.filter.<sn>.enable=true/false
vbroker.log.<ln>.filter.<sn>.logLevel={emerg|alert|crit|err|warning|
      notice|info|debug}
```

Each log message being output records the source from where it is emanating. This source name is actually a part of the log record itself. A fine-grained filtering mechanism is provided which allows filtering based on the source names in addition to the global switch provided in the log manager. Using properties, the filter in a logger can be configured to allow log messages based on particular source names and verbosity in context of the source name to be forwarded to the appenders. To configure these attributes of the filter, the above properties can be used. “ln”, here denotes the logger name. First all the source names that we want to control are registered with the filter using the first property. Then setting for each source name is fine tuned using the second and the third properties. “sn” denotes a source name that is registered in the comma separated source names in the first property. A special source name “all” denotes all the source names that have not been configured using the above properties.

## Setting the properties

The above properties can be fed into the logger framework using a properties file containing these properties, pointed to by the environment variable VDLOG_PROP_FILE.

ORB and all its services use the default logger named “default” and hence the ORB overrides the setting of the default logger again by using the VisiBroker for C++ property manager when the above properties are fed in using either “–D” command line parameters or through a properties file pointed to by “–ORBpropStorage” command line parameter.

# 36

# Web Services Overview

A Web Service is an application component that you can describe, publish, locate, and invoke over a network using standardized XML messaging. Defined by new technologies like SOAP, Web Services Description Language (WSDL), and Universal Discovery, Description and Integration (UDDI), this is a new model for creating e-business applications from reusable software modules that are accessed on the World Wide Web and also providing a means for integration of older disparate applications.

## Web Services Architecture

The standard Web Service architecture consists of the three roles that perform the web services publish, find, and bind operations:

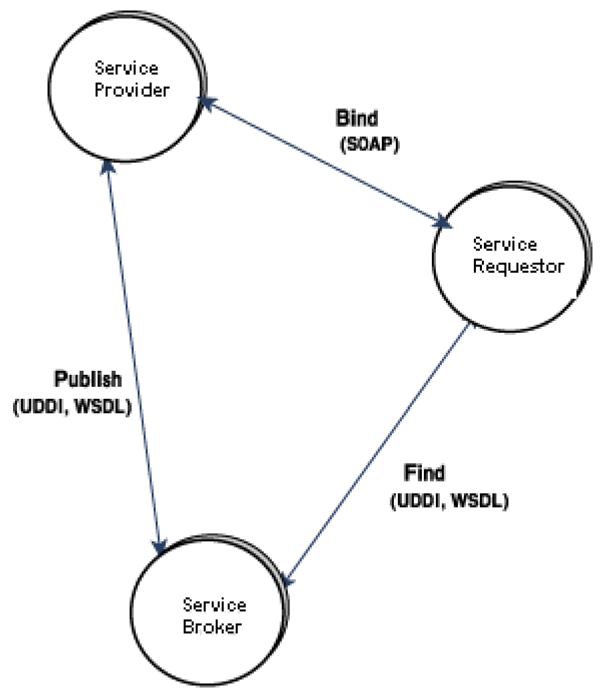
- The *Service Provider* registers all available web services with the Service Broker
- The *Service Broker* publishes the web services for the Service Requestor to access. The information published describes the web service and its location. Apart from publishing the web service, it also co-ordinates in hosting the web service.
- The *Service Requestor* interacts with the Service Broker to find the web services. The Service Requestor can then bind or invoke the web services.

