Programmer's Guide

VERSION 3.3

VisiBroker™ for C++

Inprise Corporation, 100 Enterprise Way
Scotts Valley, CA 95066-3249

Printed in the U.S.A.

VCA0033WW21001 1E0R798
9899000102-9 8 7 6 5 4 3 2 1
D4

Contents

Chapter 6

Activating objects and implementations 6-1

Tables

Figures

Chapter

1

Preface

The *VisiBroker for C++ Programmer's Guide* provides information on developing distributed object-based applications with the VisiBroker ORB.

This Preface lists the contents of the *VisiBroker for C++ Programmer's Guide*. It also describes typographic and platform conventions used throughout the manual and provides references for more information about VisiBroker for C++ and Common Object Request Broker Architecture (CORBA).

Organization of this manual

This manual includes the following sections:

Part I, "VisiBroker fundamentals"

- Chapter 2, "VisiBroker basics," introduces CORBA concepts, describes VisiBroker features, and describes the software development process using VisiBroker for C++.
- Chapter 3, "Setting up your environment," describes how to set up the VisiBroker environment so that you can run your applications and use VisiBroker commands.
- Chapter 4, "Quick start for development with VisiBroker for C++," uses an example program to illustrate application development with VisiBroker.
- Chapter 5, "Accessing distributed objects with object references," describes how client applications refer to and use distributed objects.
- Chapter 6, "Activating objects and implementations," explains how objects are implemented and made available for use by client applications. The Object Activation Daemon (OAD) is described in this chapter.

Part II, "Advanced VisiBroker development"

- Chapter 7, "Handling exceptions," provides information on handling errors using C++ exceptions.

- Chapter 8, “Managing threads and connections,” discusses how VisiBroker handles multi-threaded clients and object servers. It also explains how VisiBroker manages network connections between clients and servers.
- Chapter 9, “Using the IDL to C++ compiler,” describes how VisiBroker generates C++ code from IDL code.
- Chapter 10, “Parameter passing rules for the IDL to C++ mapping,” describes the conventions for passing parameters on operation requests.
- Chapter 11, “Smart Agent architecture,” describes how VisiBroker’s Smart Agents can be used to locate objects across a network and ensure object availability through Smart Agent cooperation with the Object Activation Daemon.
- Chapter 12, “Using the tie mechanism: An alternative to inheritance,” explains how to create implementations that do not inherit from the `CORBA::Object` class.
- Chapter 13, “Using interface repositories,” describes how to start and use VisiBroker’s Interface Repository.
- Chapter 14, “Using the Dynamic Invocation Interface,” describes how client applications can dynamically build operation requests at runtime.
- Chapter 15, “Dynamically creating object implementations,” describes how object servers can dynamically create object implementations to fulfill client requests.
- Chapter 16, “Instrumenting and modifying the ORB with interceptors,” describes *interceptors*, extensions that the Visibroker ORB invokes at particular points in its processing.
- Chapter 17, “Customizing remote object invocations using smart stubs,” describes how you can modify invocations on remote objects using smart stubs.
- Chapter 18, “Enabling object persistence using the Object Database Activator,” describes how you can use an object database product with VisiBroker to implement persistent objects.
- Chapter 19, “Discovering object instances using the Location Service,” describes how to use the Location Service which enables developer code to discover, or be notified of, the availability of object instances.
- Chapter 20, “Event loop integration,” explains how you can integrate VisiBroker events with other event-driven programming systems, such as XWindows.
- Chapter 21, “Dynamically managed types,” explains how you can use the DynAny classes to create and interpret data types at runtime.
- Chapter 22, “Using object wrappers,” describes how your client or server application can be notified when an operation request is made or received, or to trap an operation request.
- Chapter 23, “Using the ORB management interface,” describes how your client application can manage a server application by retrieving and setting the server’s attributes.

Part III, “Appendixes”

- Appendix A, “Using VisiBroker for C++ 3.3 with other VisiBroker ORBs,” describes how VisiBroker for C++ version 3.3 interoperates with VisiBroker for Java, VisiBroker for C++ 2.0, and other ORBs.
- Appendix B, “CORBA exceptions,” lists the CORBA exceptions that are raised by VisiBroker and explains their meaning.
- Appendix C, “Handling ORB communication events,” describes how to use VisiBroker communication event handlers.

Typographic conventions

This manual uses the following conventions:

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

Platform conventions

This manual uses the following symbols to indicate that information is platform-specific:

All Windows platforms including Windows 3.1, Windows NT, and Windows 95

Windows NT only

Windows 95 only

All UNIX platforms

Where to find additional information

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

- *VisiBroker for C++ Installation and Administration Guide*. This guide contains the instructions for installing VisiBroker for C++ on Windows and UNIX, and information for system administrators who are deploying distributed applications built using VisiBroker.
- *VisiBroker SSL Pack Programmer's Guide*. This guide provides information about the SSL Pack API and about how to use the SSL feature.
- *VisiBroker Naming and Event Services Programmer's Guide*. This guide provides information on using the VisiBroker Naming and Event services. These services add the naming and event service facilities to VisiBroker.
- The Inprise web site also provides a variety of useful information to developers and others who are interested in evaluating our products and ORB technology. From the web site you can obtain a variety of white papers concerning distributed computing and VisiBroker. Also, visit ORB Central on the Inprise web site to view information specific to developers using VisiBroker ORBs. ORB Central includes a section listing Frequently Asked Questions (FAQ) and their answers. Visit the Inprise web site at http://www.inprise.com/visibroker.

For more information about the CORBA specification, refer to *The Common Object Request Broker: Architecture and Specification*—97-02-25. This document is available from the Object Management Group and describes the architectural details of CORBA. You can access the CORBA specification at the OMG web site: http://www.omg.org.

Contacting Inprise Technical Support

Inprise offers a variety of support options to help you get the most from your Inprise products. For information about these options, see the "Services" section of Inprise's web site at http://www.inprise.com, or contact our Sales Department at 1-800-632-2864.

When contacting VisiBroker Technical Support, be prepared to provide complete information about your environment, the version of the VisiBroker product you are using, and a detailed description of the problem.

Part

I

VisiBroker fundamentals

This part of the *VisiBroker for C++ Programmer's Guide* includes these chapters.

Chapter

2

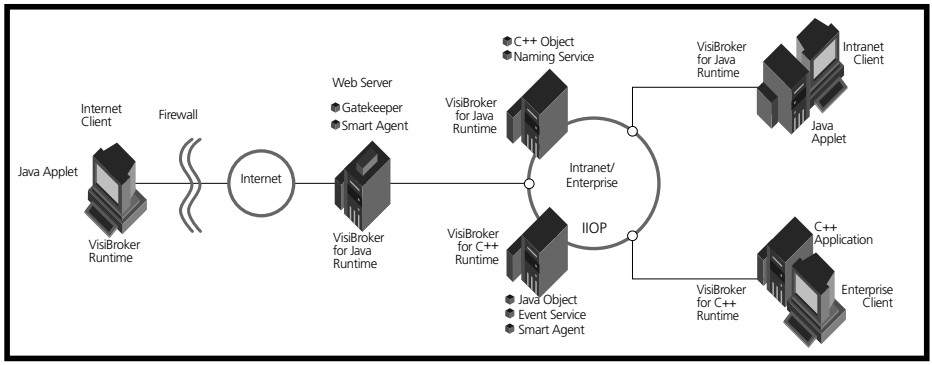
VisiBroker basics

This chapter introduces VisiBroker for C++, a complete implementation of the CORBA 2.0 specification. This chapter describes VisiBroker's features and components. It contains the following sections:

Overview

VisiBroker for C++ provides a complete CORBA 2.0 ORB runtime and supporting development environment for building, deploying, and managing distributed C++ applications that are open, flexible, and interoperable across platforms. Objects built with VisiBroker for C++ are easily accessed by Web-based applications that communicate using OMG's Internet Inter-ORB Protocol (IIOP), the standard for communication between and among distributed objects running on the Internet and intranets. VisiBroker has a native implementation of IIOP, ensuring high-performance, interoperable distributed-object applications for the Internet, intranets, and enterprise computing environments.

Figure 2.1 Architecture of VisiBroker for C++

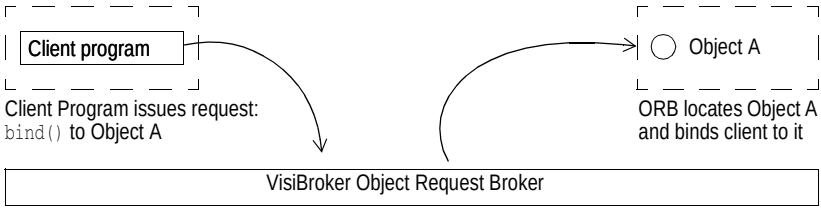


What is CORBA?

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

The Object Request Broker (ORB) in Figure 2.2 connects a client application with the objects it wants to use. The client program does not need to know whether the object 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 2.2 Client program acting on an object through an ORB



Note The ORB 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 objects.

VisiBroker for C++ features

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

VisiBroker smart agent architecture

VisiBroker's Smart Agent (`osagent`)—an extension to the CORBA-specified Basic Object Adaptor (BOA)—is a dynamic, distributed directory service that provides facilities for both client applications and object implementations. Multiple Smart Agents on a network cooperate to provide load balancing and high availability for client access to server objects. The Smart Agent keeps track of objects that are available on a network, and locates objects for client applications at invocation time. VisiBroker 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 restart the server or to connect your client to another server on a different host. For details on the Smart Agent, see Chapter 11, "Smart Agent architecture."

Enhanced object discovery with the location service

For static invocation, 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*, or delegates, the Location Service is useful for developing enhanced load balancing of client requests to server objects. See Chapter 19, "Discovering object instances using the Location Service," for more information.

Implementation and object activation support

VisiBroker's Object Activation Daemon (OAD) can be used to automatically start object implementations when clients need to use them. Additionally, VisiBroker provides an activator class that enables you to defer object activation until a client request is received. You can defer activation for a particular object, or for an entire class of objects on a server. See Chapter 6, "Activating objects and implementations," for details on these features.

Robust thread and connection management

VisiBroker provides native support for single and multithreading 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 the connection ends. With the thread pooling model, threads

are allocated based on the amount of request traffic to the server object. This means that a highly active client will be serviced by multiple threads—ensuring that the requests are quickly executed—while less active clients can share a single thread, and still have their requests immediately serviced.

VisiBroker's connection management minimizes the number of client connections to the server. All client requests 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 Chapter 8, "Managing threads and connections," for details on how VisiBroker manages threads and connections.

IDL compilers

VisiBroker for C++ comes with two IDL compilers that make object development easier,

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

See Chapter 9, "Using the IDL to C++ compiler," and "Updating an interface repository with idl2ir" on page 13-5 for details on these compilers.

Dynamic invocation with DII and DSI

For dynamic invocation, VisiBroker provides implementations of both the Dynamic Invocation Interface (DII) and the Dynamic Skeleton Interface (DSI). The DII allows client applications to dynamically create requests for objects that were not defined at compile time. The DII is covered in Chapter 14, "Using the Dynamic Invocation Interface." The DSI allows servers to dispatch client operation requests to objects that were not defined at compile time. See Chapter 15, "Dynamically creating object implementations," for complete details.

Interface and implementation repositories

The Interface Repository (IR) is an online database of meta information about ORB objects. Meta information stored for objects includes information about modules, interfaces, operations, attributes, and exceptions. Chapter 13, "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 Implementation Repository is an online database of meta information about implementations of ORB objects. The Object Activation Daemon is VisiBroker's

interface to the Implementation Repository that is used to automatically activate the implementation when a client references the object. See Chapter 6, “Activating objects and implementations.”

Optimized binding and transport

When your application binds to an object, VisiBroker selects and establishes the most efficient communication mechanism. Depending on the platform and the location of the requested object, the bind may be established through a pointer reference, shared memory or a TCP/IP socket. Chapter 15, “Dynamically creating object implementations,” describes optimized binding in detail.

Customizing the ORB with interceptors and smart stubs

VisiBroker’s interceptors enable developers to view under-the-cover communications between clients and servers. Interceptors can be used to extend the ORB with customized client and server code that enables load balancing, monitoring, or security to meet specialized needs of distributed applications. See Chapter 16, “Instrumenting and modifying the ORB with interceptors,” for information.

VisiBroker smart stubs allow intervention in client invocations of remote objects. Smart stubs are useful for caching remote object requests. For example, a smart stub could be inserted to significantly improve performance for clients that access static information from server objects. See Chapter 17, “Customizing remote object invocations using smart stubs,” for details.

Persistifying objects with object databases

Integration with object databases is simplified with VisiBroker for C++’s Object Database Activator (ODA). The ODA enables use of an object database product (such as ObjectStore) to implement truly persistent objects. When an object implementation is activated, the object is initialized from the object database. When the object is deactivated, the object’s state is saved to the database. See Chapter 18, “Enabling object persistence using the Object Database Activator,” for more information.

Security for object communication with SSL (optional feature)

VisiBroker uses the industry-standard Secure Socket Layer (SSL) protocol to establish secure connections between clients and servers. VisiBroker’s implementation of the SSL-enabled Basic Object Adaptor (BOA) provides clients and servers with privacy through data encryption, integrity using checksums to detect corruption, and authentication. The SSL BOA offers developers the flexibility to choose the most appropriate authentication method for a particular deployment. For more information, see the *VisiBroker SSL Pack Programmer’s Guide*.

Event loop integration

You may find that the objects you implement require interaction with an event-driven environment. Object implementations are also event-driven because they must wait for client requests. VisiBroker gives you the ability to incorporate your object's event polling into the network or windowing component's event loop. This frees you from the complexity of managing nested event loops in your code. Chapter 20, "Event loop integration," explains the details of event loop integration.

VisiBroker CORBA compliance

VisiBroker for C++ is fully compliant with the CORBA specification (version 2.1) from the Object Management Group (OMG). For more details, refer to the CORBA specification located at http://www.omg.org.

VisiBroker development environment

VisiBroker for C++ is used in both the development and deployment phases. For information about the deployment phase, see "Deploying VisiBroker applications" on page 2-7. The VisiBroker development environment includes the following components:

- Administration and programming tools
- C++ header files
- VisiBroker ORB

To add more functionality to your development and testing environment, you might want to include the VisiBroker Manager.

VisiBroker Manager

VisiBroker Manager is a separate product that provides intuitive graphical tools for visualizing, monitoring, and managing distributed objects from a central location. VisiBroker Manager provides an interactive graphical interface to your network's naming service, interface repositories, implementation repositories, and Smart Agents. For more information, see the *VisiBroker Manager User's Guide* on the Inprise web site at http://www.inprise.com/visibroker.

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.
`idl2cpp`	This tool generates C++ stubs and skeletons from an IDL file.

Administration tools

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

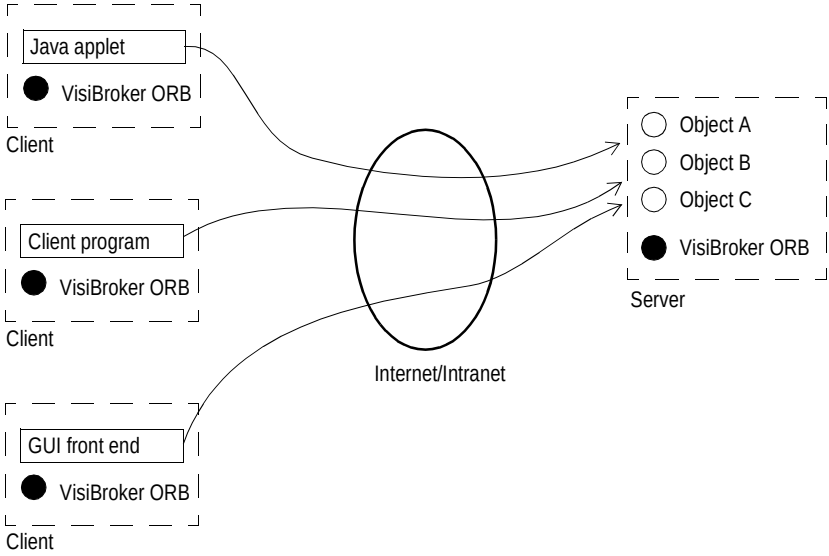
Tool	Purpose
`irep`	Used to manage the Interface Repository. See Chapter 13, "Using interface repositories."
`oad`	Used to manage the Object Activation Daemon (OAD). See Chapter 6, "Activating objects and implementations."
`oadutil list`	Lists ORB object implementations registered with the OAD.
`oadutil reg`	Registers an ORB object implementation with the OAD.
`oadutil unreg`	Unregisters an ORB object implementation with the OAD.
`osagent`	Used to manage the Smart Agent. See Chapter 11, "Smart Agent architecture."
`osfind`	Reports on objects running on a given network.

Deploying VisiBroker applications

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.

The ORB libraries must be installed on every client and server machine. At least one Smart Agent must be running on the network. See the *VisiBroker for C++ Installation and Administration Guide* for details.

Figure 2.3 Client and server programs deployed with VisiBroker C++ and Java ORBs



Note In Figure 2.3, clients can be GUI front ends, applets, client programs. Server implementations contain the business logic on the middle tier

Chapter

3

Setting up your environment

This chapter describes how to set up your VisiBroker runtime environment so that you may begin developing your own client applications and object implementations. It includes the following sections:

Setting the path environment variable

The `PATH` environment variable is set automatically during installation to include the `bin` directory of the VisiBroker distribution.

Note Installation instructions are provided in the *VisiBroker for C++ Installation and Administration Guide*.

If for some reason you need to change the `PATH` environment variable, the following sections explain how to do so.

Updating the PATH on Windows

95 Assuming that the VisiBroker distribution was installed in **c:**, you can set your `PATH` with the following DOS command. Alternately, you may want to set the `PATH` in your **autoexec.bat** file.

```
prompt> set PATH=c:\vbroker\bin;%PATH%
```

NT Although the DOS `set` command can be used to set environment variables in Windows NT, you may find it easier to use the System control panel to automatically set the `PATH`. Assuming that the VisiBroker distribution is installed in **c:\vbroker**, open the System control panel**,** choose "PATH" as the variable to edit, and add the following to the `PATH`:

```
c:\vbroker\bin;
```

Any changes made to your environment variables with System control panel will not be reflected in currently running applications. All subsequently launched applications and DOS prompts will use the new settings.

Updating the PATH on UNIX

U If you are using `csh` and you installed VisiBroker in **/usr/local/vbroker**, you can update the PATH environment using the following command:

```
prompt> set path = ($path /usr/local/vbroker/bin)
```

U If you are using the Bourne shell and you have installed VisiBroker in **/usr/local/vbroker**, you can update the `PATH` environment variable using the following commands:

```
prompt> PATH=$PATH:/usr/local/vbroker/bin
prompt> export PATH
```

Setting the VBROKER_ADM environment variable

The `VBROKER_ADM` environment variable defines the administration directory where important configuration information for VisiBroker's interface repository, Object Activation Daemon, and Smart Agent are stored.

W The `VBROKER_ADM` environment variable is automatically set in the Windows registry when you install VisiBroker. You can change the registry setting by using the `vregedit` tool.

You can override the registry by setting the `VBROKER_ADM` environment variable. Assuming you want your own directory **C:\my\adm** to be used, you could set the VBROKER_ADM environment variable as follows:

```
prompt> set VBROKER_ADM=c:\my\adm
```

U If you are using `csh` and you installed VisiBroker in **/usr/local**, set the `VBROKER_ADM` environment variable as follows:

```
prompt> setenv VBROKER_ADM /usr/local/vbroker/adm
```

U If you are using the Bourne shell and you installed VisiBroker in **/usr/local**, set the `VBROKER_ADM` environment variable as follows:

```
prompt> VBROKER_ADM=/usr/local/vbroker/adm
prompt> export VBROKER_ADM
```

Setting the OSAGENT_PORT environment variable

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

W The OSAGENT_PORT variable is automatically set in the Windows registry when you install VisiBroker. You can change the registry setting by using the vregedit tool.

You can override the registry by setting the OSAGENT_PORT environment variable. Assuming you want the Smart Agent to listen on port number 10000, you could set the OSAGENT_PORT environment variable as follows:

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

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

```
prompt> setenv OSAGENT_PORT 10000
```

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

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

Starting the Smart Agent (osagent) service

The Smart Agent provides the ORB's object location functions and must be started on at least one host in your local network before you attempt to run client applications or servers. The Smart Agent is described in detail in Chapter 11, "Smart Agent architecture." For complete information on how to use this tool, see the *VisiBroker for C++ Installation and Administration Guide*.

95 To start the Smart Agent under Windows 95, select its icon from the VisiBroker Program Group or enter the following command at the DOS prompt:

```
prompt> osagent
```

NT The Smart Agent is installed as an NT service under Windows NT, allowing you to control it with the Service Manager provided with Windows NT. You may also start the Smart Agent in console mode from the DOS prompt entering the following command:

```
prompt> osagent -C
```

U To start the Smart Agent on a UNIX system, enter the following command:

```
prompt> osagent &
```

Starting the Object Activation Daemon

The Object Activation Daemon (OAD) is an optional feature that allows you to register objects that are to be started automatically when clients attempt to access

them. The OAD is described in Chapter 6, "Activating objects and implementations." Before starting the OAD, you should first start the Smart Agent. For complete information on how to use this tool, see the *VisiBroker for C++ Installation and Administration Guide*.