The Service Provider hosts the web service and makes it available to clients via the Web. The Service Provider publishes the web service definition and binding information to the Universal Description, Discovery, and Integration (UDDI) registry. The Web Service Description Language (WSDL) documents contain the information about the web service, including its incoming message and returning response messages.

The Service Requestor is a client program that consumes the web service. The Service Requestor finds web services by using UDDI or through other means, such as email. It then binds or invokes the web service.

The Service Broker manages the interaction between the Service Provider and Service Requestor. The Service Broker makes available all service definitions and binding information. Currently, SOAP (an XML-based, messaging and encoding protocol format for exchange of information in a decentralized, distributed environment) is the standard for communication between the Service Requestor and Service Broker.

## Standard Web Services Architecture



## VisiBroker Web Services Architecture

There are two aspects to the architecture:

– Exposing the CORBA interface for Service Requestors to make invocations using WSDL.

– Providing a runtime environment for enabling CORBA objects to be accessible for the Service Requestors through SOAP/HTTP. This involves the infrastructure to support Services Providers and a Service Broker.

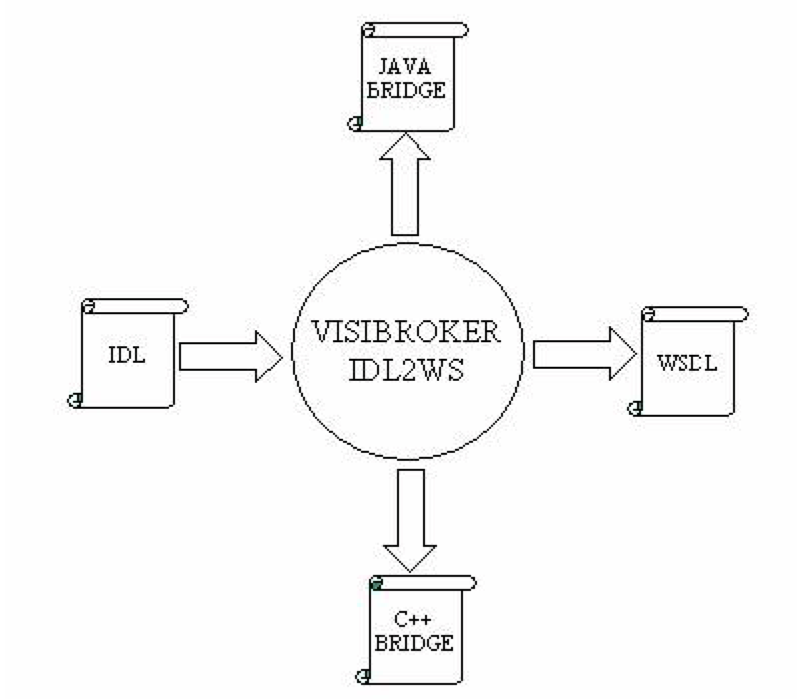
The first aspect is achieved by using a Web Service development tool that converts an IDL interface to a WSDL document using the standard as specified by OMG's CORBA to WSDL/SOAP Inter-working specification. Service Requestors or Web Services clients to make invocations can use the generated WSDL using SOAP over HTTP as transport.

To provide a Web services runtime, VisiBroker uses Apache Axis technology to handle the intricacies of a Services Broker. Using a proprietary type-specific bridge (generated by the tool), deployed stateless CORBA objects can be made accessible. The type-specific bridge instances act as the Service Providers bringing forward the functionality of the CORBA object back end to the Service Requestors.

### Web Services Artifacts

The figure below explains the Web Services development tool provided with VisiBroker that generates the WSDL document and the Bridge code from an IDL file. The WSDL document is useful for the Services Requesters and along with the service description; it also provides the SOAP binding information, which allows any SOAP compliant client to make invocation.

The generated bridge artifact is actually a language/type-specific service provider component that gets deployed in the Service Broker (Axis runtime) and an instance of this is responsible for adapting the incoming SOAP message from the Service Requester to a bound CORBA object.



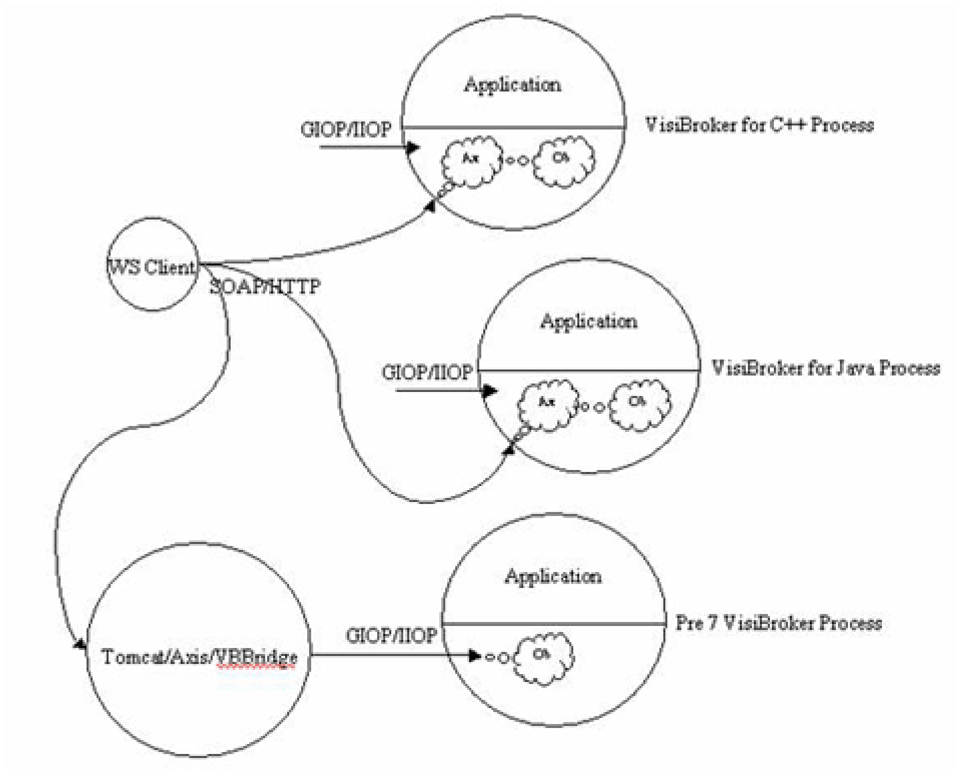
## Web Service Runtime

To explain the runtime behavior, the figure below shows a SOAP client making use of the generated WSDL to make SOAP/HTTP invocations on three CORBA objects exposed as Web Services in VisiBroker for C++, Java and a pre-7.0 VisiBroker process.