To start the OAD under Windows 95, select its icon from the VisiBroker Program Group or enter the following command at the DOS prompt:

```
prompt> oad
```

NT The OAD is installed as an NT service under Windows NT, allowing you to control it with the Service Manager provided with Windows NT. You may also start the OAD in console mode from the DOS prompt entering the following command:

```
prompt> oad -C
```

U To start the OAD on a UNIX system, enter the following command:

```
prompt> oad &
```

Logging output

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

W Table 3.1 summarizes the names of the various log files that may be produced on Windows.

Table 3.1 Summary of log file names produced on Windows in verbose mode

File name	Description
`oad.log`	Produced by the Object Activation Daemon when started with the `-v` flag.
`osagent.log`	Produced by the Smart Agent when started with the `-v` flag.
`visout.log`	Contains any output to `cout` that is produced by a client or server.
`vislog.log`	Contains any output to `clog` that is produced by a client or server.
`viserr.log`	Contains any output to `cerr` that is produced by a client or server.

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

1 An attempt will be made to write the file to the log directory within the directory pointed to by the `VBROKER_ADM` variable. The following example shows how to set the `VBROKER_ADM` variable on Windows to a specific directory.

Example

```
SET VBROKER_ADM=c:\my\adm\log
```

2 If the application or tool does not have write permission for this directory, an attempt will then be made to write the file to the directory **\vbroker\log** on the drive from which the application or tool was started.

3 If step 2 fails, an attempt will then be made to write the file to the current directory.

Chapter

4

Quick start for development with VisiBroker for C++

This chapter describes the development of distributed, object-based applications with VisiBroker for C++, using a sample application. It includes the following sections:

Overview

In this chapter, you will build a sample client program that can query on the balance in a bank account. From this example, you will learn how to

- Implement a simple interface in IDL for the `Account` object.
- Generate client stubs and server skeletons using the `idl2cpp` compiler.
- Implement the client program.
  - Initialize the ORB.

  - Bind to the `Account` object.
  - Obtain the balance.
  - Handle exceptions.
- Implement the server object.
  - Initialize the ORB and BOA.
  - Instantiate and register an implementation of the server object with the BOA.
  - Prepare to receive requests.
- Build the example.
- Run the example.
  - Start the Smart Agent.
  - Start the `Account` server.
  - Run the client program.

The code for this example is provided in the **account.html** file under the directory `examples/account` where VisiBroker for C++ package was installed. If you do not know the location of the VisiBroker for C++ package, see your system administrator.

The following files are included for the example:

Table 4.1 Files included with the quick start example

File	Description
account.idl	Contains the IDL interface for the `Account` object.
account_srvr.cpp U **account_srvr.C**	`Account` server. Creates an instance of the `Account` object and calls `CORBA::BOA::impl_is_ready()` to make this object available to client programs. The `Account` server implementation implements the `balance()` method which returns the balance in a person's account whose name is provided as input (by generating a random number).
W **account_clnt.cpp** U **account_clnt.C**	This is the `account` client. It binds to an `Account` object and invokes `balance()` on the object reference obtained.
W **account_srvrtie.cpp** U **acount_srvtie.C**	Implements the `account_server` using the tie mechanism instead of using the inheritance approach. This chapter does not discuss this feature—see Chapter 12, "Using the tie mechanism: An alternative to inheritance," for details.
Makefile	Used to build all the test targets.

Prerequisites for running the example

You must configure your runtime environment before you attempt to execute the sample application presented in this chapter. See Chapter 3, "Setting up your environment," for details on configuring your environment.

Development process

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

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

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

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

   Using the `idl2cpp` compiler, we'll produce a client stub (which provides the interface to the `Account` object's `balance()` method) and a server skeleton (which provides an interface to the `balance()` method).

3. Write the client program code.

   To complete the implementation of the client program, we must initialize the ORB, bind to the `Account` object, invoke the `balance()` method on the `Account` object, 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 `Account` class, provide an implementation of the `balance()` method, and implement the server's `main` routine.

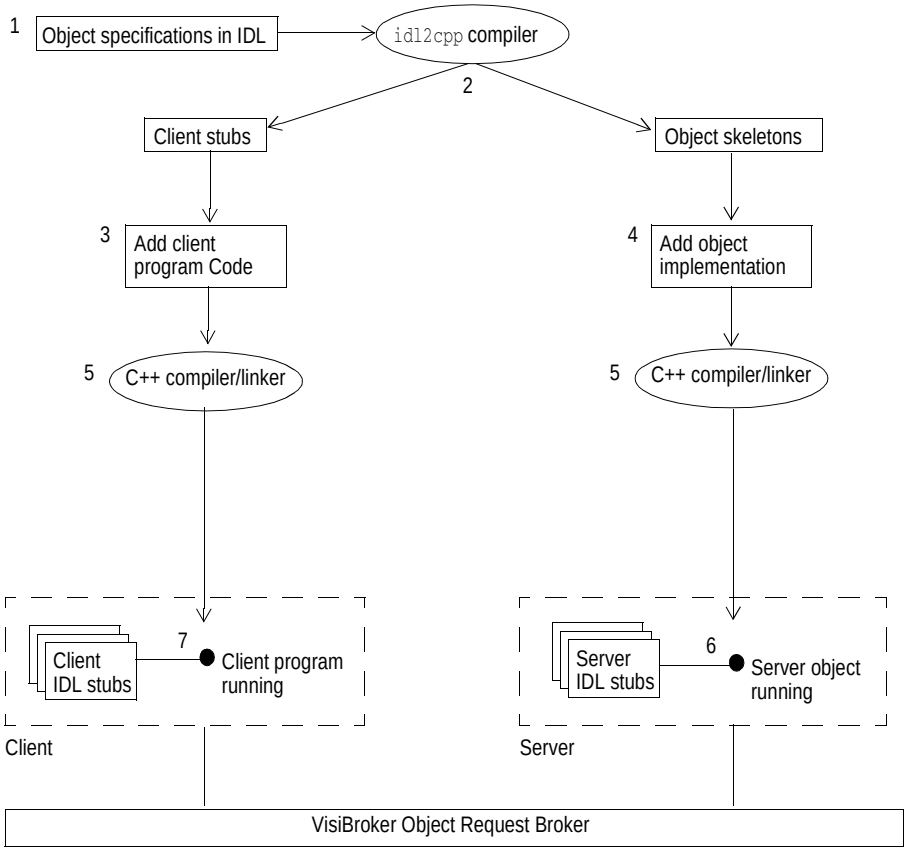
5. Compile the client and server code.

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

6. Start the server.

7. Run the client program.

Figure 4.1 Process for developing the sample bank account application



Writing the account interface in IDL

The first step to creating an application with VisiBroker is to specify all of your objects and their interfaces using the CORBA Interface Definition language (IDL). IDL has a syntax similar to C++, but can be mapped to a variety of programming languages. The IDL mappings for the C++ language are covered in the *VisiBroker for C++ Reference*.

IDL sample 4.1 shows the contents of the `account.idl` file that defines the `Account` interface. The `Account` interface provides a single method for obtaining the current balance.

IDL sample 4.1 account.idl file provides the Account interface definition

```
interface Account {
        float balance();
};
```

Generating client stubs and server skeletons

The interface specification you create in IDL is used by VisiBroker's `idl2cpp` compiler to generate C++ stub routines for the client program, and skeleton code for the object implementation. The stub routines are used by the client program for all method 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 executable client program and server object. These steps are shown in Figure 4.1 on page 4-4.

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

```
prompt> idl2cpp account.idl
```

For more information on the command line options for the `idl2cpp` compiler, see the *VisiBroker for C++ Reference*.

Files produced by the idl2cpp compiler

The `idl2cpp` compiler generates these four files from the **account.idl** file,

- **account_c.hh**—Contains the definitions for the `Account` classes.
- **account_c.cc** (UNIX), **account_c.cpp** (Windows)—Contains internal stub routines used by the client.
- **account_s.hh**—Contains the definitions for the `_sk_Account` skeleton class.
- **account_s.cc** (UNIX), **account_s.cpp** (Windows)—Contains the internal routines used by the server.

You will use the **account_c.hh** and **account_c.cc** (UNIX) or **account_c.cpp** (Windows) files to build the client application. The **account_s.hh** and **account_s.cc** (UNIX) or **account_s.cpp** (Windows) files are for building the server object. All generated files have either a "**cc**" (UNIX only) or "**hh**" suffix to help you distinguish them from source files.

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

W The files that you create yourself traditionally use the "**cpp**" and "**h**" extensions.

U The files that you create yourself traditionally use the "**C**" and "**h**" extensions.

Looking at the client account class

Code sample 4.1 highlights the important aspects of the account class, defined in the **account_c.hh** file. The `Account` class is derived from the `CORBA::Object` class.

Code sample 4.1 Portion of the Account class generated by the idl2cpp compiler

```
...
class Account: public virtual CORBA::Object {
   ...
   public:
   static Account_ptr _duplicate(Account_ptr obj) {
   if (obj) obj->_ref();
      return obj;
   }
   static Account_ptr _nil() { return (account_ptr)NULL;}
   static Account_ptr _narrow(CORBA::Object *obj);
   static Account_ptr _clone(Account_ptr obj) {
      CORBA::Object_var obj_var(__clone(obj));
      return _narrow(obj_var);
   }
   static Account_ptr _bind(const char *object_name = NULL,
               const char *host_name = NULL,
               const CORBA::BindOptions* opt = NULL,
               CORBA::ORB_ptr orb=NULL);
   virtual CORBA::Float balance();
   ...};
```

_bind() method

When your client program invokes the `_bind()` method, the ORB uses the Smart Agent to locate the appropriate server object and return a handle to the `Account` object. If the ORB cannot locate or connect to an `Account` object, the `_bind()` method will raise a system exception. The binding process is covered in detail in Chapter 5, "Accessing distributed objects with object references." Additional aspects of the Smart Agent are covered in Chapter 11, "Smart Agent architecture."

balance() method

The `balance()` method generated by the `idl2cpp` compiler for your client program is actually a stub method. When your client program invokes the `balance()` method, a request is sent to the ORB with all the necessary parameters. The ORB ensures that the request is sent to the `Account` server object. Once the method is executed on the server, the ORB returns the results to your client program.

Other methods

Several other methods are provided that allow your client program to duplicate, initialize, and narrow an `Account` object reference. These methods are discussed in Chapter 5, "Accessing distributed objects with object references."

Looking at the Server _sk_Account class

The **account_s.hh** include file contains the C++ definitions for the `_sk_Account` class from which you derive your implementation of the `AccountImpl` server object. This class contains a skeleton method `_balance()`. This skeleton method is used by the ORB

on the server side to unpack the parameters from your client program's `balance()` request and invoke the actual `balance()` method on the server object.

The `balance()` method is a pure virtual function. You create the actual implementation of this method for the `AccountImpl` server object. The following excerpt shows the `_sk_Account` class definition contained in the **account_s.hh** file.

Code sample 4.2 Portions of the _sk_Account class

```
class _sk_Account : public Account {
   protected:...
   public:...
   // The following operations need to be implemented by the server.
   virtual CORBA::Float balance() = 0;

   // Skeleton Operations implemented automatically
   ...};
```

Implementing the client

The file named **account_clnt.cpp** (Windows) or **account_clnt.C** (UNIX), part of the example, contains the implementation of the client program. Because your program uses the `Account` class, it must include the **account_c.hh** file.

The bank client program performs these steps:

1. Initializes the ORB.
2. Binds to an `Account` object.
3. Invokes the `balance()` method on the `Account` object, using the object reference returned by the `_bind()` method.
4. Prints out the balance.

Initializing the ORB

The first task that your client program needs to do is initialize the ORB object, as shown in Code sample 4.3.

Code sample 4.3 Initializing the ORB

```
#include <iostream.h>
#include "account_c.hh"

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

Binding to the account object

Before your client program can invoke the `balance()` method, it must first use the `_bind()` method to establish a connection to the server that implements the `Account` object. The implementation of the `_bind()` method is generated automatically by the `idl2cpp` compiler. The `_bind()` method requests the 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 `AccountImpl` object. A reference to the proxy object is returned to your client program.

When multiple instances of an object are offered by the same or different servers, an object name argument may be specified to indicate the specific object the client wishes to use.

In this simple example, the object name is omitted so any available object that offers the `Account` interface will be used. Object names are described in detail in Chapter 5, "Accessing distributed objects with object references."

Code sample 4.4 Binding to the Account object

```
...
      // Locate an account.
      Account_var account = Account::_bind();
...
```

Obtaining the balance

Once your client program has established a connection with an `Account` object, the `balance()` method can be used to obtain the balance. The `balance()` method 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.

Code sample 4.5 Invoking the balance() method

```
...
      // Get the balance of the account.
      CORBA::Float acct_balance = account->balance();
      // Print out the balance.
...
```

Handling exceptions

Code sample 4.6 shows the complete **account_clnt.cpp** (Windows) or **account_clnt.C** (UNIX) file. Notice how the `try` and `catch` statements are used to detect any failures, print a message, and exit the client program.

Code sample 4.6 Client program for the quick start example

```
#include <iostream.h>
#include "account_c.hh"
```

```
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) {
            cerr << e << endl;
   }
}
```

Implementing the account server

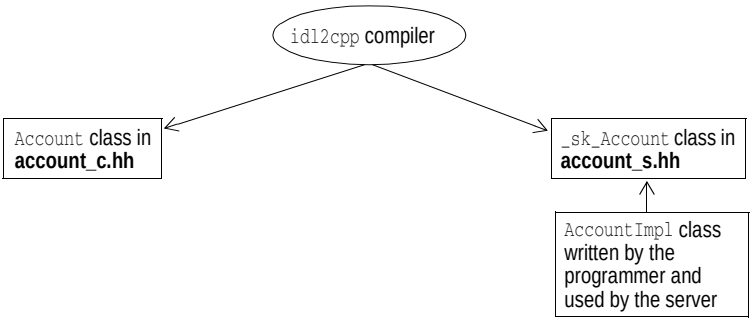
There are a few tasks you must complete to implement the account server.

- Derive the `AccountImpl` class from the `Account` class.
- Provide an implementation for the `balance()` method.
- Implement the server's main routine.

Understanding the Account class hierarchy

The `Account` class that you implement is derived from the `_sk_Account` class that was generated by the `idl2cpp` compiler. Look closely at the `_sk_Account` class definition and notice that it is derived from the `Account` class that is defined in the **account_c.hh** file. Figure 4.2 shows the class hierarchy.

Figure 4.2 Class hierarchy for the AccountImpl interface



Creating the AccountImpl class

The AccountImpl class defines the constructor and the balance() method. The constructor sets the account balance to a random amount between 0 and 10,000 dollars.

Code sample 4.7 AccountImpl class

```
#include "account_s.hh"
#include <math.h>

class AccountImpl: public _sk_Account {
   public:
         AccountImpl(const char *object_name) : _sk_Account(object_name) {
                     // Set a random balance between 0 and 10000 dollars.
               _balance = abs(rand()) % 100000 / 100.0;
         }
         CORBA::Float balance() {
               return _balance;
         }
   private:
               CORBA::Float _balance;
};
```

Looking at the Account server's main routine

The bank server performs these steps in the main routine:

1. Initializes the ORB.
2. Instantiates an AccountImpl object.
3. Registers the AccountImpl object with the ORB.
4. Enters a loop waiting for client requests.

Before instantiating the AccountImpl object, the main routine must make two calls—one to the ORB, and the other to the Basic Object Adaptor (BOA). The BOA is the interface between the object implementation and the ORB. The BOA allows your object to notify the ORB when it is ready to accept client requests and informs it when client requests are received.

The argc and argv parameters passed to the ORB_init() and BOA_init() methods are the same parameters that are passed to the main routine. These parameters can be used to specify options for the ORB and BOA. They are described in the *VisiBroker for C++ Reference*.

Code sample 4.8 Initializing the ORB and BOA in the server

```
...
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB and BOA
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);
   ...
```

After instantiating the `AccountImpl` object, the server tells the BOA that the object is ready by invoking the `obj_is_ready()` method. Lastly, the server calls the `impl_is_ready()` method to start the event loop that receives client requests. The details of the event loop are discussed in Chapter 20, "Event loop integration."

Code sample 4.9 Instantiating an object, registering the object, and waiting for client requests

```
...
   // Create a new account object.
   AccountImpl account("Jack B. Quick");

         // Export the newly created object.
   boa->obj_is_ready(&account);
   cout << account << " is ready." << endl;
         // Wait for incoming requests
   boa->impl_is_ready();
...
```

Code sample 4.10 shows the complete **account_srvr.cpp** (Windows) or **account_srvr.C** (UNIX) file.

Code sample 4.10 Complete Account server implementation

```
#include "account_s.hh"
#include <math.h>
class AccountImpl: public _sk_Account {
   public:
      AccountImpl(const char *object_name) : _sk_Account(object_name) {
         // Set a random balance between 0 and 10000 dollars.
         _balance = abs(rand()) % 100000 / 100.0;
      }
      CORBA::Float balance() {
         return _balance;
      }
   private:
      CORBA::Float _balance;
};
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB and BOA.
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);
      // Create a new account object.
      AccountImpl account("Jack B. Quick");
      // Export the newly created object.
      boa->obj_is_ready(&account);
      cout << account << " is ready." << endl;
      // Wait for incoming requests
      boa->impl_is_ready();}
   catch(const CORBA::Exception& e) {
      cerr << e << endl;
   }
}
```

Building the example

The **account_clnt.cpp** file that you created and the generated **account_c.cc** file are compiled and linked together to create the client program. The **account_srvr.cpp** file that you created, along with the generated **account_s.cc** and the **account_c.cc** 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.

Use the -DSTRICT preprocessor option when linking the implementations to resolve the link errors generated for **orb.lib**. Otherwise, the linker may display an error message suggesting that a constructor is missing from **orb.lib**.

U The **account_clnt.C** file that you created and the generated **account_c.cc** file are compiled and linked together to create the client program. The **account_srvr.C** file that you created, along with the generated **account_s.cc** and the **account_c.cc** 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**.**

The **examples/account** directory of your VisiBroker release contains a **Makefile** for this example. The examples directory also contains a file named **stdmk**, which is included by the Makefile and defines all site-specific settings. You may need to customize the **examples/stdmk** file.

Compiling the example with make or nmake

W Assuming the VisiBroker distribution was installed in **C:\vbroker**, use the following commands to compile the example:

```
prompt> C:
prompt> cd \vbroker\examples\account
prompt> nmake
```

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

U Assuming the VisiBroker distribution was installed in **/usr/local**, issue these commands:

```
prompt> cd /usr/local/vbroker/examples/account
prompt> make
```

In this example, make is the standard UNIX facility. If you do not have it in your PATH, see your system administrator. Code sample 4.11 shows a sample **Makefile** for UNIX.

Code sample 4.11 Sample Makefile for Solaris

```
CC = CC         # set to your C++ compiler
ORBDIR = /usr/local/vbroker # directory where VisiBroker was installed
CCINCLUDES = -I. -I$(ORBDIR)/include
CCFLAGS = $(CCINCLUDES) # compiler flags you might need
            # such as "-g"
ORBLIB = -L$(ORBDIR)/lib -lorb # The VisiBroker library (single threaded)
LDFLAGS = -lsocket -lnsl -ldl # System libraries required by Solaris

ORBCC = $(ORBDIR)/bin/idl2cpp

SRCS = account_c.cc account_s.cc account_clnt.C account_srvr.C

.SUFFIXES: .o .cc .hh

.cc.o:
   $(CC) $(CCFLAGS) -c -o $@ $<

.C.o:
   $(CC) $(CCFLAGS) -c -o $@ $<

#
# Account specific build parameters
#
all: account_client account_server

account_c.cc: account.idl
   $(ORBCC) account.idl

account_s.cc: account.idl
   $(ORBCC) account.idl

account_client: account_c.o account_clnt.o
   $(CC) -o account_client account_clnt.o \
   account_c.o $(ORBLIB) $(LDFLAGS)

account_server: account_s.o account_c.o account_srvr.o
   $(CC) -o account_server account_srvr.o \
   account_s.o account_c.o $(ORBLIB) $(LDFLAGS)

clean:
   rm -f *.o *.hh *.cc core account_client account_server
```

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. See the *VisiBroker for C++ Installation and Administration Guide* for details on starting the Smart Agent.

The Smart Agent is described in detail in Chapter 11, "Smart Agent architecture."

Starting the Account server

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

```
prompt> account_server
```

U Start your `Account` server by typing

```
prompt> account_server &
```

Running the client

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

```
prompt> account_client
```

U To start your client program, type

```
prompt> account_client
```

You should see output similar to that shown below.

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

Chapter

5

Accessing distributed objects with object references

This chapter describes how client programs can obtain references to the objects they wish to use, and helps you to understand how interface and object names are used to identify objects, what options a client can specify when binding to an object implementation, and how object references can be manipulated. It includes the following major sections:

Overview

To invoke operations on an object, your client program must first obtain its object reference. This is usually accomplished by using a bind method and specifying the object's type. The ORB locates the host that offers an object with the requested type. If the server that implements the requested object is not currently executing, the ORB can ensure that the appropriate server is started. After the bind is completed, the client program can use the object reference to invoke operations on the object. The client's method invocations are translated into operation requests that are sent to the object.

An object reference may also be converted to a string, in a process known as *stringification*. This string may then be written to a file or passed to another client. At any time, a *stringified* object reference can be converted to an object reference. Then, a client can use it without binding again.

You can also use the VisiBroker Naming Service to locate objects using logical assigned names. The VisiBroker Naming Service is packaged separately.

Binding to objects

When you define an object interface in an IDL specification, you assign it an interface name. IDL sample 5.1 shows how the `Account` object, introduced in Chapter 4, was given the name "Account" in the IDL specification. The `idl2cpp` compiler generates a C++ class with this interface name and provides a static `_bind()` method that your client programs can use to obtain a reference to the object.

IDL sample 5.1 Assigning an interface name to an object in IDL