VisiBroker process comes with the infrastructure for a HTTP/SOAP listener (internally Apache Axis Technlogy), which is by default turned off. By setting the command line property vbroker.ws.enable=true, this runtime infrastructure can be started. Once the infrastructure is started, the Service providers (bridge) for the CORBA objects can be deployed using Axis's WSDD mechanism. Using few VisiBroker proprietary CORBA object binding related WSDD elements, the deployed bridge instances can be bound to CORBA objects and any SOAP invocations on the bridge is adapted to an in-process CORBA invocation. The bridge in reality is a morph of the Axis's server side generated code, with each web service implementation skeleton mapped to a method on a type specific CORBA object stub. Because the bridge is generated directly from IDL, all the type-safety and fidelity of IDL types is inherently built in. Also, because the bridge is loaded in the same process as the CORBA objects, all invocation on the CORBA object from the bridge is optimized because of VisiBroker's “inprocess” bidder.

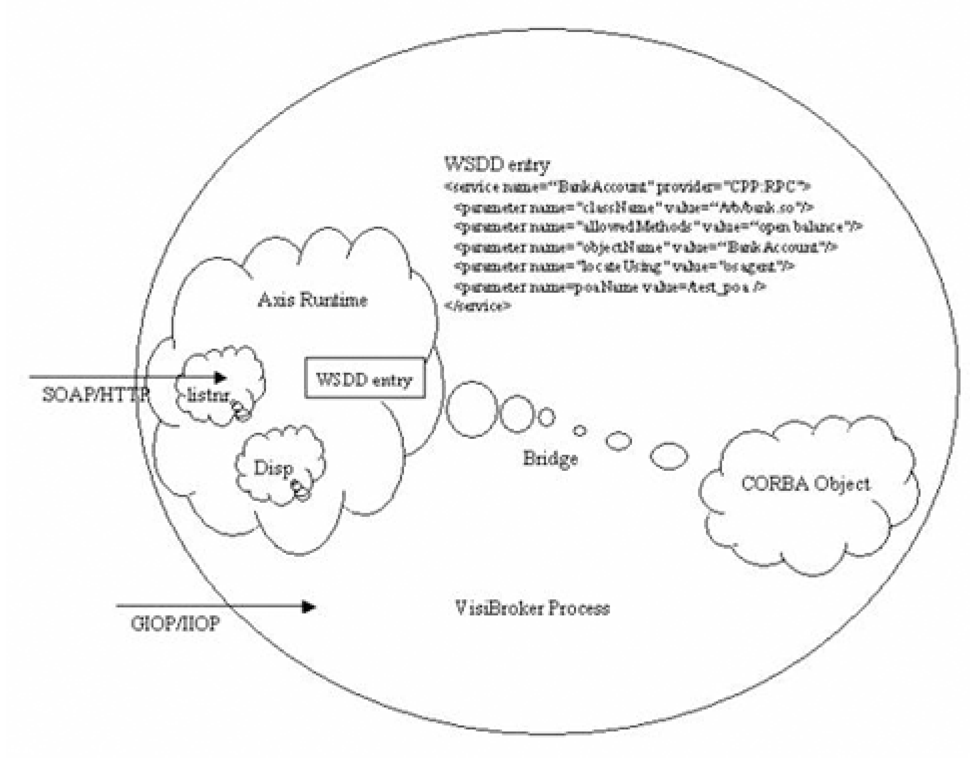
In the figure the cloud "Ax" depicts the Axis + HTTP listener component loaded into the VisiBroker process. "Ob" cloud depicts a CORBA object inside the ORB. The association between the "Ax" and "Ob" cloud as shown by the two small circles between them indicates the deployment of a bridge on the Axis runtime exposing the CORBA object to Service Requesters. Existing CORBA clients can continue making

GIOP over IIOP invocations through the GIOP/IIOP listener as usual without any impact.



To support exposing CORBA objects in Pre 7.0 VisiBroker deployments, the bridge can be deployed on an Axis instance running externally to the VisiBroker process. The only difference in this case is that that SOAP to GIOP adaptation will be remote and hence will be over the wire. In the above figure, this is shown by deploying the bridge on Axis for Java embedded in Apache Tomcat. The cloud "Ob" indicates the CORBA object instance running on a remote Pre 7 VisiBroker Process and the request from the bridge comes in through the GIOP/IIOP end point.

The figure below explains the components inside a VisiBroker process. The "Axis runtime" cloud contains the Axis runtime, the HTTP listener along with the SOAP request dispatcher. A CORBA object inside the process is exposed as a Web Service by deploying its Service provider or the bridge as a Web Service using the Axis WSDD mechanism. When a SOAP client makes an invocation on the Web Service, the HTTP listener picks up the SOAP request and the request is passed to the dispatcher. The dispatcher invokes on the Axis runtime passing in the SOAP request. The Axis runtime decodes the SOAP request and makes invocation on an instance of the deployed Service provider (bridge). The bridge then makes use of the binding information provided in the WSDD to bind to the actual CORBA object and make the CORBA invocation.