```
interface Account {
   float balance();
};
```

Using interface names

The repository ID of the interface implemented by an object is registered with the VisiBroker Smart Agent when a server containing that object invokes the `BOA::obj_is_ready()` method for that object. The interface name is a less specific form of the repository ID by which an object is identified when your client program invokes the `_bind()` method.

For information on obtaining an interface name from an object reference, see "Obtaining object and interface names" on page 5-12.

When to use object names

A server must specify an object name when instantiating an object if the object is to be registered with the Smart Agent and made available to client programs through the `_bind()` method. The `idl2cpp` compiler generates a `NULL` object name as a default parameter for the object's constructor. If two or more servers plan to offer objects with the same interface name, each server can assign a unique object name to their object so that clients can distinguish between the various object instances. You must also assign an object name if you plan to register an object implementation with the Object Activation Daemon, described in Chapter 6, "Activating objects and implementations."

Specifying object names

Consider the account example from Chapter 4 and imagine that you need to have two `Account` objects available—one for Jack B. Quick and one for Jane Doe. You may even want to implement separate servers for each account, possibly executing on different hosts. Each server would instantiate a particular `Account` object, but each would specify a different object name. Code sample 5.1 shows the server changes that you would need to make to create two `Account` objects with different object names.

Code sample 5.1 Specifying an object name for each object's implementation

```
...
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB and BOA.
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

      // Create an account object for Jack B. Quick
      AccountImpl quick("Jack B. Quick");
      // Export the newly created Quick account.
      boa->obj_is_ready(&quick);
      cout << quick << " is ready." << endl;

      // Create an account object for Jane Doe
      AccountImpl doe("Jane Doe");
      // Export the newly created Quick account.
      boa->obj_is_ready(&doe);
      cout << doe << " is ready." << endl;

      // Wait for incoming requests
      boa->impl_is_ready();
   }  ...
}
```

Binding to specific object implementations

Your client program is not required to specify an object name when binding to an object if the same service is available from multiple servers or if there is only one server that implements the object. The `idl2cpp` compiler generates a default, `NULL` parameter for the object name for the `_bind()` method.

Expanding the account example to represent two different accounts will require that you modify the client program's `_bind()` invocation to specify a particular `Account` object. Code sample 5.2 shows the original client code used to bind to a default object that offers the `Account` interface.

Code sample 5.2 Use of the _bind() method without an object name

```
...
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();
      ...
}
```

Code sample 5.3 shows how you would modify the client program to specify an object name with the `_bind()` method.

Code sample 5.3 Modified _bind() method, using an object name

```
...
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("Jack B. Quick");
      ...
}
```

Actions performed during the _bind() process

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

- The ORB contacts the Smart Agent to locate an object implementation that offers the requested interface. If an object name was specified when `_bind()` was invoked, that name will be used to further qualify the directory service search. The Object Activation Daemon (OAD), described in Chapter 6, "Activating objects and implementations," may be involved in this process if the server object has been registered with the OAD.
- When an object implementation is located, the ORB attempts to establish a connection between the object implementation that was located and your client program.
- Once the connection is successfully established, the ORB will create a proxy object and return a reference to that object. The client will invoke methods on the proxy object which will, in turn, interact with the server object. If the object implementation is located in the same process as the client, a proxy object is not created. Instead, a pointer to the object implementation itself is returned.

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.

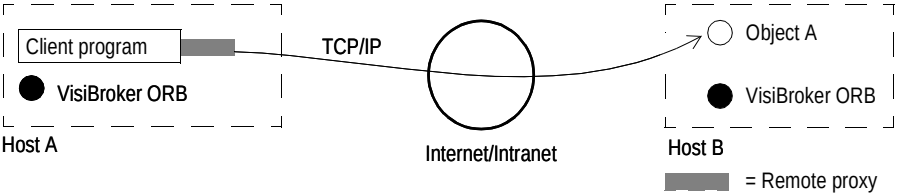
Optimized binding

VisiBroker automatically determines the most efficient way to establish a connection between a client program and an object implementation.

Binding to an object on a remote host

If VisiBroker determines the object implementation you have requested resides on a remote host, a TCP/IP connection will be established between the client and server object. A *proxy object* will be created for your client to use. All methods invoked on the proxy object will be packaged as operation requests and sent to the server on the remote host. The server will unpack the request, invoke the desired method, and send the results back to your client. VisiBroker uses the Internet Inter-ORB Protocol (IIOP) to package the client requests and server responses, which allows interoperability with other ORB products that support IIOP.

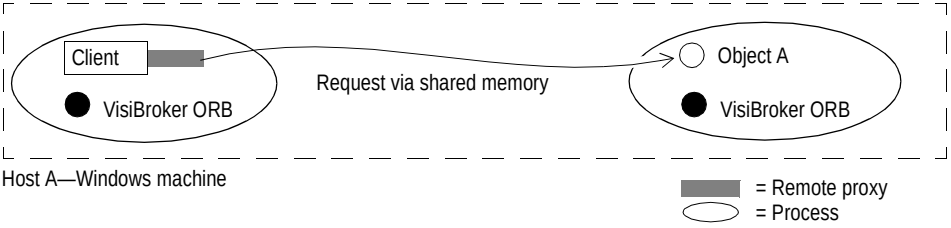
Figure 5.1 Binding when client and server processes are located on different hosts



Binding to an object on the same host

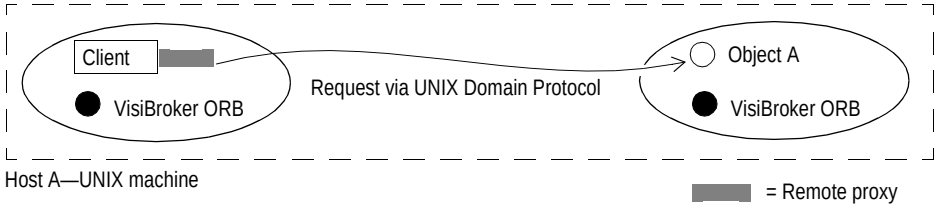
W If the ORB determines that the requested object implementation resides on the local Windows host, a connection will be established between the client and server object using shared memory—only if both the client and server are multithreaded. The ORB will instantiate a proxy object for your client to use. All methods invoked on the proxy object will be packaged as requests and sent to the server using shared memory.

Figure 5.2 Binding when multithreaded client and server processes are located on the same Windows host



U If the ORB determines that the requested object implementation resides on the local UNIX host, a connection will be established between the client and server object using UNIX Domain Protocol—only if both the client and server are multithreaded. The ORB will instantiate a proxy object for your client to use. All methods invoked on the proxy object will be packaged as requests and sent to the server using UNIX Domain Protocol.

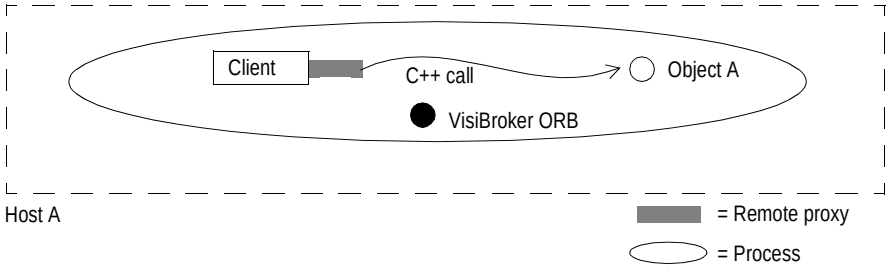
Figure 5.3 Binding when multithreaded client and server processes are located on the same UNIX host



Binding to an object in a single process

The previous discussions have assumed that object implementations have taken the form of a server process. Although this is often the case, a client program and an object implementation can also be packaged within a single process. When your client program invokes a bind in this scenario, the ORB will return a pointer to the object implementation itself. That pointer will be widened to the object type used by your client program. All methods invoked on your client's object will be invoked as C++ virtual functions on the object implementation. The ORB will not be involved after the bind process.

Figure 5.4 Binding when the client and object implementation are located in the same process



Optional bind parameters

This section describes options that you can use to control the behavior of the `_bind()` method. Code sample 5.4 shows the `_bind()` method generated for the `Account` interface by the IDL compiler. The default value for all of the parameters is `NULL`.

Code sample 5.4 _bind method generated for the Account class

```
class Account: public virtual CORBA::Object
{...
   static Account_ptr _bind(const char *object_name = NULL,
                            const char *host_name = NULL,
                            const CORBA::BindOptions* opt = NULL,
                            CORBA::ORB_ptr orb=NULL);
   ...
};
```

Binding to a particular host

In addition to the object name, your client program can specify a particular host on which a desired object implementation is executing. This can be useful if your program knows that a particular object implementation is located on a particular host. The `host_name` parameter is a string containing either the host's fully qualified name or IP address. If you do not specify this parameter, the ORB will locate an implementation that meets all of the other bind parameters.

Controlling the connection to the object with BindOptions

Code sample 5.5 shows the `BindOptions` structure, passed as the third argument to the `_bind()` method, which enables you to control various aspects of the connection between the client program and the object implementation. If the third parameter is `NULL`, the default bind options will be used. Each of the structure's members will be discussed in the following sections.

Code sample 5.5 BindOptions structure

```
struct BindOptions {
                    CORBA::Boolean   defer_bind;
                    CORBA::Boolean   enable_rebind;
                    CORBA::ULong     send_timeout;
                    CORBA::ULong     receive_timeout;
                    CORBA::ULong     connection_timeout;
};
```

Deferring binds

When you set the `defer_bind` member to 1, the `_bind()` method creates a proxy object (if necessary) and returns an object reference to your client program. A connection will not be established with the object implementation until your client program actually invokes a method on the object. If you set `defer_bind` to 0, then the connection will be established when `_bind()` is invoked.

If you do not specify the `BindOptions` parameter, the default behavior will be to establish the connection at the time the `_bind()` method is invoked.

Enabling rebinds

If the connection between your client program and the object implementation fails because of a network error, VisiBroker will automatically attempt to rebind to the server process or a replica of that server. This fault tolerant processing is described in Chapter 11, "Smart Agent architecture." The rebind process is enabled when the `enable_rebind` member is set to 1. If you wish to prevent this rebinding process, set `enable_rebind` to 0.

If you do not specify the `BindOptions` parameter, the default `_bind()` behavior is to attempt to rebind to the server if an error occurs.

Setting a time-out for sending a request

You set the `send_timeout` member to specify the number of seconds your client program will wait for a request to be delivered to a server object. If the time-out period expires before the message is delivered to the server object, a `::CORBA::NO_RESPONSE` exception is raised.

If you do not specify the `BindOptions` parameter, `send_timeout` is set to 0, indicating that your client program wishes to block indefinitely.

Setting a time-out for receiving a response

You set `receive_timeout` to specify the number of seconds your client program will wait for a response to be received from a server object. If the time-out period expires before the message is received from the server object, a `CORBA::NO_RESPONSE` exception is raised.

By default, `receive_timeout` is set to 0, which indicates that your client program wishes to block indefinitely.

Setting a connection time-out

You set the `connection_timeout` option to specify the number of seconds your client program will wait for a connection to be established with a server object. If the time-out period expires before a connection is established, a `CORBA::NO_IMPLEMENT` exception is raised.

By default, `connection_timeout` is set to 0 to indicate that your client program wishes to use your platform's default connection time-out. See your platform's documentation for information on this value.

Specifying a scope for the BindOptions

VisiBroker allows you to specify three distinct levels of `BindOptions`. You can specify the options for each invocation of the `_bind()` method, for a particular object reference, or for all object references used by your client program.

Setting default BindOptions for a process

VisiBroker provides a global `BindOptions` structure that contains default values for the `_bind()` method. These defaults are used if you do not explicitly specify a `BindOptions` parameter when you invoke the `_bind()` method. Code sample 5.6 shows the static methods you can use to query and set these default values.

Code sample 5.6 Static methods for getting and setting the default bind options for a process

```
class Object {
   static const CORBA::BindOptions *_default_bind_options();
   static void _default_bind_options(const CORBA::BindOptions&);
   ...
};
```

Overriding the default BindOptions for a process

You can override the default, process-level bind options by passing a new `BindOptions` parameter when you invoke the `_bind()` method. These new options will remain in effect for the life of the object reference returned by `_bind()`, regardless of any changes to the process-level bind options.

Setting object-level BindOptions

You can change bind options after you have invoked the `_bind()` method. Code sample 5.7 shows a method you can use on the object reference returned by `_bind()`. This method allows you to change the send and receive time-out values for any valid object reference. If you change the connection time-out and a rebind occurs, the new connection time-out value will apply. The bind options you set remain in effect for this object reference for as long as the reference is valid.

Code sample 5.7 Method for setting object-level bind options

```
class Object {
   ...
   void _bind_options(const CORBA::BindOptions& opt);
   ...
};
```

Alternatives to using bind

VisiBroker provides you with an `object_to_string()` method which converts an object reference to a string. The CORBA specification refers to this process as *stringification*. Once an object reference has been stringified, it may be saved to a file, or passed as a command line argument when starting a client program.

VisiBroker provides the `string_to_object()` method for converting a stringified object reference back into an object reference that can be used to invoke methods on the object. This method allows a client program to obtain and use an object reference without having to invoke the `_bind()` method. For more information on these methods, see "Converting a reference to a string" on page 5-12.

VisiBroker also provides a Naming Service, packaged separately, that can be used to locate objects by their logically assigned names.

Performing operations on object references

The object reference returned to your client program by the `_bind()` method represents an ORB object. Your client program can use the object reference to invoke methods on the object that have been defined in the object's IDL interface specification. In addition, there are methods that all ORB objects inherit from the class `CORBA::Object` that you can use to manipulate the object.

Checking for nil references

You can use the CORBA class method is _nil() shown below to determine if an object reference is nil. This method returns 1 if the object reference passed is nil. It returns 0 if the object reference is not nil.

Code sample 5.8 Method for checking for a nil object reference

```
class CORBA {
   ...
   static Boolean is _nil(CORBA::Object_ptr obj);
   ...
};
```

Obtaining a nil reference

You can obtain a nil object reference using the CORBA::Object class _nil() method. It returns a NULL value that is cast to an Object_ptr.

Code sample 5.9 Method for obtaining a nil reference

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

Duplicating an object reference

When your client program invokes this method, 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() method 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() method for each object interface you specify. The _duplicate() method shown accepts and returns a generic Object_ptr.

Code sample 5.10 Method for duplicating an object reference

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

Note The _duplicate() method has no meaning for the BOA or ORB because these objects do not support reference counting.

Releasing an object reference

You should release an object reference when it is no longer needed. One way of releasing an object reference is by invoking the `CORBA::Object` class `_release()` method.

Caution Always use the `release()` method. Never invoke operator delete on an object reference.

Code sample 5.11 Method for releasing an object reference

```
class CORBA {
   class Object {
      ...
      void _release();
      ...
   };
};
```

You may also use the `CORBA` class `release()` method, which is provided for compatibility with the CORBA specification.

Code sample 5.12 CORBA method for releasing an object reference

```
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, VisiBroker automatically deletes the object reference. Code sample 5.13 shows the `_ref_count()` method for retrieving the reference count.

Note When a remote client duplicates or releases an object reference, the server's object reference count is not affected.

Code sample 5.13 Method for obtaining the reference count

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

Cloning object references

The IDL compiler generates a `_clone()` method for each object interface that you specify. Unlike the `_duplicate()` method, `_clone()` will create an exact copy of the object's entire state and establish a new, separate connection to the object

implementation. The object reference returned and the original object reference will represent two distinct connections to the object implementation. Code sample 5.14 shows the `_clone()` method generated for the account interface introduced in Chapter 4, "Quick start for development with VisiBroker for C++."

Code sample 5.14 _clone() method for the Account class

```
class Account: public virtual CORBA::Object
{
   ...
   static Account_ptr _clone(Account_ptr obj) {
   ...
};
```

Cloning an object reference overrides the ORB's default connection behavior. See "Opening a new connection for a thread using _clone()" on page 8-7 for details.

Converting a reference to a string

VisiBroker provides an 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. A client program can convert an object reference to a string and pass it to another client program. The second client may then de-stringify the object reference and use it without having to explicitly bind to the object. Code sample 5.15 shows the conversion methods. Note that the caller of `object_to_string()` is responsible for calling `CORBA::string_free()` on the returned string.

Note Locally scoped object references that are stringified are not guaranteed to be valid beyond the life of the process that created the reference. See "Checking for globally scoped object references" on page 6-4 for more information.

Code sample 5.15 Methods for converting an object reference to a string and vice versa

```
class ORB {
   public:
      // Convert an object reference to a string
      char *object_to_string(CORBA::Object_ptr obj);
      // Convert a char * to an object reference
      CORBA::Object_ptr string_to_object(const char *);
   ...
};
```

Obtaining object and interface names

Code sample 5.16 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 Chapter 13, "Using interface repositories."

Note If you did not specify an object name when you invoked the `_bind()` method, invoking the `_object_name()` method with the resulting object reference will return `NULL`.

Code sample 5.16 Methods for obtaining the interface name, object name, and repository id

```
class Object {...
   const char *_interface_name() const;
   const char *_object_name() const;
   const char *_repository_id() const;
...};
```

Determining the type of an object reference

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 method returns 0 if the object is not of the type specified. Note that this may require remote invocation to determine the type.

Code sample 5.17 Method for determining the type of an object reference

```
class Object {
   ...
   CORBA::Boolean _is_a(const char *repository_id);
   ...
};
```

Code sample 5.18 shows the `_is_equivalent()` method that you can use to check if two object references refer to the same object implementation. This method returns 1 if the object references are equivalent. This method returns 0 if the object references are distinct, but does not necessarily indicate that the object references are to distinct objects. This is a lightweight method and does not involve actual communication with the server object.

Code sample 5.18 Method for comparing object references

```
class Object {
   ...
   CORBA::Boolean _is_equivalent(CORBA::Object_ptr other_object);
   ...
};
```

You can use the `_hash()` method shown in Code sample 5.19 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.

Code sample 5.19 _hash() method

```
class Object {
   ...
   CORBA::ULong _hash(CORBA::ULong maximum);
   ...
};
```

Determining the location and state of bound objects

Given a valid object reference, your client program can use the `_is_bound()` method, shown in Code sample 5.20 to retrieve the current state of the bind for the object. The method returns 1 if the object is bound and 0 if the object is not bound.

Code sample 5.20 Method for querying the state of the bind for an object reference

```
class Object {
   public: ...
      CORBA::Boolean _is_bound() const;
   ...
};
```

Code sample 5.21 shows two methods your client program can use after a successful `_bind()` invocation to determine the location of the object implementation. The `_is_local()` method returns 1 if the client program and the object implementation reside within the same process or address space.

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.

Code sample 5.21 Methods for determining the location of an object implementation

```
class Object {
   ...
   CORBA::Boolean _is_local() const;
   CORBA::Boolean _is_remote() const;
   ...
};
```

Note If the object is in the same process where the method is invoked, `_is_local()` returns 1.

Checking for non-existent objects

You can use the `_non_existent()` method, shown in Code sample 5.22, 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.

Code sample 5.22 Method for determining the existence an object implementation

```
class Object {
   ...
   CORBA::Boolean _non_existent();
   ...
};
```

Obtaining the current BindOptions

Given a valid object reference, your client program can use the `_bind_options()` method in Code sample 5.23 to retrieve the bind options currently in effect for that object.

Code sample 5.23 Method for retrieving an object's bind options

```
class Object {
   ...
   CORBA::BindOptions* _bind_options();
   ...
};
```

See "Specifying a scope for the BindOptions" on page 5-8 for more information.

Widening and narrowing object references

Converting an object reference's type to a super-type is called *widening*. Code sample 5.24 shows an example of widening an `Account` pointer to an `Object` pointer. The pointer `acct` can be cast as an `Object` pointer because the `Account` class inherits from the `Object` class.

Code sample 5.24 Widening an object reference

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

The process of converting an object reference's type from a general super-type to a more specific sub-type is called *narrowing*. VisiBroker maintains a typegraph for each object interface so that narrowing can be accomplished by the object's `_narrow()` method. If the `_narrow()` method determines it is not possible to narrow an object to the type you request, it will return a `nil` reference.

Code sample 5.25 Narrowing an object reference to a sub-type

```
Account               *acct;
Account               *acct2;
Object                *obj;

acct  = Account::_bind();
obj   = (CORBA::Object *)acct;
acct2 = Account::_narrow(obj);
```

Note The `_narrow()` method 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()`.

Chapter

6

Activating objects and implementations

This chapter discusses how objects are implemented and made available to client programs, and enables you to understand how servers and their objects are activated, how the Basic Object Adaptor is used, and how the Object Activation Daemon (OAD) activates objects. It includes the following major sections:

What is the Basic Object Adaptor?

VisiBroker's Basic Object Adaptor, or BOA, provides several important functions to client programs and the object implementations they use, including

- Providing several policies for activating object implementations, and determining how client programs can access these implementations.
- Registering object implementations with VisiBroker's Smart Agent.
- Installing and registering the object with the Implementation Repository, and activating the object upon client request with the Object Activation Daemon.

- Storing information about object implementations residing on a server with the Implementation Repository.

It is important to realize that an object implementation may reside in the same process as a client program or it may reside in a separate process called a server. Servers may contain and offer a single object or multiple objects. Furthermore, servers may be activated by the BOA on demand or they may be started by some entity external to the BOA.