```
<service name="BankAccount" provider="CPP:RPC">
  <parameter name="className" value="/lib/bank.so"/>
  <parameter name="allowedMethods" value="open balance"/>
  <parameter name="objectName" value="Bank Account"/>
  <parameter name="locateUsing" value="bsagent"/>
  <parameter name=poaName value=/test_poa />
</service>
```

In the above context, the Service Broker includes only a SOAP node on a HTTP transport. Other services needed for a Web Services deployment such as a UDDI service etc are not provided. Various implementations of these are available and can easily be used.

## Exposing a CORBA object as Web Service

To expose a CORBA object as a Web Service in VisiBroker for C++, the following steps need to be performed.

Development:

1. Generate WSDL document for the IDL interface from IDL file
2. Generate the interface type specific C++ bridge from IDL file
3. Build the bridge into a shared library

Deployment:

1. Enable/Configure Web Service Runtime
2. Deploy the bridge-shared library in the VisiBroker process using Axis WSDD mechanism.

This section illustrates an example provided in the "vbroker/ws/bank" sub directory of examples directory. This example is an adaptation of the "vbroker/basic/bank_agent" example and consists of two interfaces Account and AccountManager. The AccountManager allows for creation of new named accounts. If an account for a particular name already exists, the account is retrieved without creating a new account. Account interface allows for querying of balance in the account. The Server sets up a POA under the root POA and activates an object implementing the AccountManager interface. On making the open operation on this object, separate objects implementing Account interface are created, stored and returned. The code sample shown below illustrates the two interfaces.

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

};
```

In this example, it will be shown how this state-full application can be enhanced to support SOA using Web Services. As a first step in the development, the state-full operations need to be converted to a coarser grained abstraction suitable for SOA. The interface shown below is one such example. This interface as shown, supports a single operation that opens a named account if the account does not already exist and returns the balance in the account.

```
// BankWebService.idl
module BankWebService {
interface AccountManagerWebService {

// opens account if not already opened, and returns balance
float openAndQueryBalance(in string name);

};
};
```

A CORBA object is then implemented which implements this interface, which internally uses the Account and AccountManager interfaces and activated on a known POA with a well known object ID.

Once the server has been enhanced to for stateless operations, web service support can be implemented as illustrated in the following sections.

## Development

### Generating WSDL from IDL

The idl2wsc compiler (idl2wsc.exe on Windows) generates WSDL document for the IDL file according to OMG's CORBA to WSDL/SOAP Inter-working specification. Running the compiler as below for the BankWebService.idl generates a WSDL document named BankWebService.wsdl. This WSDL document can then be published through external means to potential Web Service clients or Client development teams.

```
prompt> idl2wsc BankWebService.idl
```

### Generating the C++ interface type specific bridge

Using the idl2wsc compiler with –gen_cpp_bridge option, the C++ Bridge for a particular interface type can also be generated. The following command will generate the bridge code in file named BankWebService_ws_s.cpp and BankWebService_ws_s.hh. This code is opaque to the applications and should not be changed.

```
prompt> idl2wsc -src_suffix cpp -gen_cpp_bridge BankWebService.idl
```

Please note that the above two commands can be combined.

The generated C++ Bridge needs to be then packaged as a shared library linked in with the stub code of BankWebService.idl to be deployed as a Web Service.

For a complete list of the options available, refer the idl2wsc section of "Programmer tools for C++" chapter.

# Deployment

## Creating Deployment WSDD

The first step to deploy is to edit the Axis WSDD document for the bridge or the service provider. WSDD or Web Service Deployment Descriptor is a standard Axis means to instruct on deployment related information. A template WSDD for the bridge is created during the bridge creation. A sample WSDD is shown below which aims to deploy a Web Service hosted in a CORBA object with object id "BankManagerWebService" in a poa with name "bank_agent_poa". The object reference to this object is bound using osagent.

```
   <deployment xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:CPP="http://xml.apache.org/axis/wsdd/providers/CPP">
<service
name="BankWebService.AccountManagerWebServiceService"
        provider="CPP:RPC"
        description="VisiBroker C++ web service">
        <parameter
            name="className"
            value="[PATH]/libbridge.so"/>
        <parameter
            name="allowedMethods"
            value="openAndQueryBalance"/>
        <parameter
            name="objectName"
          value="BankManagerWebService"/>
       <parameter
            name="locateUsing"
            value="osagent"/>
       <parameter name="poaName" value="/bank_agent_poa"/>
  </service>
</deployment>
```

## Using the created WSDD to deploy

During initialization, Axis C++ reads a configuration file named axiscpp.conf, which is located in $AXISCPP_DEPLOY/etc on Unix  or %AXISCPP_DEPLOY% on Windows, to let the user specify preferences such as parser library to be used and the location of deployment descriptor file.

In the configuration file, WSDDFilePath has to be defined so that Axis knows where to find the WSDD and what services are deployed. XMLParser is only required when the user wants to use a different parser from which is shipped with VisiBroker. VisiBroker WSRT has its own transport implementation, so the settings for transport and the channel are not used.

## A sample axiscpp.conf file

```
# The comment character is '#'
#
# WSDDFilePath:  The path to the WSDD
# XMLParser: Axis XML parser library

WSDDFilePath: /usr/local/VisiBroker/etc/server.wsdd
```

There are two ways to deploy a service:

- Modify the WSDD file to add the service manually.
- Using the tool AdminClient from Axis C++.

**Note**

Because AdminClient depends on the Axis C++ client side libraries, which are not shipped with VisiBroker for C++, to use the tool; one has to get the tool and the required libraries from Apache Axis. (AdminClient is not available in Axis C++ 1.5)

### Web Services Runtime Configuration

Create a property file server.prop to set up the Web Service runtime. Following is a sample property file. The following properties configure the Service Broker to start up a HTTP server on host 143.186.141.54 at port 19000. The connection manager is set up to allow maximum of 30 concurrent connections with 300 seconds to mark the connection idle time. The thread pool to service the incoming SOAP request is setup to have maximum of 300 threads with thread idle time set to 300 seconds. For a complete list of configurable properties, refer the "Web Service Runtime Properties" section of "VisiBroker properties" chapter.

```
vbroker.ws.enable=true
vbroker.ws.listener.host=143.186.141.54
vbroker.ws.listener.port=19000
vbroke.ws.keepAliveConnection=true
vbroker.ws.connectionMax=30
vbroker.ws.connectionMaxIdle=300
vbroker.ws.dispatcher.threadMin=0
vbroker.ws.dispatcher.threadMax=300
vbroker.ws.dispatcher.threadMaxIdle=300
```

Run the Server as follows:

```
prompt> Server -ORBpropStorage server.prop
```

## WSDD Reference

Users may go to `http://www.oio.de/axis-wsdd/` or `http://www.oio.de/axis-wsdd/` for the details of the WSDD.