Note For information on BOA options that you can set, see the *VisiBroker for C++ Reference*.

Object server activation policies

VisiBroker implements activation policies that describe the way an object implementation is started and the manner in which it may be accessed by a client program. These activation policies apply only to objects with a global scope, not locally scoped objects.

Shared server policy

When the shared server policy is specified, only one server is launched regardless of the number of clients—the clients share the server. Shared servers are the most common types of servers.

Unshared server policy

Unshared servers are processes that are used at most by one client. A client program causes this type of server to be activated. Once that client exits, the unshared server exits.

Server-per-method policy

This activation policy requires a server process to be started for each method that is invoked. After the method has been completed, the server will exit. Subsequent method invocations on the same object will require a new server process to be started.

Initializing and registering an object

An ORB object is created when its implementation class is instantiated in C++ by an implementation process or server. An object server must then use the BOA to activate the object implementations it offers so that these implementations can receive client operation requests. Object implementations may be activated with either a *local* or a *global* scope.

Globally scoped objects

Objects that have a global scope are registered with the Smart Agent, described in Chapter 11, "Smart Agent architecture," which allows client programs to obtain references to them by invoking the `_bind()` method. References to globally scoped objects are called *persistent object references* because they remain valid beyond the lifetime of the process that obtained the reference. Globally scoped object references can be converted to a string and then saved to a file or passed to other programs. Objects with a global scope are used to implement servers that provide long-term tasks or to implement servers that are to be activated by the OAD.

Note An object with a global scope is registered with the Smart Agent when the `boa::obj_is_ready()` method is invoked. Unless overridden, the default scope of any named object will be global.

Locally scoped objects

Objects that are not given an object name when they are instantiated have a local scope. References to these objects are called transient object references. Only those clients that are explicitly passed a transient object reference may invoke methods on that object. You may want to specify a local scope if you want to implement lightweight objects that will not be generally available to other client programs. A common use for locally scoped objects is for call-back objects.

Code sample 6.1 shows you how you could modify the application introduced in Chapter 4, "Quick start for development with VisiBroker for C++," so that the named `Account` object will not be registered with the Smart Agent.

Note The desired scope must be set before the object is actually created.

Code sample 6.1 Creating a named object with a local scope

```
int main(int argc, char* const* argv) {
      try {
            // Initialize the ORB and BOA.
            CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
                  CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

            // Set local registration scope
            boa->scope(CORBA::BOA::SCOPE_LOCAL);

            // Create a new account object.
            AccountImpl account("Jack B. Quick");

            // Since AccountImpl object is transient, it won't get
            // registered with the Smart Agent
            boa->obj_is_ready(&account);
            cout << account << " is ready." << endl;
            ...
```

Handling locally scoped object references

Clients can only access locally scoped objects when they are passed a reference to the object as an argument or return value. Object references with a local scope are only valid as long as the server that implements the object remains running.

Checking for globally scoped object references

Knowing the scope of an object reference can be useful if you plan to stringify it and save it to a file. Code sample 6.2 shows a method your client program can use to determine whether an object reference has a local or global scope. The `_is_persistent()` method returns 0 if the object has a local scope and a non-zero value if the scope is global.

Code sample 6.2 Method for checking for persistent object implementations

```
class Object {
      ...
      CORBA::Boolean _is_persistent() const;
      ...
};
```

Note Persistent object references may fail-over to a different object implementation unless the `BindOptions` are set to not allow rebind operation to occur. See "Enabling rebinds" on page 5-7.

Registering objects with the BOA

Once a server has instantiated an ORB object that it plans to offer, the BOA must be notified that the object has been initialized. The BOA is notified when the server is ready to receive requests from client programs.

The `obj_is_ready()` method tells the BOA that a particular ORB object is ready to receive requests from client programs. If your server offers more than one ORB object, it must invoke the `obj_is_ready()` method for each object, passing a pointer to the object as an argument.

If the object passed to the `obj_is_ready()` method represents a globally scoped object, the BOA will register the object with VisiBroker's Smart Agent. If the object is locally scoped, no such registration will occur.

Once a server has instantiated and activated all its objects, the server can invoke the `impl_is_ready()` method or enter another event loop. For information on integrating VisiBroker with other event loops, see Chapter 20, "Event loop integration."

Note Invoking the `impl_is_ready()` method will block the caller until the `CORBA::ORB::shutdown()` method is invoked or until the server is terminated.

Activating objects directly

In the account example introduced in Chapter 4, the `AccountImpl` object was activated directly by the server. Direct activation of an object involves instantiating all the C++ implementation classes, invoking the `BOA::obj_is_ready()` method for each object, and then invoking `BOA::impl_is_ready()` to begin receiving requests. Code sample 6.3 shows how this processing would occur for a server offering two `AccountImpl` objects—one with the object name of Jack B. Quick and the other named Jane Doe. Once the objects have been instantiated and activated, the server invokes `BOA::impl_is_ready()` to prevent the `main` routine from returning.

Note The `BOA::obj_is_ready()` must be called for each object offered by the implementation. If a client attempts to invoke the `_bind()` method on an object before the server has invoked `obj_is_ready()`, a `NO_IMPLEMENT` exception will be raised in the client.

Code sample 6.3 Server activating two objects directly

```
...
int main(int argc, char* const* argv) {
      try {
            // Initialize the ORB and BOA.
            CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
            CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

            // Create an account object for Jack B. Quick
            AccountImpl quick("Jack B. Quick");
            // Export the newly created Quick account.
            boa->obj_is_ready(&quick);
            cout << quick << " is ready." << endl;

            // Create an account object for Jane Doe
            AccountImpl doe("Jane Doe");
            // Export the newly created Quick account.
            boa->obj_is_ready(&doe);
            cout << doe << " is ready." << endl;

            // Wait for incoming requests
            boa->impl_is_ready();
      }
      ...
}
```

Automatically activating servers with the Object Activation Daemon

The Object Activation Daemon (OAD) is VisiBroker's 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`) or programmatically with the `BOA::create()` method. There is also an ORB interface to the OAD, described in "IDL interface to the OAD" on page 6-11. In each case, the repository id, object name, the activation policy, and the executable program representing the implementation must be specified.

The OAD is a separate process that only needs to be started on those hosts where object servers are to be activated on demand. See the *VisiBroker for C++ Installation and Administration Guide* for information on starting the OAD.

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**. This file's path name is dependent on the value of the `VBROKER_ADM` variable. If VisiBroker was installed in **/usr/local/vbroker/**, the path to this file would be **/usr/local/vbroker/adm/impl_dir/impl_rep**. These defaults can be overridden using the OAD environment variables, described in the *VisiBroker for C++ Installation and Administration Guide*.

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 required parameters are the interface name, object 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. For more information, see the *VisiBroker for C++ Installation and Administration Guide*.

Note When registering an object implementation, use the same object name that is used when the implementation object is constructed. Only named objects may be registered with the OAD.

Registering objects using BOA::create()

In addition to using the `oadutil` command manually or in a script, VisiBroker allows applications written in C++ to use the `BOA::create()` method to register object implementations with the OAD and the Smart Agent. The OAD stores the information in the implementation repository, allowing the object implementation to be located and activated when a client attempts to bind to the object.

Code sample 6.4 BOA::create() method and its parameters

```
class CORBA { ...
   typedef OctetSequence ReferenceData;
   ...
         class BOA {
         virtual CORBA::Object_ptr create(
               const ReferenceData& ref_data,
               CORBA::InterfaceDef_ptr inf_ptr,
               CORBA::ImplementationDef_ptr impl_ptr) = 0;
         ...};
};
```

Distinguishing between multiple instances of an object

Your implementation can use the `ref_data` parameter 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 ORBs.

Note 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.

Naming objects using the ImplementationDef class

The `impl_ptr` parameter supplies the information that the BOA needs to register an ORB object. The `ImplementationDef` class defines the interface's repository ID, object name, and reference id properties used by the BOA. Code sample 6.5 shows the methods for querying and setting these properties.

Code sample 6.5 ImplementationDef class

```
class CORBA {
   ...
   class ImplementationDef
   {
      public:
      ...
            const char  *repository_id() const;
            void        repository_id(const char *val);
            const char  *object_name() const;
            void        object_name(const char *val);
            CORBA::ReferenceData_ptr id() const;
            void        id(const CORBA::ReferenceData_ptr& data);
            ...
      };
};
```

The `repository_id` attribute represents the name specified in the object's IDL specification. The `object_name` property is the name of this object, provided by the implementer or the person installing the object. The id property is chosen by the implementation and has no meaning to the BOA or the OAD.

Setting activation properties using the CreationImplDef class

The ImplementationDef class does not supply all the information that the OAD needs to activate an object implementation. The CreationImplDef class is derived from ImplementationDef and adds the properties the OAD requires to activate an ORB object—path_name, activation_policy, args, and env. Methods for setting and querying their values are also provided. Code sample 6.6 shows the CreationImplDef class.

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, discussed in "Object server activation policies" on page 6-2. The args and env properties represent command line arguments and environment settings for the server.

Code sample 6.6 CreationImplDef class

```
class CORBA {
   ...
   enum Policy {
      SHARED_SERVER,
      UNSHARED_SERVER,
      SERVER_PER_METHOD
   };
};
class extension {
   ...
   class CreationImplDef: public CORBA::ImplementationDef
   {
      public:
            CreationImplDef(const char *repository_id,
                  const char *object_name,
                  const CORBA::ReferenceData& id,
                  const char *path_name,
                  const CORBA::StringSequence& args,
                  const CORBA::StringSequence& env);
            static CORBA::CreationImplDef_ptr _narrow(
                  CORBA::ImplementationDef_ptr ptr);
            CORBA::Policy activation_policy() const;
            void activation_policy(CORBA::Policy p);
            const char *path_name() const;
            void path_name(const char *val);
            CORBA::StringSequence *args() const;
            void *args(const CORBA::StringSequence& val);
            CORBA::StringSequence *env() const;
            void env(const CORBA::StringSequence& val);
            ...
         };
   };
```

Example of object creation and registration

Code sample 6.7 shows how to use the `CreationImplDef` class and the `BOA::create()` method to create an ORB object and register it with the OAD. These methods would be used in a separate, administrative process, not in the object implementation itself.

Code sample 6.7 Creating an ORB object and registering with the OAD

```
int main(int argc, char* const* argv) {
{
   CORBA::Object_ptr obj;

   // Initialize the ORB and BOA
   CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
   CORBA::BOA_var boa = orb->BOA_init(argc, argv);

   // Optional reference data
   CORBA::ReferenceData id;

   extension::CreationImplDef impl_def("IDL:Account:1.0", "Jack B. Quick", id,
            "/usr/home/dir/acctsrv",
            NULL /* no args */, NULL /* no envs */);
   obj = boa->create(id, NULL, &impl_def);
   if (! CORBA::is_nil(obj))
            cout << "ORB object created successfully" << endl;
            exit (1);
   }
   ...
```

Note If the `impl_def` parameter passed to `BOA::create()` cannot be narrowed to a `CreationImplDef` reference, the create will fail and a `CORBA::BAD_PARAM` exception will be raised.

Dynamically changing an ORB implementation

Code sample 6.8 shows the `BOA::change_implementation()` method which can be used to dynamically change an object's implementation. You can use this method to change the object's activation policy, path name, arguments, and environment variables.

If the `impl` parameter cannot be narrowed to a `CreationImplDef`, this method will fail and a `CORBA::BAD_PARAM` exception will be raised.

Code sample 6.8 change_implementation() method

```
class BOA {
   ...
   virtual void change_implementation(
         const CORBA::Object &obj,
         const CORBA::ImplementationDef& impl);
      ...
};
```

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.

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.

Unregistering objects

When the services offered by an object are no longer available or temporarily suspended, the object should be unregistered with the OAD. When an 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 `BOA::change_implementation()` method to change the object's implementation. As with the registration process, unregistering may be done either at the command line or programmatically. There is also an ORB object interface to the OAD, described in "IDL interface to the OAD" on page 6-11.

Unregistering objects using the oadutil tool

The `oadutil` tool can be used to unregister an object implementation. You can invoke this tool from the command line or from within a script. If a particular interface name is unregistered, all objects instances associated with that interface name will be unregistered. You can specify both the interface and object name if you only wish to unregister a specific object within an interface. For complete information on using this command, see the *VisiBroker for C++ Installation and Administration Guide*.

Unregistering objects using the BOA::dispose() method

An object's implementation can use the `BOA::dispose()` method to unregister an ORB object's information from the OAD's implementation repository. In addition, the `BOA:deactivate_obj()` method is automatically invoked to deactivate the object. Any connections that existed between any client program and the object will be terminated.

Code sample 6.9 BOA::dispose() method

```
class CORBA {
      class BOA {
            ...
            virtual void dispose(CORBA::Object_ptr);
            ...
      };
};
```

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. For complete details on using the `oadutil` command see the *VisiBroker for C++ Installation and Administration Guide*.

IDL interface to the OAD

The OAD is implemented as an 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. IDL sample 6.1 shows the IDL interface specification for the OAD.

IDL sample 6.1 OAD interface specification

```
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);
```

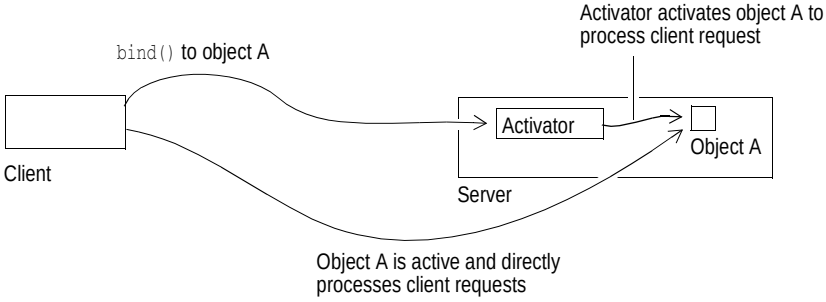
```
        void change_implementation(in extension::CreationImplDef old_info,
                in extension::CreationImplDef new_info)
                raises (NotRegistered,InvalidPath,IsActive);
        attribute boolean destroy_on_unregister;
        void unreg_implementation(in CORBA::RepositoryId 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();
};
```

Deferring object activation until a client request

When you design your object implementation, you may want to defer the activation of one or more ORB objects until a client requests them. By deferring object activation, you benefit from shorter server initialization time and more conservative resource initialization. There are two ways to defer object activation:

- **Deferring activation for a single object.** For information on deferring object activation for a single object, see "Deferring activation for a single object" on page 6-14.

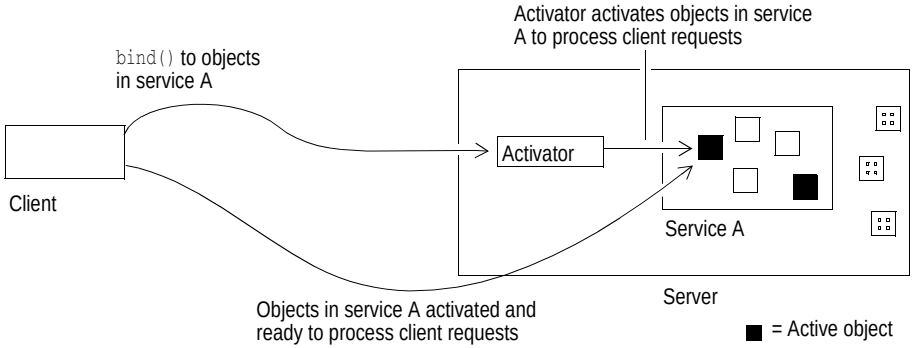
  Figure 6.1 Deferring activation for a single object



- **Deferring activation for a service.** When a server offers a collection of objects that provides clients with a related set of processing, these objects are referred to collectively as a *service* (for example, an object location service). If you have a very large number of objects in a server, you can save system resources by only activating the objects when clients request them. An `Activator` can be configured to

start all objects of a certain service upon request. See "Deferring object activation using service Activators" on page 6-16.

Figure 6.2 Deferring activation for a service



Activator class

Code sample 6.10 shows the `Activator` class, which provides methods invoked by the BOA to activate and deactivate an ORB object.

Code sample 6.10 Activator class

```
class Activator {
   public:
         virtual CORBA::Object_ptr activate(CORBA::ImplementationDef impl)=0;
         virtual void deactivate(Object_ptr, ImplementationDef_ptr impl);
};
```

Deriving your own class from the `Activator` class lets you override the `activate()` and `deactivate()` methods that the ORB will use for the `AccountImpl` object. This lets you delay the instantiation of the `AccountImpl` object until the BOA receives a request for that object. It also lets you provide clean-up processing when the BOA deactivates the object. Code sample 6.11 shows how to create an `Activator` for the `AccountImpl` class.

Code sample 6.11 Deriving an AccountImplActivator class, implementing the activate() and deactivate() methods

```
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.
   return CORBA::Object::_duplicate(new AccountImpl(impl->object_name()));
   }
```

```
   void AccountImplActivator::deactivate(CORBA::Object_ptr obj,
            CORBA::ImplementationDef_ptr impl) {
      // When the BOA deactivates us, release the Account object.
      obj->release();
   }
}
```

Deferring activation for a single object

To defer activation for a single object, the `BOA::obj_is_ready()` and `BOA::impl_is_ready()` methods may be used with the `ActivationImplDef` class to instantiate objects upon receipt of a client request.

Code sample 6.12 BOA::obj_is_ready() and BOA::impl_is_ready() methods

```
class CORBA {
      class BOA {
            ...
            virtual void obj_is_ready(Object_ptr,
                  ImplementationDef_ptr impl=NULL) = 0;
            virtual void impl_is_ready(ImplementationDef_ptr impl=NULL) = 0;
            ...
      };
};
```

In previous examples, the `obj_is_ready()` method was only passed an object reference. The `impl_is_ready()` method was invoked with no parameters at all. It is possible to pass an `ActivationImplDef` pointer to the `obj_is_ready()` method, which can be used to override the activation and deactivation methods used by the BOA.
Code sample 6.13 on page 6–14 shows the `ActivationImplDef` class, which specifies an `Activator` reference and provides methods for setting and retrieving that reference. Note that this class is derived from `ImplementationDef`.

Code sample 6.13 ActivationImplDef class for deferring activation of a single object

```
class extension {
   ...
   class ActivationImplDef: public CORBA::ImplementationDef{
      public:
               ActivationImplDef(const char *repository_id,
                  const char *object_name,
                  const ReferenceData& id,
                  Activator_ptr act);
               Activator_ptr activator_obj();
               void activator_obj(Activator_ptr val);
               ...
   };
};
```

Example of deferred object activation for a single object

Code sample 6.14 shows how to use the `ActivationImplDef` class, along with the `AccountImplActivator` class, to defer the activation of the `AccountImpl` object until a client

request is received. The instantiation of the AccountImpl object no longer appears in the main routine. Instead, the AccountImpl object will be instantiated when the BOA receives a client request and invokes the activate() method.

In this example, the invocation of BOA::obj_is_ready() is passed a NULL object pointer as well as an ActivationImplDef reference. The creation of the AccountImpl object named Jack B. Quick will now be deferred until the first client request for that object is received.

Code sample 6.14 Using the ActivationImplDef class with the BOA::obj_is_ready() method

```
void main(int argc, char * const * argv)
{
   // Initialize ORB and Basic Object Adaptor (BOA)
   CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
   CORBA::BOA_var boa = orb->BOA_init(argc, argv);

   CORBA::ReferenceData id;
   extension::ActivationImplDef impl("IDL:Account:1.0", "Jack B. Quick", id,
               (extension::Activator_ptr) new AccountImplActivator);

   // obj_is_ready is passed an ActivationImplDef object to override
   // the activation of the AccountImpl object.
   boa->obj_is_ready(CORBA::Object::_nil(), &impl);

   // activate other objects
   ...
   // Begin event loop of receiving requests
   boa->impl_is_ready();
   return(1);
};
```

If an implementation has only one object, the separate steps of invoking the obj_is_ready() method followed by the impl_is_ready() method can be combined into a single invocation of the impl_is_ready() method. This will invoke the obj_is_ready() method on the ActivationImplDef object and then enter the blocking event loop.

Code sample 6.15 shows this use of the single invocation of the impl_is_ready() method with an ActivationImplDef object. Since the impl_is_ready() method can accept only one object's implementation, this approach cannot be used if multiple objects reside in the same implementation.

Note If the second parameter to the impl_is_ready() method is set to 1, the method will block. If the second parameter is set to 0, the method will not block.

Code sample 6.15 Use of a single impl_is_ready() method

```
void main(int argc, char * const * argv)
{
   // Initialize ORB and Basic Object Adaptor (BOA)
   CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
   CORBA::BOA_var boa = orb->BOA_init(argc, argv);

   CORBA::ReferenceData id;
   extension::ActivationImplDef impl("IDL:Account:1.0", "Jack B. Quick", id,
               (extension::Activator_ptr) new AccountImplActivator);
```

```
    // impl_is_ready is passed an ActivationImplDef object to override
    // the activation of the AccountImpl object.
    boa->impl_is_ready(&impl, 1);

    return(1);
};
```

Deferring object activation using service Activators

The use of an `Activator` and the `ActivationImplDef` class as described in "Deferring activation for a single object" on page 6-14 allows the deferred activation of a single object implementation until a client requests that implementation. You can also defer the activation of multiple object implementations with a single `Activator`—doing so is called service activation.

Service activation can be used when a server needs to provide implementations for a large number of objects (commonly thousands of objects) but only a small number of implementations need to be active at any specific time. The server can supply a single `Activator` which will be 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 objects implementations whose state is stored in a database. The `Activator` is responsible for loading all objects of a given type or logical distinction. When 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, it writes the object's state to the database and releases the implementation.

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 class from `CORBA_Activator`, overriding the `activate()` and `deactivate()` methods.
4. Register the service name and the Activator class 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 `examples` directory of your VisiBroker installation. The examples directory contains the following files:

Table 6.1 Files in the odb example for service activation

Name	Description
db.idl	IDL for DB and DBObject interfaces.
server.C or **server.cpp**	Creates objects using service activators, returns IORs for the objects, and deactivates the objects.
server_r.C or **server_r.cpp**	Multithreaded server program.
creator.C or **creator.cpp**	Calls the DB interface to create 100 objects and stores the resulting stringified object references in a file (**objref.out**).
client.C or **client.cpp**	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** or **server.exe**
	—**creator** or **creator.exe**
	—**client** or **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 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 **db.idl**) is provided to enable manual creation of these objects.

db.idl interface

The **db.idl** interface enables manual creation of objects that implement the `DBObject` interface.

IDL sample 6.2 db.idl 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 `DBObject`s 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 **db.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 in Code sample 6.16 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`.

Code sample 6.16 Example of implementing a service-activated object

```
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) {}
   ...
};
```

Note that 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 C++ 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

CORBA_Activator and overriding the activate() method, using the overridden activate() method to instantiate C++ classes specific to the object.

In the odb example, the DBActivator class derives from CORBA_Activator, and overrides the activate() and deactivate() methods. The DBObject is constructed in the activate() method.

Code sample 6.17 Example of overriding activate() and deactivate()

```
class DBActivator: public CORBA_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;
```

As shown in Code sample 6.18 on page 6-19, the DB_Activator class creates an operation based on its CORBA::ReferenceData parameter. 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 CORBA::ReferenceData, the requested object's unique identifier.

Code sample 6.18 Example of implementing a service activator

```
CORBA::Object_ptr DBActivator::activate(CORBA::ImplementationDef_ptr impl)
{
   CORBA::ReferenceData_var id(impl->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

As shown in Code sample 6.19 below, the DBActivator service activator is created and registered with the BOA using the BOA::impl_is_ready() call in the main server program. 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 ORB that they belong to the **DBService** service.

Code sample 6.19 Example of instantiating the service activator

```
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 that the call to `BOA::impl_is ready()` is a variation on the usual call to `BOA::impl_is_ready()`—it takes two arguments:

- A service name.
- An instance of an Activator class 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.

Code sample 6.20 First server call to BOA::obj_is_ready

```
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. Refer to Code sample 6.18 on page 6-19 to see this second call in context.

Deactivating objects and implementations

The correct approach to deactivating objects depends on how the instance is activated. Instances can be activated manually through C++ instantiation, by the OAD or by the BOA.

Exiting a server process

A server that is started manually can be considered deactivated when the server exits. A server may use the `CORBA::ORB::shutdown()` method to cause a previously invoked `BOA::impl_is_ready()` method to unblock. Once this method is invoked, the BOA will prevent any new remote operations from being invoked on any objects contained by the server. Once all existing operation requests have returned, the original blocking invocation of the `BOA::impl_is_ready()` method will return and the server's main program can safely exit.

Caution You cannot invoke the `BOA::impl_is_ready()` method after the `CORBA::ORB::shutdown()` method has been invoked.

Deactivating objects registered with the BOA

The `BOA::deactivate_obj()` method is provided to deactivate objects that were activated by the BOA. After this method is invoked, the object will be removed from the Smart Agent's list of objects offered by that implementation. Code sample 6.21 shows the definition of the `deactivate_obj()` method.

Code sample 6.21 BOA::deactivate_obj() method

```
class CORBA {
      class BOA {
            ...
            virtual void deactivate_obj(Object_ptr);
                  ...
      };
};
```

Caution You should not invoke `delete()` on an object since other references to the object may still exist. Instead, invoke `BOA:deactivate_obj()` method and then the `release()` method.

Deactivating an instance of an object

In addition, an object that was created by instantiating its C++ class, will be unregistered when it is destroyed. You can invoke the `CORBA::release()` method to indicate that you no longer wish to hold your reference to the instance. This method will invoke the object's destructor if no other references to the object exist, at which time it will be unregistered from the list of objects maintained by the Smart Agent.

Deactivating implementations started by the OAD

Implementations started by the OAD can be deactivated by using the `BOA::deactivate_impl()` method. Once this method is invoked, the implementation becomes unavailable to service client requests. The implementation can only be reactivated if it is reregistered with the OAD as described in "Automatically activating servers with the Object Activation Daemon" on page 6-5.

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 ORB informs the `Activator` that the object should be reactivated.

Code sample 6.22 Example of deactivating service-activated object implementations

```
// 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 CORBA_Activator {
   virtual CORBA::Object_ptr activate(CORBA::ImplementationDef_ptr impl) {
         CORBA::ReferenceData_var id(impl->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
                _impls[i] = DBObject::_nil();
          }
          _impls.length(0);
       }
};
```

Note that sections of code where multiple threads could be accessing the `_impl` data structure are guarded by using `VISMutex` to provide mutual exclusion.

Part

II

Advanced VisiBroker development

This part of the *VisiBroker for C++ Programmer's Guide* includes these chapters.

Chapter

7

Handling exceptions

This chapter describes how exceptions are reflected and handled in the CORBA model, and enables you to understand how to catch and evaluate system and user exceptions. It includes the following major sections:

Exceptions in the CORBA model

The CORBA specification defines a set of system exceptions that can be raised when errors occur in the processing of a client request. You can define *user* exceptions in IDL for objects you create and specify the circumstances under which those exceptions are to be raised. If an object raises an exception while handling a client request, the ORB is responsible for reflecting this information back to the client.

Looking at the Exception class

VisiBroker uses C++ classes to represent both system and user exceptions. Since both types of exceptions require similar functionality, `SystemException` and `UserException` classes are derived from a common `Exception` class. When an exception is raised, your client program can narrow, or cast down, from the `Exception` class to a specific `UserException` or `SystemException`. Code sample 7.1 shows portions of the `Exception` class definition.

Code sample 7.1 Portions of the Exception class definition

```
class Exception
{
      ...
      public:
               Exception(const Exception &);
               ~Exception();
               Exception &operator=(const Exception &);
               ...

               friend ostream& operator<<(ostream& strm,
                           const Exception& exc);
               const char *_name() const;
               const char *_repository_id() const;
};
```

Methods provided by the Exception class

All exceptions have a name and a repository ID, though the name of the exception is sufficient for error reporting. The repository ID includes the name as well as additional information about the exception. You can invoke the `_name()` and `_repository_id()` methods on an exception to obtain this information.

If you have a client program that requests a bind for an object whose server is currently not running, an exception will be raised. If your application invokes the `_name()` method on the exception object, a string containing "`CORBA::NO_IMPLEMENT`" will be returned. If your program called the `_repository_id()` method, it would return a string containing "`IDL:org.omg/CORBA/NO_IMPLEMENT:1.0`".

System exceptions

System exceptions are usually raised by the ORB, though it is possible for object implementations to raise them through interceptors discussed in Chapter 16, "Instrumenting and modifying the ORB with interceptors." When the ORB raises a `SystemException`, it will be one of the CORBA-defined error conditions shown in Table 7.1 on page 7-3.

Code sample 7.2 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 *_narrow(Exception *exc);
               ...
};
```

Obtaining completion status

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

Code sample 7.3 CompletionStatus values

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

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

Code sample 7.4 Retrieving completion status

```
CompletionStatus completed();
```

Getting and setting the minor code

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

Code sample 7.5 Retrieving and setting minor codes

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

Determining the type of a SystemException

The design of the VisiBroker exception classes allows your program to catch any type of exception and then determine its type by using the `_narrow()` method. A static method, `_narrow()` accepts a pointer to any `Exception` object. As with the `_narrow()` method defined on `CORBA::Object`, if the pointer is of type `SystemException`, `_narrow()` will return the pointer to you. If the pointer is not of type `SystemException`, `_narrow()` will return a `NULL` pointer. See Appendix B, "CORBA exceptions," for details.

Table 7.1 CORBA-defined system exceptions

Exception name	Description
`BAD_CONTEXT`	Error processing context object.
`BAD_INV_ORDER`	Routine invocations out of order.
`BAD_OPERATION`	Invalid operation.
`BAD_PARAM`	An invalid parameter was passed.
`BAD_TYPECODE`	Invalid typecode.
`COMM_FAILURE`	Communication failure.
`DATA_CONVERSION`	Data conversion error.
`FREE_MEM`	Unable to free memory.

Table 7.1 CORBA-defined system exceptions (continued)

Exception name	Description
IMP_LIMIT	Implementation limit violated.
INITIALIZE	ORB initialization failure.
INTERNAL	ORB internal error.
INTF_REPOS	Error accessing interface repository.
INV_FLAG	Invalid flag was specified.
INV_INDENT	Invalid identifier syntax.
INV_OBJREF	Invalid object reference specified.
MARSHAL	Error marshalling parameter or result.
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.
UNKNOWN	Unknown exception.

Handling system exceptions

Your applications should always check for system exceptions after making ORB-related calls. Code sample 7.6 illustrates how the account client program, discussed in Chapter 4, "Quick start for development with VisiBroker for C++," prints an exception using the << operator.

Code sample 7.6 Printing an exception

```
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) {
         cerr << e << endl;
   }
   // Catch other exceptions
   ...
```

Code sample 7.7 Output from client program showing an exception being thrown

```
Exception: CORBA::NO_IMPLEMENT
         Minor: 0
         Completion Status: NO
```

If you were to execute the client program with these modifications without a server present, the output shown in Code sample 7.7 would indicate that the operation did not complete and the reason for the exception.

Narrowing exceptions to a system exception

You can modify the account client program to attempt to narrow any exception that is caught to a `SystemException`. Code sample 7.8 shows how you might modify the client program. Code sample 7.9 shows how the output would appear if a system exception occurred.

Code sample 7.8 Narrowing an exception to a system exception

```
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_var sys_excep;
               sys_excep = CORBA::SystemException::_narrow(&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;
               }
      }
}
```

Code sample 7.9 Output from the system exception

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

Catching specific types of system exceptions

Rather than catching all types of exceptions, you may choose to specifically catch each type of exception that you expect. Code sample 7.10 shows this technique.

Code sample 7.10 Catching specific types of exceptions

```
...
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;
      }
      // 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. Code sample 7.11 shows the `UserException` code from which the `idl2cpp` compiler will derive the user exceptions you specify for your object.

Code sample 7.11 UserException class

```
class UserException: public Exception
{
      public:
            ...
            static const char*_id;
            virtual  ~UserException();
            static UserException *_narrow(Exception *exc);
};
```

Defining user exceptions

Suppose that you want to enhance the account application, introduced in Chapter 4, "Quick start for development with VisiBroker for C++," so that the `account` object will raise an exception. If the `account` object has insufficient funds, you want a user

exception named AccountFrozen to be raised. The additions required to add the user exception to the IDL specification for the Account interface are shown in bold.

Code sample 7.12 Defining user exceptions

```
// Account.idl

interface Account {
      exception AccountFrozen {
      };
      CORBA::Float balance() raises(AccountFrozen);
};
```

The idl2cpp compiler will generate this C++ code for a AccountFrozen exception class.

Code sample 7.13 AccountFrozen class generated by the idl2cpp compiler

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

                  AccountFrozen() {}
                  static CORBA::Exception *_factory() {
                           return new AccountFrozen(); }
                  ~AccountFrozen() {}
                  virtual const CORBA_Exception::Description& _desc() const;
                  static AccountFrozen *_narrow(CORBA::Exception *exc);
                  CORBA::Exception *_deep_copy() const {
                           return new AccountFrozen(*this); }
                  void _throw() const { throw *this; }
      ...
```

Modifying the object to throw the exception

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

Code sample 7.14 Modifying the object implementation to throw an exception

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

Catching user exceptions

When an object implementation raises an exception, the ORB is responsible for reflecting the exception to your client program. Checking for a `UserException` is similar to checking for a `SystemException`. To modify the account client program to catch the `AccountFrozen` exception, make modifications like those shown in Code sample 7.15.

Code sample 7.15 Catching a UserException

```
...
      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. Code sample 7.16 shows how to modify the IDL interface specification to add a reason code to the `AccountFrozen` user exception.The object implementation that raises the exception is responsible for setting the reason code. The reason code is printed automatically when the exception is put on the output stream.

Code sample 7.16 Adding a reason code to the AccountFrozen exception

```
// Account.idl

interface Account {
      exception AccountFrozen {
            int reason;
      };
      CORBA::Float balance() raises(AccountFrozen);
};
```

Chapter

8

Managing threads and connections

This chapter discusses the use of multiple threads in client programs and object implementations, and will help you understand the thread and connection model that VisiBroker uses. It includes the following major sections:

Why is multithreading useful?

A *thread* is a lightweight process that executes a sequence of instructions and reduces overhead by sharing fundamental parts of a program with other threads. Threads are lightweight so that there can be hundreds of them present within a process.

Designing programs with multiple independent threads provides concurrence within an application and improves performance. Multithreading enables applications to 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 an application 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.

Multithreading is 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, or 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.

What thread management does VisiBroker provide?

VisiBroker provides native support for single-threading and multithreading. For platforms that support threads, VisiBroker provides two sets of libraries—a single-threaded library (**orb** or **liborb**) and a multithreaded library (**orb_r** or **liborb_r**). The multithreaded library is thread-safe and *reentrant*, allowing servers to service multiple requests concurrently. If you use the multithreaded library, the VisiBroker ORB will automatically use threads for its internal processing, resulting in more efficient request management.

Note For applications that never intend to use threads, the single-threaded library offers a set of interfaces similar to the multithreaded library. This allows you to use the single-threaded library initially and then recompile and relink with the multithreaded library in the future.

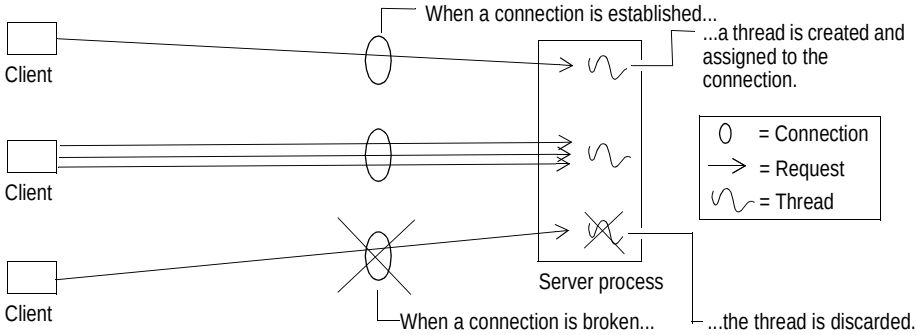
VisiBroker provides two thread models for multithreaded servers:

- Thread-per-session
- Thread pooling

Thread-per-session

When your server selects the thread-per-session model, a new worker thread will be allocated each time a new client connects to the server. A worker thread will be assigned to handle all the requests received from this client's connection. When the client disconnects from the server, the worker thread is destroyed.

Figure 8.1 Thread assignment in the thread-per-session model

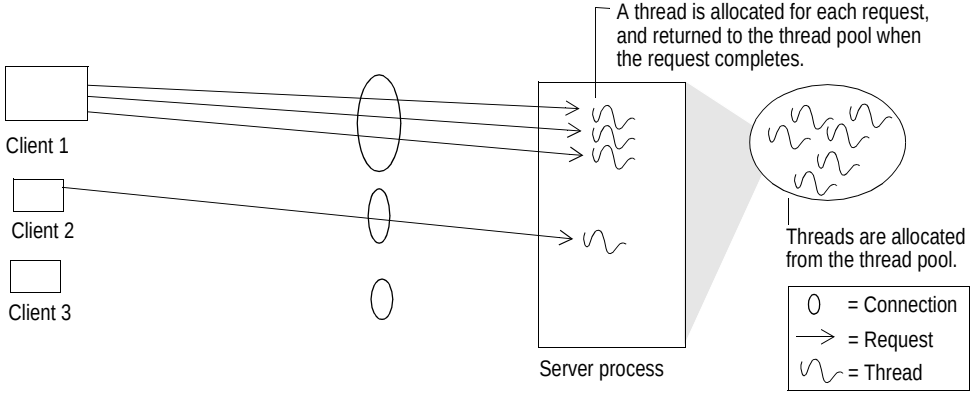


This policy is more efficient than the single-threaded policy because it allows for more parallelism within the server.

Thread pooling

With thread pooling, 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 client requests.

Figure 8.2 Thread assignment in the thread pooling model



Using this model, threads are allocated based on the amount of request traffic to the server object. This means that a highly active client will be serviced by multiple threads—ensuring that the requests are quickly executed—while less active clients can share a single thread, and still have their requests immediately serviced. 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 connections.

VisiBroker conserves system resources by dynamically allocating the number of threads in the thread pool based on the number of concurrent client requests. If the

client becomes very active, threads are allocated to meet its needs. If the 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. 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.

What connection management does VisiBroker provide?

VisiBroker automatically selects the most efficient way to manage connections between client programs and servers, based on the thread policies selected by each.

Overall, VisiBroker's connection management ensures high bandwidth by minimizing the number of client connections to the server. All client requests 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 program is bound to two objects in the server process. Each of its `bind()` requests share a common connection to the server process, even though each `bind()` request is for a different object in the server process.

Figure 8.3 Connections for a client binding to two objects in the same server process

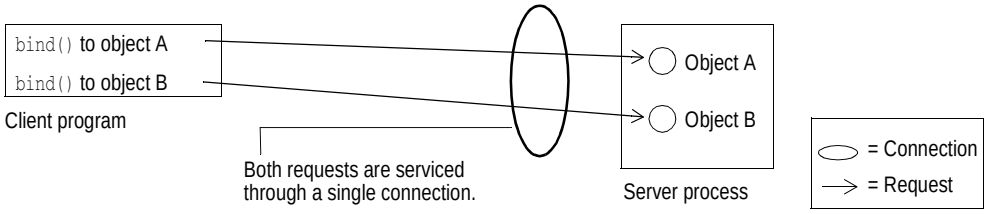
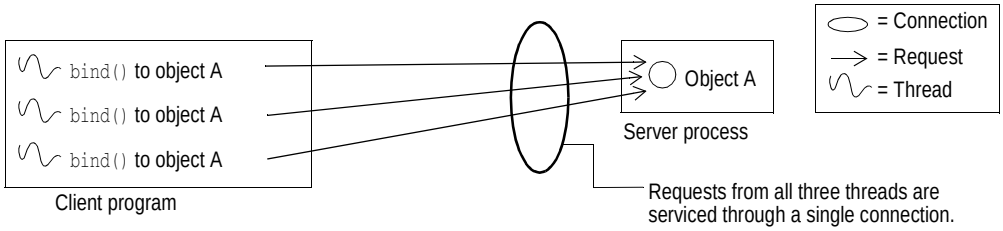


Figure 8.4 shows the connections for a multithreaded client that has several threads bound to an object on the server.

Figure 8.4 Connections for threads in a client binding to an object in a server process



As Figure 8.4 shows, all binds from all threads are serviced by the same connection. For the scenario shown in Figure 8.4, the most efficient multithreading model to use is the thread pooling model (which is the default). If the thread-per-session model is used with this scenario, only one thread on the server will be allocated to service all requests from all threads in the client program—which could 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. This is especially useful for Web-based applications.

Note In VisiBroker 2.0, binds in different threads opened new connections. If you want to perform this behavior in VisiBroker 3.3, you must use `clone()`. See "Opening a new connection for a thread using _clone()" on page 8-7 for more information.

Selecting a threading model for object servers

Object servers can choose between three thread models: *single threaded*, *thread-per-session*, or *thread pooling*. You choose between single threading and multithreading by linking with a particular library. You choose which multithreading model you will use by selecting a particular BOA. The following sections cover these subjects in detail.

Linking VisiBroker libraries

If you want to use multithreading, you must select the reentrant versions of the VisiBroker libraries—regardless of platform, the reentrant libraries have an "`_r`" suffix.

U For UNIX systems, this library should be used for multithreading:

Table 8.1 Multithreading library for Solaris systems

File name	Description
liborb_r.so	Reentrant version of the library **liborb.so**

W For Windows and Windows NT, these libraries should be used for multithreading.

Table 8.2 Multithreading libraries for Windows and Windows NT systems

File name	Description
ORB_R.LIB	Reentrant version of the library **ORB.LIB**
ORB_R_DLL	Reentrant version of **ORB.DLL**

If you want to use single threading, use the versions of these libraries that do not have an "_r" suffix. For UNIX use **liborb.so**, and for Windows use **ORB.LIB** and **ORB.DLL**.

Coding considerations when using the multithreaded library

To use VisiBroker's multithreaded library, all code within a server that implements an 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 mutex or critical section.

Selecting a multithreading policy

When you select the multithreaded library, you can choose between the thread-per-session and thread pooling models. You select a multithreading model by choosing a BOA.

- **Thread-per-session**. The `TSession` BOA is used to implement the thread-per-session model as described in "Thread-per-session" on page 8-2. `TSession` was the default multithreading BOA for the 2.0 release of VisiBroker, and is the default when using '`-ORBbackCompat`'.
- **Thread pooling**. The `TPool` BOA is used to implement the thread pooling model as described in "Thread pooling" on page 8-3. `TPool` is the default multithreading BOA for the 3.3 release of VisiBroker when not using backwards compatibility.

You can specify one of these thread policies by passing a BOA option to the server when it is started, as shown in Code Sample 8.2, or by hard-coding the desired thread policy in the `BOA_init()` method invocation, as shown in Code Sample 8.2. The BOA options are summarized in Appendix A, "Using VisiBroker for C++ 3.3 with other VisiBroker ORBs."

If the thread model is specified with both the BOA argument and in the `BOA_init()` method, the `BOA_init()` parameter takes precedence over the BOA argument.

Code Sample 8.1 Using the BOA argument to set a thread pooling model