The parameters used by VisiBroker include:

- className: The name of the shared (dynamic link) library that is loaded by Axis Server Engine when a request arrives on this service. Typically, this is the interface name based on the IDL.
- allowedMethods: The methods that are allowed to be invoked on this class. The CORBA object can have more methods than listed here; the methods listed here are available for Web Service.
- objectName: The name of the object. This is a mandatory parameter.
- locateUsing: This parameter specifies which mechanism the provider uses to locate the object. It has three possible values:

`osagent` —The object is assumed to be in osagent. The `bind()` method is used to locate the object. If `poaName` is also specified, `objectName` is located under that POA. This is the default value of the parameter.

`nameservice` The object is assumed to be in Naming Service. The `resolve()` method on the root context is used to locate the object. The objectName must be the full name of the object starting from root context.

`ior` —The objectName provided is assumed to be an IOR. The string_to_object() method on the ORB is used to obtain the object. The IOR can be in standard form. For example:

```
corbaname::xxx
IOR:xxx
corbaloc::xxx
```

A sample of the service element in WSDD:

```
<service
    name="AServiceBankWebService.AccountManagerWebServiceService"
    provider="CPP:RPC " description="VisiBroker C++ web service">
    <parameter name="className"
        value="/usr/local/VisiBroker/servies/libaccount_manager.so"/>
        <parameter name="allowedMethods" value="openAndQueryBalance"/>
        <parameter name="objectName" value="BankManagerWebService" />
        <parameter name="locateUsing" value="osagent" />
        <parameter name="poaName" value="/bank_agent_poa">
    </parameter>
</service>
```

## Limitations

Because of some Axis limitations, the following restrictions apply in the current version.

- An IDL file can have only a single interface definition. This is because Axis WSDL2WS tool currently does not support multiple port-types in the WSDL.
- Every single bridge needs to be bundled in a separate shared library.

## SOAP/WSDL compatibility

SOAP version 1.1 and WSDL version 1.1 is supported.

# 37

# Reducing ORB runtime footprints

The ORB functionality is separated into various libraries instead of having all functionalities in one monolithic library. This gives VisiBroker application the advantage of using only the specified ORB functionality required.

However, this does not mean that the VisiBroker application developed will be smaller in size. But rather, it helps in linking up the application with the minimum required ORB libraries, that the application would need, for reduced ORB memory footprint usage.

Note that the previous CORBA applications do not need to change their build process and link with the new ORB library names. Existing applications also do not have to modify their build process as they can still link up their application with "`orb_r`" libraries on Windows or "`liborb`" libraries on UNIX or Linux platforms.

These libraries are now just the archives of all the dynamically linked ORB libraries. Most compilers can determine the dynamically linked ORB libraries in "orb_r" or "liborb" that the application depends on and thus determine which should be loaded during the runtime.

## Different ORB Libraries

The ORB is broken into several libraries pertaining to the ORB functionality or usage. The VisiBroker applications can now decide which libraries should be used during the runtime.

These ORB libraries are listed below:

- "Core ORB library" (orbcore library)
- "Smart Agent (osagent) Usage library" (dsuser library)
- "Location Service library"(locsvc library)
- "Interface Repository library" (ir library)
- "Dynamic Any library" (dynany library)
- "Server Manager Usage library" (svrmgr library)
- "Gatekeeper (firewall) library" (fw library)

## Core ORB library

The Core ORB library has all the basic ORB functionality without the components of using the Smart Agent (osagent), Location Service, Interface Repository, Dynamic Any & Server Manager. All VisiBroker applications must at least link with the Core ORB library.

Based on our product examples, an example of link usage for UNIX and Windows platforms is shown below.