```
prompt> server -OAid TPool
```

Code Sample 8.2 Specifying the thread pooling model with the BOA_init() method

```
void main(int argc, char *const *argv)
{
   ...
   CORBA::BOA_ptr boa = orb->BOA_init(argc, argv, "TPool");
   ...
```

Note The single threaded BOA (`TSingle`) is automatically selected if you link with the single threaded library.

Using threads from client programs

Client programs can use threads in several ways in relation to an object implementation.

- The client's main thread can obtain an object reference by invoking the `bind()` method and pass the reference to any of the threads it creates.
- Each client thread can use the `_clone()` method on an object reference passed by the main thread. See the following section, "Opening a new connection for a thread using _clone()," for more information.
- Each client thread can issue its own `bind()` request.

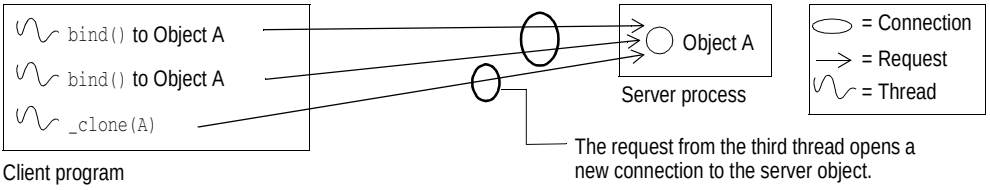
Opening a new connection for a thread using _clone()

If your client program is multithreaded and you need greater control over threads and connections, you can use the `_clone()` method. One reason you might want to use the `_clone()` method is for backward compatibility with VisiBroker 2.0. In VisiBroker 2.0, binds in different threads open new connections. However, in VisiBroker 3.3, all binds from all threads are serviced by the same connection—unless `_clone()` is used.

Furthermore, VisiBroker 2.0 only offered the thread-per-session model which allocates a thread for each connection. If you are not planning to use the thread pooling model in the server (the VisiBroker 3.3 default) then you may wish to use the `_clone()` method in your client program to improve performance. Without cloning, only one thread will be allocated on the server per client connection—even if a client is extremely active and has numerous threads making requests through its connection to the server.

Figure 8.5 shows how the `_clone()` method works.

Figure 8.5 Using the _clone() method to open a new connection for a thread



As shown in Figure 8.5, the `_clone()` method takes an object reference, and binds to the object from the current thread using a new connection to the server process.

Manipulating thread and connection management

VisiBroker provides a number of parameters that can be set to manage threads. On the server side, the thread pool can be manipulated by

- Configuring the maximum size of the thread pool with the `thread_max()` method. This pertains to both the `TSession` and `TPool` BOAs, although it has more application in thread pooling.
- Overriding the operating system default for thread stack size with the `thread_stack_size()` method. This is valuable if the request involves extremely large amounts of data. This applies to all multithreaded BOAs.

VisiBroker provides parameters for managing connections as well. On both the client and server, connections can be manipulated by configuring the maximum number of concurrent connections with the `connection_max()` method. When the maximum number of connections is reached, the least-recently-used connection times out and is used for a new connection. These parameters apply to all BOAs.

See "Thread management parameters" on page 8-9 and "Connection management parameters" on page 8-10 for more information.

Setting thread and connection parameters

Thread and connection parameters can be set in one of three ways:

- ORB and BOA APIs—You make calls on these methods from within your CORBA application.
- `ORB_init()` and `BOA_init()`—You make calls on these methods to update the state of the ORB or BOA at runtime.
- Command-line—You make calls on these methods by passing options when the server is started.

Command-line arguments override the default settings in the server object for the thread management options. For example, the following command starts the `account_server` object on the server with a maximum thread pool size of 50 threads.

```
% account_server -OAthreadMax 50
```

This command-line argument works because the example code obtains a handle to the BOA using `BOA_init()`, which takes the command-line arguments as parameters.

Consider the following code sample that shows how the ORB and BOA are initialized.

Code Sample 8.3 Initialization of the ORB and BOA

```
int main(int argc, char* const* argv {
   try {
         //Initialize the ORB and BOA
         CORBA::ORB_var orb=CORBA::ORB_init(argc, argv);
         CORBA::BOA_var boa=orb->BOA_init(argc, argv);
   ...
```

Thread and connection management can be changed from within the body of the program using the BOA API. Code Sample 8.4 shows usage of the `thread_max()` BOA API.

Code Sample 8.4 Setting thread and connection management policies using BOA APIs

```
...
boa->thread_max(50);
...
```

Alternatively, thread and connection management can be changed using calls to `ORB_init()` or `BOA_init()` from within the body of the program. Code Sample 8.5 shows usage of `BOA_init()` to set the maximum number of threads.

Code Sample 8.5 Setting thread and connection management policies using BOA_init()

```
...
int argc=2;
char *argv[]={"-OAthreadMax", "50"};
orb->BOA_init(argc, argv);
```

Some parameters that are specific to a particular BOA do not have BOA APIs, and therefore can only be set at runtime using `BOA_init()`, as shown above.

Thread management parameters

Table 8.3 shows the thread APIs and command-line arguments that can be set to alter thread management on the server side.

Table 8.3 Thread management parameters for the server side

Method	Arguments	Description
`void thread_max(CORBA::ULong)`	`-OAthreadMax`	Used to specify the maximum number of threads the BOA can allocate to the thread pool. By default, no limit is placed on the number of threads in the thread pool.
`CORBA::ULong thread_max()`		Used to query the number of threads currently in the thread pool. When this method returns 0, the thread pool will continue to grow without any maximum.
`void thread_stack_size(CORBA::ULong)`	`-OAthreadStackSize`	Used to specify the stack size of threads (in bytes) that the BOA creates. By default, this is the default thread stack size that the operating system's thread creation method supports.

Table 8.3 Thread management parameters for the server side (continued)

Method	Arguments	Description
`CORBA::ULong thread_stack_size()`		Used to query the thread stack size.
	`-OAthreadMaxIdle`	Use to specify the time (in seconds) which a thread can exist without servicing any requests. Threads that are idle beyond the time specified can be returned to the system by VisiBroker. By default, this is set to 300 seconds. **Note:** This option is only applicable to the thread pooling BOA.

Connection management parameters

Table 8.4 shows the thread APIs and command-line arguments that can be set to alter connection management on the client or server.

Table 8.4 Connection management APIs for the client and server

Method	Arguments	Description
`void connection_max (CORBA::ULong max_conn)`	`-OAconnectionMax (server)` `-ORBconnectionMax (client)`	Used to set the maximum number of concurrent connections the BOA or ORB can accept. By default, this is set to the maximum number of file descriptors the operating system supports.
`CORBA::ULong connection_max()`		Used to query the maximum number of concurrent connections allowed for the BOA or ORB.
`CORBA::ULong connection_count()`		Used to query the current number of connections that the BOA or ORB has outstanding.
	`-OAconnectionMaxIdle (server)` `-ORBconnectionMaxIdle (client)`	Used to specify how long (in seconds) a connection can be idle without any traffic. Connections that are idle beyond this time can be shutdown by VisiBroker. By default, this is set to 0 which means that connections will never automatically time-out. This option should be set for Internet applications.

Chapter

9

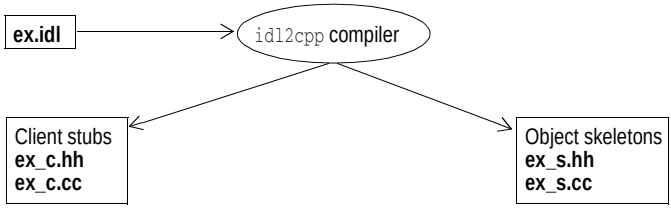
Using the IDL to C++ compiler

This chapter describes the VisiBroker IDL compiler. It includes the following major sections:

How the IDL compiler generates C++ 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 C++ code. Figure 9.1 shows how the compiler generates code for the client program and for the object implementation, or server.

Figure 9.1 C++ files generated by the IDL compiler



For details on usage syntax for the `idl2cpp` compiler, see the *VisiBroker for C++ Reference*.

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. IDL sample 9.1 shows an IDL specification for an object named `example`. The `example` object has only one method, `op1`.

IDL sample 9.1 example IDL specification

```
// IDL specification for the example object
interface example
{
      long op1(in char x, out short y);
};
```

Looking at code generated for clients

Code sample 9.1 shows how the IDL compiler generates two client files—**ex_c.hh** and **ex_c.cc**. These two files provide an `example` class in C++ that the client will use. By convention, files generated by the IDL compiler always have either a "**.cc**" or "**.hh**" suffix to make them easy to distinguish from files that you create yourself. If you wish, you can alter the convention to produce files with a different suffix. See the *VisiBroker for C++ Reference*.

Caution You should not modify the contents of the files generated by the IDL compiler.

Code sample 9.1 example class generated in ex_c.hh

```
class example : public virtual CORBA::Object {
   protected:
         example(const char *obj_name = NULL) : CORBA::Object(obj_name, 1);
         example(NCistream& strm) :CORBA::Object(strm);
         virtual ~example();
   public:
         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 _duplicate(example_ptr obj);
         static example_ptr _nil();
         static example_ptr _narrow(CORBA::Object *obj);
         virtual CORBA::Long op1(CORBA::Char x, CORBA::Short& y);
};
```

Methods (stubs) generated by the IDL compiler

Code sample 9.1 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. The CORBA::Object class methods are described in Chapter 5, "Accessing distributed objects with object references."

Example pointer type (_ptr) definition

The IDL compiler always provides a pointer type definition. Code sample 9.2 shows the type definition for the example class.

Code sample 9.2 _ptr type definition

```
typedef example *example_ptr;
```

Example class with automatic memory management (_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.

See "Argument passing considerations" on page 10-1 for information about the difference between _ptr and _var.

Code sample 9.3 example_var class

```
class example_var {
   public:
            example_var();
            example_var(example_ptr ptr);
            example_var(const example_var& var);
            ~example_var();
            example_var& operator=(example_ptr p);
            example operator=(const example_ptr p);
            example_ptr operator->();
            ...
      protected:
            example_ptr _ptr;
};
```

The following table describes the methods in the _var class.

Table 9.1 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.
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 code generated for servers

Code sample 9.4 shows how the IDL compiler generates two server files: **ex_s.hh** and **ex_s.cc**. These two files provide an _sk_example class in C++ that the server will use to derive an implementation class. The _sk_example class is derived from the client's example class.

Caution You should not modify the contents of the files generated by the IDL compiler.

Code sample 9.4 _sk_example class generated in ex_s.hh

```
class _sk_example : public example {
   protected:
            _sk_example(const char *object_name = (const char *)NULL);
            virtual ~_sk_example();
   public:
            virtual CORBA::Long op1(CORBA::Char x, CORBA::Short& y) = 0;
};
```

Methods (skeletons) generated by the IDL compiler

Notice that the op1 method defined in the IDL specification in IDL sample 9.1 is generated, along with an _op1 method. The _sk_example class declares a pure virtual method named op1. The implementation class that is derived from _sk_example must provide an implementation for this method.

The _sk_example class is called a *skeleton* and its method (_op1) is invoked by the BOA 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. See also "Globally scoped objects" on page 6-3.

Class template generated by the IDL compiler

In addition to the `_sk_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 `_sk_example`. Templates can be useful for providing a wrapper class for existing applications that cannot be modified to inherit from a new class. Code sample 9.5 shows the template class generated by the IDL compiler for the `example` class.

Code sample 9.5 Template class generated for the example class

```
template <class T>
class _tie_example : public example
{
   public:
            _tie_example(T& t, const char *obj_name=(char *)NULL);
            ~_tie_example();
            CORBA::Long op1(CORBA::Char x, CORBA::Short& y);
   private:
            T& _ref;
};
```

For complete details on using the `_tie` template class, see Chapter 12, "Using the tie mechanism: An alternative to inheritance."

You may also generate a `_ptie` template for integrating an object database with your servers, as described in Chapter 18, "Enabling object persistence using the Object Database Activator."

Defining interface attributes in the IDL

In addition to operations, an interface specification can also define attributes as part of the interface. By default, all attributes are *read-write* and the IDL compiler will generate two methods—one to set the attribute's value, and one to get the attribute's value. You can also specify *read-only* attributes, for which only the reader method is generated.

IDL sample 9.2 shows an IDL specification that defines two attributes—one read-write and one read-only. Code sample 9.6 shows the resulting class definition generated by the IDL compiler for the client program. Code sample 9.7 shows the class definition generated for the object implementation.

IDL sample 9.2 IDL specification with two attributes—one read-write and one read-only

```
// IDL
interface test
{
   attribute long count;
   readonly attribute string name;
};
```

Code sample 9.6 Class generated for the client program

```
class test : public virtual CORBA::Object {
      ...
      // Methods for read-write attribute
      virtual CORBA::Long count();
      virtual void count(CORBA::Long val);

      // Method for read-only attribute.
      virtual char * name();
      ...
};
```

Code sample 9.7 Class generated for the server

```
class _sk_test : public test {
//The following operations need to be implemented.
      virtual CORBA::Long count() = 0;
      virtual void count(CORBA::Long val) = 0;
      virtual char * name() = 0;
};
```

Specifying oneway methods with no return value

IDL allows you to specify operations that have no return value, called *oneway* methods. These operations may only have input parameters. When a `oneway` method is invoked, a request is sent to the server but there is no confirmation from the object implementation that the request was actually received. VisiBroker uses TCP/IP for connecting clients to servers. This provides reliable delivery of all packets so the client can be sure the request will be delivered to the server, as long as the server remains available. Still, the client has no way of knowing if the request was actually processed by the object implementation itself.

Note Oneway operations cannot raise exceptions or return values.

IDL sample 9.3 Defining a oneway operation

```
// IDL
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 C++ classes generated by the IDL compiler will reflect the inheritance relationship. All methods, data type definitions, constants and enumerations declared by the parent interface will be visible to the derived interface.

IDL sample 9.4 Example of inheritance in an interface specification

```
// IDL
interface parent
{
   void operation1();
}

interface child : parent
{
   ...
   long operation2(in short s);
};
```

Code sample 9.8 shows the C++ code that is generated from the interface specification shown in IDL sample 9.4.

Code sample 9.8 C++ code generated from IDL sample 9.4

```
...
class parent : public virtual CORBA::Object
{
   ...
   void operation1( );
   ...
};

class child : public virtual parent
{
   ...
   CORBA::Long operation2(CORBA::Short s);
   ...
};
```

Chapter

10

Parameter passing rules for the IDL to C++ mapping

This chapter describes the parameter passing rules followed by the CORBA IDL to C++ mapping. It includes the following major sections:

Overview

The information in this chapter is reprinted with permission from Chapter 16, section 18, of *The Common Object Request Broker: Architecture and Specification—97-02-25*. For additional information consult the OMG document located on the OMG web site at http://www.omg.org.

Argument passing considerations

The mapping of parameter passing modes balances the need for both efficiency and simplicity. For primitive types, enumerations, and object references, the modes are straightforward—passing the type *P* for primitives and enumerations and the type `A_ptr` for an interface type *A*.

Aggregate types are complicated by the question of when and how parameter memory is allocated and deallocated. Mapping `in` parameters is straightforward

because the parameter storage is caller-allocated and read-only. The mapping for `out` and `inout` parameters is more problematic. For variable-length types, the callee must allocate some, if not all, of the storage. For fixed-length types, such as a *Point* type represented as a struct containing three floating point members, caller allocation is preferable (to allow stack allocation).

To accommodate both kinds of allocation, avoid the potential confusion of split allocation, and eliminate confusion with respect to when copying occurs, the mapping is `T&` for a fixed-length aggregate `T`, and `T*&` for a variable-length `T`. This approach has the unfortunate consequence that usage for structs depends on whether the struct is fixed- or variable-length; however, the mapping is consistently `T_var&` if the caller uses the managed type `T_var`.

The mapping for `out` and `inout` parameters additionally requires support for deallocating any previous variable-length data in the parameter when a `T_var` is passed. Even though their initial values are not sent to the operation, `out` parameters are included because the parameter could contain the result from a previous call. The mapping requires that a compliant implementation must free the inaccessible storage associated with a parameter passed as a `T_var` managed type. The following examples demonstrate this behavior.

IDL sample 10.1 Deallocator behavior

```
struct S {string name; float age;};
void f(out S p);
```

Code sample 10.1 Deallocator behavior

```
// C++
S_var s;
f(s.out());
// use s
f(s.out());             // first result will be freed

S *sp;                  // need not initialize before passing to out
f(sp);
// use sp
delete sp;              // cannot assume next call will free old value
f(sp);
```

Note Implicit deallocation of previous values for `out` and `inout` parameters works only with `T_var` types, not with other types.

```
void q(out string s);

// C++
char *s;
for (int i = 0; i < 10; i++)
   q(s);                // memory leak
```

Each call to the q function in the loop results in a memory leak because the caller is not invoking string_free on the out result. There are two ways to fix this, as shown in the following code sample:

```
// C++
char *s;
String_var svar;
for (int i = 0; i < 10; i++) {
   q(s);
   string_free(s);      // explicit deallocation
   // OR:
   q(svar.out());       // implicit deallocation
}
```

Using a plain char* for the out parameter means that the caller must explicitly deallocate its memory before each reuse of the variable as an out parameter, while using a String_var means that any deallocation is performed implicitly upon each use of the variable as an out parameter.

Variable-length data must be explicitly released before being overwritten. For example, before assigning to an inout string parameter, the implementer of an operation may first delete the old character data. Similarly, an inout interface parameter should be released before being reassigned. One way to ensure that the parameter storage is released is to assign it to a local T_var variable with an automatic release, as in the following example.

```
interface A;
void f(inout string s, inout A obj);

// C++
void Aimpl::f(char *&s, A_ptr &obj){
   String_var s_tmp = s;
   s = /* new data */;
   A_var obj_tmp = obj;
   obj = /* new reference */;
}
```

To allow the callee the freedom to allocate a single contiguous area of storage for all the data associated with a parameter, the OMG has adopted the policy that the callee-allocated storage is not modifiable by the caller. However, trying to enforce this policy by returning a const type in C++ is problematic, since the caller is required to release the storage, and calling delete on a const is an error. A compliant mapping therefore is not required to detect this error.

Passing null data in to and out of IDL-defined operations

For parameters that are passed or returned as a pointer (T*) or reference to pointer (T*&), a compliant program is not allowed to pass or return a null pointer; the result of doing so is undefined. In particular, a caller may not pass a null pointer under any of the following circumstances:

- in and inout string
- in and inout array (pointer to first element)

A caller may pass a reference to a pointer with a null value for the `out` parameters, however, since the callee does not examine the value but rather just overwrites it. A callee may not return a null pointer under any of the following circumstances:

- `out` and return variable-length struct
- `out` and return variable-length union
- `out` and return string
- `out` and return sequence
- `out` and return variable-length array, return fixed-length array
- `out` and return any

Since OMG IDL has no concept of pointers in general, or null pointers in particular, allowing the passage of null pointers to or from an operation would project C++ semantics onto OMG IDL operations. A `BAD_PARAM` exception is raised if this error is detected.

Summary of mapping IDL to C++ parameter types

Table 10.1 displays the mapping for the basic OMG IDL parameter passing modes and return type according to the type being passed or returned. Fixed-length arrays are the only case where the type of an `out` parameter differs from a return value, which is necessary because C++ does not allow a function to return an array. The mapping returns a pointer to a *slice* of the array, where a slice is an array with all the dimensions of the original specified except the first one.

Table 10.1 Basic argument and result passing

Data type	In	Inout	Out	Return
short	Short	Short&	Short&	Short
long	Long	Long&	Long&	Long
longlong	LongLong	LongLong&	LongLong&	LongLong
unsigned short	UShort	UShort&	UShort&	UShort
unsigned long	ULong	ULong&	ULong&	ULong
unsigned longlong	ULongLong	ULongLong&	ULongLong&	ULongLong
float	Float	Float&	Float&	Float
double	Double	Double&	Double&	Double
long double	LongDouble	LongDouble&	LongDouble&	LongDouble
boolean	Boolean	Boolean&	Boolean&	Boolean
char	Char	Char&	Char&	Char
wchar	WChar	WChar&	WChar&	WChar
octet	Octet	Octet&	Octet&	Octet
enum	enum	enum&	enum&	enum
object reference ptr[1]	objref_ptr	objref_ptr&	objref_ptr&	objref_ptr
struct, fixed	const struct&	struct&	struct&	struct
struct, variable	const struct&	struct&	struct*&	struct*
union, fixed	const union&	union&	union&	union
union, variable	const union&	union&	union*&	union*

Table 10.1 Basic argument and result passing (continued)

Data type	In	Inout	Out	Return
string	const char*	char*&	char*&	char*
wstring	const WChar*	WChar*&	WChar*&	WChar*
sequence	const sequence&	sequence&	sequence*&	sequence*
array, fixed	const array	array	array	array slice*[2]
array, variable	const array	array	array slice*&[2]	array slice*[2]
any	const any&	any&	any*&	any*

1. Including pseudo-object references.
2. A slice is an array with all the dimensions of the original except the first one.

A caller is responsible for providing storage for all arguments passed as `in` arguments.

C++ memory management rules for parameter passing

Table 10.2 lists the case number associated with the storage management responsibilities, by data type, required for `inout` and `out` parameters and for return results.

Table 10.2 Caller argument storage responsibilities

Type	Inout param	Out param	Return result
short	1	1	1
long	1	1	1
long long	1	1	1
unsigned short	1	1	1
unsigned long	1	1	1
unsigned long long	1	1	1
float	1	1	1
double	1	1	1
long double	1	1	1
boolean	1	1	1
char	1	1	1
wchar	1	1	1
octet	1	1	1
enum	1	1	1
object reference ptr	2	2	2
struct, fixed	1	1	1
struct, variable	1	3	3
union, fixed	1	1	1
union, variable	1	3	3
string	4	3	3
wstring	4	3	3

Table 10.2 Caller argument storage responsibilities (continued)

Type	Inout param	Out param	Return result
sequence	5	3	3
array, fixed	1	1	6
array, variable	1	6	6
any	5	3	3

Based on the number specified in Table 10.2 for the inout and out parameters and for return results, the following cases outline the memory storage responsibilities of the caller and callee. Argument passing rules specific to data types are illustrated through examples of IDL interfaces with corresponding stub, client code, and, where appropriate, server code.

Case 1

The caller allocates all necessary storage, except that which may be encapsulated and managed within the parameter itself. For inout parameters, the caller provides the initial value, and the callee may change that value. For out parameters, the caller allocates the storage but need not initialize it, and the callee sets the value. Function returns are by value.

Parameter passing rules for basic data types

The examples in this section illustrate parameter passing rules for basic or primitive data types. The primitive data types include,

- short
- long
- unsigned short
- unsigned long
- long long
- unsigned long long
- float
- double
- boolean
- char
- wchar
- octet
- enum

in parameters

in parameters are passed by value.

IDL sample 10.2 IDL interface with a primitive data type as the in parameter

```
interface foo
   {
      void primitive(in long aLong);
   };
```

Generated stub

```
class foo {
   ...
      void primitive(CORBA::Long aLong);
};
```

Client code

```
foo *obj = ...;
CORBA::Long x = 10;
obj->primitive(x);
```

out and inout Parameters

`out` and `inout` parameters are passed by reference.

IDL sample 10.3 IDL interface with a primitive data type as the out and inout parameter

```
interface foo
{
   void primitive_inout(inout long aLong);
   void primitive_out(out long aLong);
};
```

Generated stub

```
class foo {
   ...
   void primitive_inout(CORBA::Long& aLong);
   void primitive_out(CORBA::Long& aLong);
};
```

Client code

```
foo *obj = ...;
CORBA::Long x = 10;
CORBA::Long y = 0;
obj->primitive_inout(x);
obj->primitive_out(y);
```

Server code

```
class foo_impl: public _sk_foo
{
   ...
   void primitive_inout(CORBA::Long& aLong) {
      cout << "Received: " << aLong;// receives 10;
      aLong = 5;// Change inout to 10;
   }
   void primitive_out(CORBA::Long& aLong) {
      aLong = 2;// Set out value
   }
};
```

Return parameters

Return parameters are passed by value.

IDL sample 10.4 IDL interface with a primitive data type as the return parameter

```
interface foo {
   long primitive();
};
```

Generated stub

```
class foo {
   CORBA::Long primitive();
};
```

Client code

```
foo *obj = ...;
CORBA::Long ret = obj->primitive();
```

Server code

```
class foo_impl: public _sk_foo
{
   ...
   CORBA::Long primitive() {
         return 5;
   }
};
```

Parameter passing rules for fixed and variable length data structures

CORBA defines structured types (structs, unions, arrays, and sequences) as fixed or variable type depending on whether the data structure is fixed-length or variable-length. A type is variable-length if it is one of the following types:

- The type `any`
- A bounded or unbounded string
- A bounded or unbounded sequence
- An Object reference
- A `struct` or `union` that contains a member whose type is variable-length
- An `array` with a variable-length element type
- A `typedef` to a variable-length type

in parameters

Mappings for fixed-length type (structs, unions, arrays, and sequences) `in` arguments are passed as const reference. The caller allocates the data structures and passes it to the callee.

IDL sample 10.5 IDL interface with fixed-length in parameters

```
struct S { long a; short b; };
union U switch(long) { case 1: long a; default: short b; };
typedef octet A[64];

interface foo {
   void test_struct(in S aS);
   void test_union(in U aU);
   void test_array(in A aA);
};
```

Generated stub

```
class foo {
   void test_struct(const S& aS);
   void test_union(const U& aU);
   void test_array(const A& aA);
};
```

Client code

```
foo *obj = ...;
S aStruct;
U anUnion;
A anArray;

foo->test_struct(aStruct);
foo->test_union(anUnion);
foo->test_array(anArray);

// _var can also be used as follows
S_var sVar = new S;
U_var uVar = new U;
A_var aVar = new A;

// _var classes have conversion operators to convert _var to
// const ..& as such _var can be directly passed to the operation.
foo->test_struct(sVar);
foo->test_union(uVar);
foo->test_array(aVar);
```

Server code

```
class foo_impl: public _sk_foo
{
   ...
   void test_struct(const S& aStruct) {
      ...
   }
   void test_union(const U& anUnion) {
      ...
   }
   void test_array(const A& anArray) {
      ...
   }
};
```

Mappings for variable-length types (struct, union, array, sequences, and any) `in` arguments are passed as const reference. The caller allocates the data structures and passes it to the callee.

IDL sample 10.6 IDL interface with variable-length in parameters

```
struct S { string a; short b; };
union U switch(long) { case 1: long a; default: string b; };
typedef string A[64];
typedef sequence<string> Seq;
```

```
interface foo {
   void test_struct(in S aS);
   void test_union(in U aU);
   void test_array(in A aA);
   void test_seq(in Seq s);
   void test_any(in any anAny);
};
```

Generated stub

```
class foo {
   void test_struct(const S& aS);
   void test_union(const U& aU);
   void test_array(const A& aA);
   void test_seq(const Seq& s);
   void test_any(const CORBA::Any& anAny);
};
```

Client code

```
foo *obj = ...;
S aStruct;
U anUnion;
A anArray;
Seq s;
CORBA::Any anAny;

foo->test_struct(aStruct);
foo->test_union(anUnion);
foo->test_array(anArray);
foo->test_seq(s);
foo->test_any(anAny);

// _var can also be used as follows
S_var sVar = new S;
U_var uVar = new U;
A_var aVar = new A;
Seq_var seqVar = new Seq;
CORBA::Any_var anyVar = new CORBA::Any;

// _var classes have conversion operators to convert _var to
// const ..& as such _var can be directly passed to the operation.
foo->test_struct(sVar);
foo->test_union(uVar);
foo->test_array(aVar);
foo->test_seq(seqVar);
foo->test_any(anyVar);
```

Server code

```
class foo_impl: public _sk_foo
{
   ...
   void test_struct(const S& aStruct) {
      ...
   }
```

```
    void test_union(const U& anUnion) {
       ...
    }
    void test_array(const A& anArray) {
       ...
    }
    void test_seq(const Seq& aSeq) {
       ...
    }
    void test_any(const CORBA::Any& aSeq) {
       ...
    }
};
```

out and inout parameters

For `inout` parameters the caller sets the initial value. The callee changes these values. For `out` parameters, the caller need not set the initial values. The callee sets the values for `out` parameters.

IDL sample 10.7 IDL interface with fixed-length structured out and inout parameters

```
struct S { long a; short b; };
union U switch(long) { case 1: long a; default: short b; };
typedef octet A[64];

interface foo {
   void test_struct(out S outS, inout S inoutS);
   void test_union(out U outU, inout U inoutU);
   void test_array(out A outA, inout A inoutA);
};
```

Generated stub

```
class foo {
   void test_struct(S& outS, S& inoutS);
   void test_union(U& outS, U& inoutS);
   void test_array(A& outA, A& inoutA);
};
```

Client code

```
foo *obj = ...;
S outS, inoutS;
U outS, inoutS;
A outA, inoutA;

foo->test_struct(outS, inoutS);
foo->test_union(outU, inoutU);
foo->test_array(outA, inoutA);

// _var can also be used as follows
S_var outSVar;
S_var inoutSVar = new S;
U_var outUVar;
U_var inoutUVar = new U;
```

```
A_var outAVar;
A_var inoutAVar = A_alloc();

// _var classes have conversion operators to convert _var to
// ..& as such _var can be directly passed to the operation.
foo->test_struct(outSVar.out(), inoutSVar.inout());
foo->test_union(outUVar.out(), inoutUVar.inout());
foo->test_array(outAVar.out(), inoutAVar.inout());
```

Server code

```
class foo_impl: public _sk_foo
{
   ...
   void test_struct(S& outS, S& inoutS) {
      ...
   }
   void test_union(U& outU, U& inoutU) {
      ...
   }
   void test_array(A outA, A inoutA) {
      ...
   }
};
```

Return parameters

Return parameters are passed by value. For examples, see the `struct` and `union` in IDL sample 10.16 on page 10-23.

Case 2

The caller allocates storage for the object reference. For `inout` parameters, the caller provides an initial value; if the callee wants to reassign the `inout` parameter, it will first call CORBA::release on the original input value. To continue to use an object reference passed in as an `inout`, the caller must duplicate the reference prior to returning from the method. The caller is responsible for the release of all `out` and return object references. Release of all object references embedded in other structures is performed automatically by the structures themselves.

in parameters

`in` parameters are passed as pointers to an Object.

IDL sample 10.8 IDL interfaces with a reference to an Object as an in parameter

```
interface f {
   void t();
};
interface foo
{
   void test(in f obj);
};
```

Generated stub

```
class foo {
   ...
   void test(f_ptr obj);
};
```

Client code

```
foo *obj = ...;
f *arg = ...;
obj->test(f);
// The _var can also be used as follows.
f_var arg = ...;
obj->test(arg);
```

Server code

```
class foo_impl: public _sk_foo
{
   ...
   // The in parameter is passed as f_ptr and the callee
   // should not release it.
   void test_(f_ptr obj) {
      ...
      // If the callee needs to hold onto the reference, the
      // reference may be duplicated
      f_ptr copyObj = f::_duplicate(obj);
   }
};
```

out and inout parameters

`out` and `inout` parameters are passed by reference to an Object pointer. The caller allocates memory for the Object reference. For `inout` parameters the caller assigns an initial value. If the callee wants to reassign the `inout` parameter, the original value is first released. To continue to use the parameter passed as the `inout` parameter, the callee needs to duplicate the reference before assigning it to the `inout` parameter. The caller is responsible for releasing the `out` and `inout` parameters.

Note All objects must be released by calls to CORBA::release or CORBA::Object::_release. Never call the C++ operator delete on a CORBA::Object.

IDL sample 10.9 IDL interfaces with out and inout parameters passed by reference to an Object pointer

```
interface f {
   void t();
};
interface foo
{
   void test_inout(inout f obj);
   void test_out(out f obj);
};
```

Generated stub

```
class foo {
   ...
   void test_inout(f_ptr & obj);
   void test_out(f_ptr & obj);
};
```

Client code

```
foo_ptr obj = ...;
f_ptr finout = ...;
f_ptr fout = NULL;
obj->test_inout(finout);
CORBA::release(finout);
obj->test_out(fout);
CORBA::release(fout);
// Using _var
foo_var obj = ...;
f_var finout = ...;
f_var fout;
obj->test_inout(finout.inout());
obj->test_out(fout.out());
```

Server code

```
class foo_impl: public _sk_foo
{
   ...
   void test_inout(f_ptr& obj) {
      // Release input argument.
      CORBA::release(obj);
      // Or use _var to release c on return;
      // f_var relObj(obj);
      // Set out argument
      obj = ...;
   }
   void test_out(f_ptr& obj) {
      obj = ...;
   }
};
```

Return parameters

Return parameters are passed as Object pointers. The callee allocates the Object and returns it. To continue to use the return parameter, the callee needs to duplicate the reference before returning it to the caller.

IDL sample 10.10 IDL interfaces showing return parameter passed as an Object pointer

```
interface f {
   void t();
};
interface foo {
   f test();
};
```

Generated stub

```
class foo {
   f_ptr test();
};
```

Client code

```
foo_ptr obj = ...;
f_ptr ret = obj->test();
...
CORBA::release(ret);// Release the reference.
// or use the _var
f_var retObj = obj->test();
```

Server code

```
class foo_impl: public _sk_foo
{
   ...
   f_ptr test() {
      return ...;
   }
};
```

Case 3

For `out` parameters, the caller allocates a pointer and passes it by reference to the callee. The callee sets the pointer to point to a valid instance of the parameter's type. For returns, the callee returns a similar pointer. The callee is not allowed to return a null pointer in either case. In both cases, the caller is responsible for releasing the returned storage. To maintain local/remote transparency, the caller must always release the returned storage, regardless of whether the callee is located in the same address space as the caller, or is located in a different address space. Following the completion of a request, the caller is not allowed to modify any values in the returned storage—to do so, the caller must first copy the returned instance into a new instance, then modify the new instance.

Parameter passing rules for strings

out and inout Parameters

`out` and `inout` parameters are passed by reference to a char *. The caller allocates memory for `inout` parameter. The callee must release the `inout` parameter before setting it to memory allocated by the callee. The caller is responsible for releasing the memory allocated for the `inout`/`out` parameter by the callee. String allocations and deallocations must be performed by calls to the functions CORBA::string_alloc, CORBA::string_dup, and CORBA::string_free.

IDL sample 10.11 IDL interface with strings for out and inout parameters

```
interface foo
{
   void test_inout(inout string s);
   void test_out(out string s);
};
```

Generated stub

```
class foo {
   ...
   void test_inout(char *& s);
   void test_out(char *& s);
};
```

Client code

```
foo *obj = ...;
char *ios = CORBA::string_dup("in string");
char *os = NULL;
obj->test_inout(ios);
CORBA::string_free(ios);// Release inout
obj->test_out(os);
CORBA::string_free(os);// Release out
// Using _var.
CORBA::String_var ioString = (const char *)"in string";
CORBA::String_var oString;
obj->test_inout(ioString.inout());
obj->test_out(oString.inout());
```

Server code

```
class foo_impl: public _sk_foo
{
   ...
   // The callee needs to release the inout arg before assigning
   // it a new value. The parameter may be modified in place
   // as long as it does not extend the length.
   void test_inout(char *& c) {
      cout << "Received: " << c;

      // Release input parameter.
      CORBA::string_free(c);
      // Or use _var to release c on return;
      // CORBA::String_var relStr(c);
      // Set out parameter
      c = CORBA::string_dup("out string"); // new string
   }
   // The following usage of test_inout is also valid
   void test_inout(char *& c) {
      // Modify the string in place
      for (int i=0; i< strlen(c); i++)
         c[i] = toupper(c);
   }
```

```
   void test_out(char *& c) {
      c = CORBA::string_dup("out string");// Set out value
   }
};
```

Return parameters

Return parameters are passed as char *. The callee allocates the memory and returns it. The caller should release the return value.

IDL sample 10.12 IDL interface with a string return parameter

```
interface foo {
   string test();
};
```

Generated stub

```
class foo {
   char *test();
};
```

Client code

```
foo *obj = ...;
char *ret = obj->test();
CORBA::string_free(ret);
// or use the _var
CORBA::String_var retStr = obj->test();
```

Server code

```
class foo_impl: public _sk_foo
{
   ...
   char *test() {
      return CORBA::strdup("return value");
   }
};
```

Parameter passing rules for variable-length data structures

out and inout parameters

For `out` parameters the caller allocates memory for the pointer and passes it by reference to the callee. The callee sets the pointer to a valid instance of the parameter's type. For `inout` parameters the caller sets the pointer to an initial value. The caller is responsible for releasing the storage for `in` and `inout` parameters.

IDL sample 10.13 IDL interface showing structures as out and inout parameters

```
struct S { string a; short b; };
union U switch(long) { case 1: long a; default: string b; };
typedef string A[64];
typedef sequence<string> Seq;
```

```
interface foo {
   void test_struct(out S outS, inout S inoutS);
   void test_union(out U outU, inout U inoutU);
   void test_array(out A outA, inout A inoutA);
   void test_seq(out Seq outSeq, inout Seq inoutSeq);
   void test_any(out Any outAny, inout Any inoutAny);
};
```

Generated stub

```
class foo {
   void test_struct(S& outS, S& inoutS);
   void test_union(U& outS, U& inoutS);
   void test_array(A& outA, A& inoutA);
   void test_seq(Seq *&outSeq, Seq *& inoutSeq);
   void test_any(CORBA::Any *&outAny, CORBA::Any *& inoutAny);
};
```

Client code

```
foo *obj = ...;
S *outS = NULL;
S *inoutS = new S;
U *outS = NULL;
U *inoutS = new U;
A *outA = NULL;
A *inoutA = A_alloc();
Seq *outSeq = NULL;
Seq *inoutSeq = new Seq;
COBRA::Any *outAny = NULL;
CORBA::Any *inoutAny = new CORBA::Any;

foo->test_struct(outS, inoutS);
foo->test_union(outU, inoutU);
foo->test_array(outA, inoutA);
foo->test_seq(outSeq, inoutSeq);
foo->test_any(outAny, inoutAny);

// _var can also be used as follows
S_var outSVar;
S_var inoutSVar = new S;
U_var outUVar;
U_var inoutUVar = new U;
A_var outAVar;
A_var inoutAVar = A_alloc();
Seq_var outSeqVar;
Seq_var inoutSeqVar = new Seq;
Any_var outAnyVar;
Any_var inoutAnyVar = new CORBA::Any;

// _var classes have conversion operators to convert _var to
// ..& as such _var can be directly passed to the operation.
foo->test_struct(outSVar, inoutSVar);
foo->test_union(outUVar, inoutUVar);
foo->test_array(outAVar, inoutAVar);
```

```
foo->test_seq(outSeqVar, inoutSeqVar);
foo->test_any(outAnyVar, inoutAnyVar);
// Some compilers may have problems using conversion operators
// to convert from _var to appropriate types. To overcome these
// problems the appropriate in/inout functions can be used.
foo->test_struct(outSVar.out(), inoutSVar.inout());
foo->test_union(outUVar.out(), inoutUVar.inout());
foo->test_array(outAVar.out(), inoutAVar.inout());
foo->test_seq(outSeqVar.out(), inoutSeqVar.inout());
foo->test_any(outAnyVar.out(), inoutAnyVar.inout());
```

Server code

```
class foo_impl: public _sk_foo
{
   ...
   void test_struct(S *& outS, S *& inoutS) {
      outS = new S;
      cout << "Received inout: " << inoutS << endl;
      // set inout to a new value.
      delete inoutS;// Release received value
      inoutS = new S; // Set it to new value.
      ...

   }
   void test_union(U *& outU, U *& inoutU) {
      outU = new U;
      cout << "Received inout: " <<  inoutU << endl;
      // Change inout value.
      inoutU->a(15);
   }
   void test_array(A_slice *& outA, A inoutA) {
      outA = A_alloc();
      ...
      inoutA[1] = (const char *)"changed";
   }
   void test_seq(Seq *& outSeq, Seq *& inoutSeq) {
      outSeq = new Seq;
      ...
      // reset inout value using _var
      Seq_var nSeq = new Seq;
      Seq_var relSeq(inoutSeq);// releases inout
      inoutSeq = nSeq._retn();
   }
   void test_any(CORBA::Any *& outAny, CORBA::Any *& inoutAny) {
      outAny = new CORBA::Any;
      ...
      // reset inout value using _var
      Any_var nAny = new Any;
      Any_var relAny(inoutAny);// releases inout
      inoutAny = nAny._retn();
   }
};
```

Return parameters

Return parameters are passed as pointers by the callee. The caller is responsible for releasing the storage.

IDL sample 10.14 IDL interface showing structures as return parameters

```
struct S { long a; string b; };
union U switch(long) { case 1: long a; default: string b; };
typedef string A[64];
typedef sequence<string> Seq;

interface foo {
   S test_struct();
   U test_union();
   A test_array();
   Seq test_seq();
   Any test_any();
};
```

Generated stub

```
class foo {
   S *test_struct();
   U *test_union();
   A_slice *test_array();
   Seq *test_seq();
   Any *test_array();
};
```

Client code

```
foo *obj = ...;
S *rS = foo->test_struct();
U *rU = foo->test_union();
A_slice *rA = foo->test_array();
Seq *rSeq = foo->test_seq();
Any *rAny = foo->test_any();
..
delete rS;
delete rU;
A_free(rA);
delete rSeq;
delete rAny;

//or using _var
S_var rS = foo->test_struct();
U_var rU = foo->test_union();
A_var rA = foo->test_array();
Seq_var rSeq = foo->test_seq();
Any_var rAny = foo->test_any();
```

Server code

```
class foo_impl: public _sk_foo {
...
   S *test_struct() {
      S *r = new S;
      ..
      return r;
      // Or if using _var
      S_var r = new S;
      ...
      return r._retn();
   }
   U *test_union() {
      U *r = new U;
      ..
      return r;
      // Or if using _var
      U_var r = new U;
      ...
      return r._retn();
   }
   A_slice *test_array() {
      A_slice *r = A_alloc();
      ..
      return r;
      // Or if using _var
      A_var r = A_alloc();
      ...
      return r._retn();
   }
   Seq *test_seq() {
      Seq *r = new Seq;
      ..
      return r;
      // Or if using _var
      Seq_var r = new Seq;
      ...
      return r._retn();
   }
   CORBA::Any *test_any() {
      CORBA::Any *r = new CORBA::Any;
      ..
      return r;
      // Or if using _var
      CORBA::Any_var r = new CORBA::Any;
      ...
      return r._retn();
   }
};
```

Case 4

For inout strings, the caller provides storage for both the input string and the char* pointing to it. Since the callee may deallocate the input string and reassign the char* to point to new storage to hold the output value, the caller should allocate the input string using string_alloc(). The size of the out string is therefore not limited by the size of the in string. The caller is responsible for deleting the storage for the out using string_free(). The callee is not allowed to return a null pointer for an inout, out, or return value.

in parameters

in parameters are passed as const char *.