**UNIX:**

```
Server: Bank_s.o Bank_c.o Server.o
   $(CC) -o Server Server.o Bank_s.o Bank_c.o \
      $(LIBPATH) $(LIBORBCORE) $(STDCC_LIBS)
```

**Windows :**

```
Server.exe: Bank_s.obj Bank_c.obj Server.obj
   $(LINK_EXE) /out:Server.exe Server.obj \
   Bank_s.obj Bank_c.obj $(LIBORBCORE) $(STDCC_LIBS)
```

## Smart Agent (osagent) Usage library

For those VisiBroker applications that require the usage of VisiBroker's Smart Agent (osagent), the applications will not need to link explicitly with the "`dsuser`" library as the Core ORB library will automatically load the "`dsuser`" library during the ORB initialization.

However, an application that sets the property "`vbroker.agent.enableLocator`" to "false" will not load the "`dsuser`" library during the runtime.

See "Using the Smart Agent" for usage on VisiBroker's Smart Agent.

## Location Service library

For those VisiBroker applications that require the usage of the Location Service, the application must link with the "locsvc" library explicitly as well as with the Core ORB library.

See "Using the Location Service" for the usage on Location Service.

Based on our product examples, an example of link usage for the UNIX and the Windows platforms is shown below.

**UNIX**

```
Server: Bank_s.o Bank_c.o Server.o
   $(CC) -o Server Server.o Bank_s.o Bank_c.o \
      $(LIBPATH) $(LIBORBCORE) $(LIBLOCSVC) $(STDCC_LIBS)
```

**Windows**

```
Server.exe: Bank_s.obj Bank_c.obj Server.obj
   $(LINK_EXE) /out:Server.exe Server.obj \
   Bank_s.obj Bank_c.obj $(LIBORBCORE) $(LIBLOCSVC) $(STDCC_LIBS)
```

## Server Manager Usage library

For those VisiBroker applications that require the usage of the VisiBroker's Server Manager, the applications will not need to link explicitly with the "`srvmgr`" library as the Core ORB library will automatically load the "srvrmgr" library during the ORB initialization, if an application sets the property "`vbroker.orb.enableServerManager`" to "`true`".

See "Using the VisiBroker Server Manager" for usage on VisiBroker's Server Manager.

## Interface Repository library

For those VisiBroker applications that require the usage of an Interface Repository, the application will need to link explicitly with the "ir" library as well as with the Core ORB library.

See "Using Interface Repositories" for usage on Interface Repositories.

Based on our product examples, an example of link usage for UNIX and Windows platforms is shown below.

#### UNIX

```
Server: Bank_s.o Bank_c.o Server.o
   $(CC) -o Server Server.o Bank_s.o Bank_c.o \
      $(LIBPATH) $(LIBORBCORE) $(LIBIR) $(STDCC_LIBS)
```

#### Windows

```
Server.exe: Bank_s.obj Bank_c.obj Server.obj
   $(LINK_EXE) /out:Server.exe Server.obj \
   Bank_s.obj Bank_c.obj $(LIBORBCORE) $(LIBIR) $(STDCC_LIBS)
```

## Dynamic Any library

For VisiBroker applications that require usage of CORBA Dynamic Anys, the application will need to link with the "dynany" library explicitly as well as the Core ORB library.  Please see Chapter 28 for usage on Dynamically Managed Types.  Based on our product examples, an example link usage for UNIX and Windows platforms is shown below.

#### UNIX

```
Server: Bank_s.o Bank_c.o Server.o
   $(CC) -o Server Server.o Bank_s.o Bank_c.o \
      $(LIBPATH) $(LIBORBCORE) $(LIBDYNANY) $(STDCC_LIBS)
```

#### Windows

```
Server.exe: Bank_s.obj Bank_c.obj Server.obj
   $(LINK_EXE) /out:Server.exe Server.obj \
   Bank_s.obj Bank_c.obj $(LIBORBCORE) $(LIBDYNANY) $(STDCC_LIBS)
```

## Gatekeeper (firewall) library

For those VisiBroker applications that require the usage of VisiBroker's Gatekeeper firewall, the applications will not need to link explicitly with the "fw" library as the Core ORB library will automatically load the "fw" library during ORB initialization, if an application sets the property "`vbroker.orb.enableFirewall`" to "`true`".

See the Gatekeeper Guide for usage details of Gatekeeper library.

# Index

# Index

# Index

# Index

# Index

# Index

# Index