IDL sample 10.15 IDL interface with a string as the in parameter

```
interface foo
{
   void test(in string s);
};
```

Generated stub

```
class foo {
   ...
   void test(const char *s);
};
```

Client code

```
foo *obj = ...;
const char *s = "This is a Test";
obj->test(s);

// CORBA::String_var can also be used instead of const char *
CORBA::String_var aS = (const char *)"This is a Test";
// NOTE: The cast - (const char *) is done, since ANSI C
// maps string literals to (char *) instead of (const char *).
// If the cast is not done, it would result in memory errors
// when aS is deleted.
obj->test(aS);
```

Server code

```
class foo_impl: public _sk_foo
{
   ...
   // The in parameter is passed as const char * and the callee
   // should not release s.
   void test(const char *s) {
      cout << "Received " << s << endl;
   }
};
```

Case 5

For `inout` sequences and `anys`, assignment or modification of the sequence or `any` may cause deallocation of owned storage before any reallocation occurs, depending upon the state of the boolean release parameter with which the sequence or `any` was constructed.

Case 6

For `out` parameters, the caller allocates a pointer to an array slice, which has all the same dimensions of the original array except the first, and passes the pointer by reference to the callee. The callee sets the pointer to point to a valid instance of the array. For returns, the callee returns a similar pointer. The callee is not allowed to return a null pointer in either case. In both cases, the caller is responsible for releasing the returned storage. To maintain local/remote transparency, the caller must always release the returned storage, regardless of whether the callee is located in the same address space as the caller or is located in a different address space. Following completion of a request, the caller is not allowed to modify any values in the returned storage—to do so, the caller must first copy the returned array instance into a new array instance, then modify the new instance.

Return parameters

Return parameters are passed by value. For arrays, the return type is a pointer to array slice and the caller is responsible to release the return parameter.

IDL sample 10.16 IDL interface with structures as return parameters

```
struct S { long a; short b; };
union U switch(long) { case 1: long a; default: short b; };
typedef octet A[64];

interface foo {
   S test_struct();
   U test_union();
   A test_array();
};
```

Generated stub

```
class foo {
   S test_struct();
   U test_union();
   // The return parameter is A_slice *, since most compilers
   // disallow returning arrays.
   A_slice *test_array();
};
```

Client code

```
foo *obj = ...;
S rS = foo->test_struct();
U rU = foo->test_union();
A_slice *rA = foo->test_array();
...
A_free(rA);
//or using _var
A_var rA = foo->test_array();
```

Server code

```
class foo_impl: public _sk_foo {
   ...
   S test_struct() {
      S r;
      ..
      return r;
      // Or if using _var
      S_var r = new S;
      ...
      return r._retn();
   }
   U test_union() {
      U r;
      ..
      return r;
      // Or if using _var
      U_var r = new U;
      ...
      return r._retn();
   }
   A_slice *test_array() {
      A_slice *r = A_alloc();
      ..
      return r;
      // Or if using _var
      A_var r = A_alloc();
      ...
      return r._retn();
      // NOTE: A null pointer may not be returned.
   }
};
```

The returned value (array slice) should not be modified or used after the value has been returned to caller. If using `_var` variables, the callee can use `_retn` operator on `_var` to return the value. This has the effect of setting the value held by `_var` to NULL after returning the contained value.

Chapter

11

Smart Agent architecture

This chapter describes the Smart Agent (`osagent`), which locates the object implementations that client programs wish to use. This chapter enables you to configure your own ORB domain, connect Smart Agents on different local networks, and migrate objects from one host to another. It includes the following major sections:

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 the `_bind()` method 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.

When a globally scoped object implementation invokes either the `BOA::obj_is_ready()` method or the `BOA::impl_is_ready()` method, the Smart Agent registers the object or implementation so that it can be used by client programs. When an object or

implementation is deactivated, the Smart Agent removes it from the list of available objects. As with client programs, the communication with the Smart Agent is completely transparent to the object implementation. Object scoping is discussed in "Initializing and registering an object" on page 6-2.

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. See "Using point-to-point communications" on page 11-7 for information on how to override the use of broadcast messages.

Locating objects through Agent cooperation

When a Smart Agent is started on more than one 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 will be notified 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 OAD so that 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 *real* (possibly newly) spawned server. The Smart Agent does not know that the real object implementation is not *actually* active within the 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 within which broadcast message can be sent. For instructions on using the `osagent` command, see the *VisiBroker for C++ Installation and Administration Guide*.

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

```
prompt> osagent
```

If you're running Windows NT, you can start the Smart Agent as an NT service through the Services control panel. Alternatively, you can use the following command:

Code sample 11.1 Starting the Smart Agent on a Windows NT system

NT
```
prompt> osagent -C
```

Note The `-C` options allows this command to run in console mode. Otherwise, it is considered an NT service.

Ensuring Agent availability

Starting a Smart Agent on more than one host within the local network allows clients to continue to bind to objects, even if one of the Smart Agents terminates unexpectedly. If a Smart Agent becomes unavailable, all object implementations registered with that Smart Agent will be automatically re-registered with another Smart Agent. If no Smart Agents are running on the local network, object implementations will continue retrying until a new Smart Agent can be contacted.

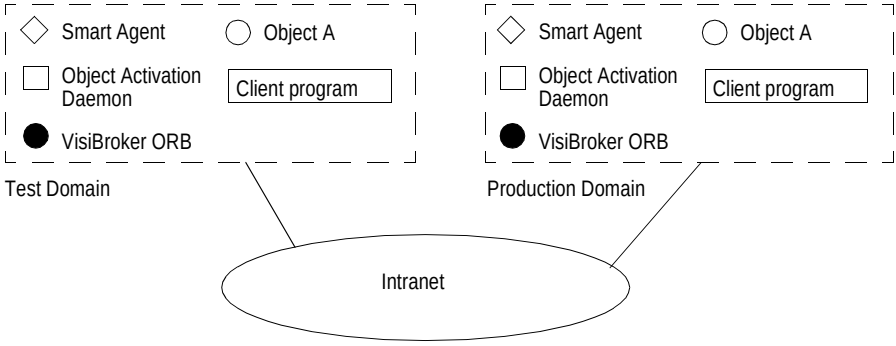
If a Smart Agent terminates, any connections between a client and an object implementation that were established before the Smart Agent terminated will continue without interruption. However, any new `_bind()` requests issued by a client will cause a new Smart Agent to be contacted.

No special coding techniques are required to take advantage of these fault-tolerant features. You only need to make sure a Smart Agent is started on two or more hosts on the local network.

Working within ORB domains

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

Figure 11.1 Running separate ORB domains at the same time



VisiBroker allows you to distinguish between multiple ORB domains on the same network by using a unique UDP port number for the Smart Agents for each domain. By default, the `OSAGENT_PORT` variable is set to 14000. If you wish to use a different port number, check with your system administrator to determine what port numbers are available. To override the default setting, the `OSAGENT_PORT` variable must be set on each host running a Smart Agent, an OAD, object implementations, or client programs assigned to that ORB domain.

Code sample 11.2 Setting the OSAGENT_PORT environment variable for a UNIX system running csh

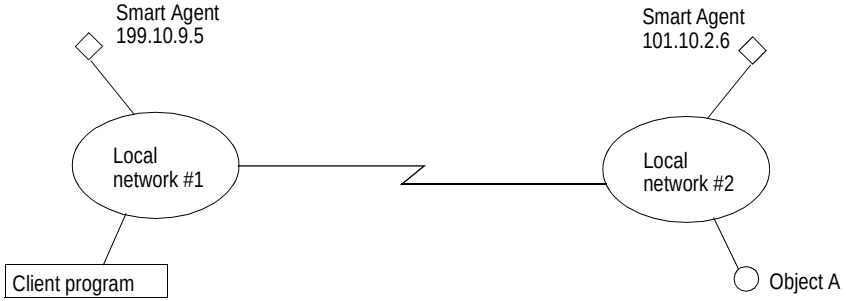
```
prompt> setenv OSAGENT_PORT 5678
prompt> osagent &
prompt> oad &
```

W Setting the `OSAGENT_PORT` environment variable overrides the Windows registry setting.

Connecting Smart Agents on different local networks

If you start multiple Smart Agents on your local network, they will discover each other by using UDP broadcast messages. Your network administrator configures a local network by specifying the scope of broadcast messages using the IP subnet mask. Figure 11.2 shows two local networks, connected by a network link.

Figure 11.2 Two Smart Agent processes and their IP addresses, located on separate, connected local networks



To allow the Smart Agent on one network to contact a Smart Agent on another local network, you must make the IP address of the remote Smart Agent available in a file named `agentaddr`. Code sample 11.3 shows what this file would contain to allow the Smart Agent on local network #1 to connect to the Smart Agent on the other network. The path to this file is specified by the `VBROKER_ADM` environment variable that is set for the Smart Agent process. You can override this file name by setting the `OSAGENT_ADDR_FILE` environment variable.

Code sample 11.3 Content of the agentaddr file for the Smart Agent on network #1

```
101.10.2.6
```

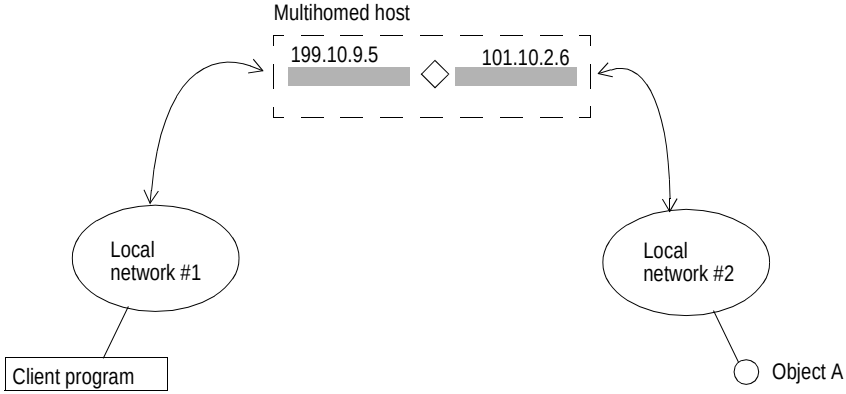
With the appropriate `agentaddr` file, the client program on network #1 could locate and use object implementations on network #2. For more information on environment variables, see the *VisiBroker for C++ Installation and Administration Guide*.

Note If a remote network has multiple Smart Agents running, you should list the IP addresses of all of the Smart Agents on the remote network.

Working with multihomed hosts

When you start the Smart Agent on a host that has more than one IP address (known as a multihomed host) it can provide a powerful mechanism for bridging objects located on separate local networks. All local networks to which the host is connected will be able to communicate with a single Smart Agent, effectively bridging the local networks.

Figure 11.3 One Smart Agent on a multihomed host bridging two local networks



U 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 may explicitly specify interface settings using the **localaddr** file as described in "Specifying interface usage for Smart Agents" on page 11-6.

W 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. Code sample 11.4 shows the sample output from a Smart Agent started with the verbose option on a multihomed host.

Code sample 11.4 Verbose output from a Smart Agent started on a multihomed host

```
Bound to the following interfaces:
Address: 199.10.9.5 Subnet: 255.255.255.0 Broadcast:199.10.9.255
Address: 101.10.2.6 Subnet: 255.255.255.0 Broadcast:101.10.2.255
...
```

As shown in Code sample 11.4, the output shows the address, subnet mask, and broadcast address for each interface in the machine. For UNIX, this output should match the results from the UNIX command `ifconfig -a`.

If you wish to override these settings, you can specify this interface information in the **localaddr** file. See "Specifying interface usage for Smart Agents" below for details.

Specifying interface usage for Smart Agents

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 are treated as comments and ignored. Code sample 11.5 shows the contents of the **localaddr** file for the multihomed host listed above.

Code sample 11.5 Contents of an example localaddr file

```
#entries of format <address> <subnet_mask> <broadcast address>
199.10.9.5 255.255.255.0 199.10.9.255
101.10.2.6 255.255.255.0 101.10.2.255
```

U Though the Smart Agent can automatically configure itself on a multihomed host running UNIX, you can use the **localaddr** file to explicitly specify the interfaces that your host contains. You can display all the available interface values for your 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
```

W The use of the `localaddr` file with multihomed hosts is required for hosts running Windows because the Smart Agent is not able to automatically configure itself. You can obtain the appropriate values for this file by accessing the TCP/IP protocol properties from the Network Control Panel. If your host is running Windows NT, the `ipconfig` command will provide the needed values. You run this command as follows:

```
prompt>ipconfig
```

Output from this command appears similar to the following:

```
Ethernet adapter El59x1:
        IP Address. . . . . . . . . : 199.10.9.5
        Subnet Mask . . . . . . . . : 255.255.255.0
        Default Gateway . . . . . . : 199.10.9.1

Ethernet adapter Elnk32:
        IP Address. . . . . . . . . : 101.10.2.6
        Subnet Mask . . . . . . . . : 255.255.255.0
        Default Gateway . . . . . . : 101.10.2.1
```

Using point-to-point communications

VisiBroker provides you with 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, the ORB will revert to using the broadcast message scheme to locate a Smart Agent.

Specifying a host as a runtime parameter

Code sample 11.6 shows how you can specify the IP address where a Smart Agent is running as a runtime parameter for your client program or object implementation. Since specifying an IP address will cause a point-to-point connection to be established, you can even specify an IP address of a host located outside your local network. This mechanism takes precedence over any other host specification.

Code sample 11.6 Specifying a Smart Agent's IP address as a runtime parameter

```
prompt> server -ORBagentaddr 199.10.9.5 &...
```

or

```
prompt> client -ORBagentaddr 101.10.2.6
```

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.

Figure 11.4 Setting the OSAGENT_ADDR environment variable using the C shell

U

```
prompt> setenv OSAGENT_ADDR 199.10.9.5
prompt> client
```

95 To set the `OSAGENT_ADDR` environment variable on a Windows 95 system, you can add the following line to your **autoexec.bat** file:

```
SET OSAGENT_ADDR=199.10.9.5
```

NT To set the `OSAGENT_ADDR` environment variable on a Windows NT system, you can use the System control panel and add the following definition:

```
OSAGENT_ADDR=199.10.9.5
```

Specifying hosts with the agentaddr file

Your client program or object implementation can use the **agentaddr** file, described in "Connecting Smart Agents on different local networks" on page 11-4, to circumvent the use of UDP broadcast message to locate a Smart Agent. Simply create a file containing the IP addresses or fully qualified hostname 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, the ORB will try each address in the file until a Smart Agent is located. This mechanism has the lowest precedence of all the mechanisms for specifying a host. If this file is not specified, the **VBROKER_ADM/agentaddr** file is used.

Ensuring object availability

You can provide object implementation fault tolerance for objects by starting instances of those objects on multiple hosts. If an implementation becomes unavailable, the ORB will detect the loss of the connection between the client program and the object implementation and will automatically contact the Smart Agent to establish a connection with another instance of the object implementation.

Caution The rebind option, covered in "Enabling rebinds" on page 5-7, must be enabled if the ORB is to attempt to re-connect the client with a replica object implementation. This is the default behavior.

Invoking methods on stateless objects

Your client program can invoke methods 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 register an interceptor for the ORB object. When the connection to an object implementation fails and the ORB reconnects the client to a replica object implementation, the bind interceptor's `rebind_succeeded()` method will be invoked by the ORB. The client must provide an implementation of this method to bring the state of the replica up to date. Interceptors are described in Chapter 16, "Instrumenting and modifying the ORB with 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 the case where a host becomes 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. Object migration can also be used to keep objects available when a host has to be shutdown for hardware or software maintenance.

Note The rebind option, discussed in "Enabling rebinds" on page 5-7, must be enabled to allow a client to reconnect to an object implementation that has migrated to a new host.

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 ORB object. When the connection to the original object is lost and the ORB re-connects the client to the object, the interceptor's `rebind_succeeded()` method will be invoked by the ORB. The client can implement this method to bring the state of the object up to date. Interceptors are described in Chapter 16, "Instrumenting and modifying the ORB with interceptors."

Migrating instantiated objects

If the ORB objects that you wish to migrate were created by a server process instantiating the implementation's C++ 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 will be routed to the object implementation on the new host.

Migrating objects registered with the OAD

If the ORB objects that you wish to migrate are registered with the OAD, you must unregister them with the OAD on the old host. Then, reregister them with the OAD on the new host. Here are the steps:

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 Chapter 6, "Activating objects and implementations," for detailed information on registering and unregistering object implementations.

Chapter

12

Using the tie mechanism: An alternative to inheritance

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

How does the tie mechanism work?

Object implementation classes normally inherit from a skeleton class generated by the `idl2cpp` compiler. The skeleton class, in turn, inherits from `CORBA::Object`. When it is not convenient or possible to change existing classes to inherit from the VisiBroker skeleton class, the *tie* mechanism offers an attractive alternative.

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

Steps for modifying the server to use the tie mechanism

To use the tie mechanism, your server must perform the following actions:

1. Initialize the ORB and BOA.
2. Instantiate an `AccountImpl`.
3. Instantiate a `_tie_Account` and initialize it with the `AccountImpl`.
4. Export the `_tie_Account` object.
5. Wait for requests.

Location of an example program using the tie mechanism

A version of the `Account` example server that uses the tie mechanism can be found in the VisiBroker for C++ distribution under **examples/account** in the directory where the VisiBroker distribution was installed. The only difference from the simple `Account` example introduced in Chapter 4, "Quick start for development with VisiBroker for C++," is in the **account_srvrtie.C** file.

Looking at the _tie template

The `idl2cpp` compiler will automatically generate a `_tie_Account` template class, as shown in Code sample 12.1. The `_tie_Account` class is instantiated by the object server and initialized with an instance of `AccountImpl`. The `_tie_Account` class delegates every operation request it receives to `AccountImpl`, the real implementation class. In this example, the class `AccountImpl` does not inherit from the `_sk_Account` class.

Code sample 12.1 Looking at the _tie template

```
...
template <class T>
class _tie_Account : public Account {
   private:
         short _rel_flag;
         T& _ref;
   public:
         _tie_Account(T& t, const char *obj_name=(char*)NULL, short _r_f=0) :
                     Account(obj_name), _ref(t) {
               _rel_flag=_r_f;
               _object_name(obj_name);
         }
         _tie_Account(           T& t, const char *service_name,
                                       const CORBA::ReferenceData& id,
                                 short _r_f=0)
               :_ref(t) {
                     _rel_flag=r_f;
                     _service(service_name, id);
         }
         ~_tie_Account() { if(_rel_flag) delete &_ref; }
         short rel_flag() { return _rel_flag; }
         void rel_flag(short _r_f) { _rel_flag=_r_f; }
         CORBA::Float balance() {
               return _ref.balance();
         }
};
```

Changing the server to use the _tie_account class

Code sample 12.2 shows the modifications to the **account_srvr.C** file required to use the `_tie_account` class. In the code sample, an `AccountImpl` is created and then attached to the `_tie_Account` object which implements the `Account` interface. This tie object is then declared ready to the ORB.

Code sample 12.2 Example of a server using the _tie class

```
int main(int argc, char* const* argv)
{
      try {
            // Initialize the ORB and BOA
            CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
            CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

            // Create a new account object.
            AccountImpl account;

            // Create the _tie object, initialize with the account object.
            _tie_Account<AccountImpl> tieServer(account, "Jack B. Quick");

            // Export the newly created _tie object.
            boa->obj_is_ready(&tieServer);
            cout << "Account object is ready." << endl;

            // Wait for incoming requests
            boa->impl_is_ready();

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

Changing the object implementation to no longer inherit from _sk_account

The only changes that are made to the `AccountImpl` class are that it no longer inherits from `_sk_account`. Instead, it is a free-standing object that implements the `balance()` method.

Code sample 12.3 Changing the inheritance of the object implementation

```
class AccountImpl
{
      public:
            AccountImpl()
            {
                  // Set a random balance between 0 and 10000
                  _balance = abs(rand()) % 100000 / 100.0;
            }
```

```
            CORBA::Float balance()
            {
                  return _balance;
            }

      protected:
            CORBA::Float _balance;
};
```

Enabling an implementation to inherit from another implementation

The tie mechanism can also be used for *implementation inheritance*, where one implementation inherits from another. Assume that after implementing the `AccountImpl` class, you decide to create a new interface, named `Checking`, that inherits from the `Account` interface. The required changes to the IDL file are shown in IDL sample 12.1.

IDL sample 12.1 Deriving a new interface from Account

```
interface Account {
      float balance();
};

interface Checking : Account {
      void deposit(in float amount);
};
```

When you run the `idl2cpp` compiler, the **account_s.hh** file will contain a `_sk_checking` class, shown in Code sample 12.4.

Code sample 12.4 _sk_checking class generated by the idl2cpp compiler

```
class _sk_Checking : public Checking
{
      public:
            ...
            // The following operations need to be implemented by the server.
            virtual CORBA::Float balance() = 0;
            virtual void deposit(CORBA::Float amount) = 0;
            ...
};
```

If you implement `CheckingImpl` in the usual way, by deriving from the `_sk_Checking` class, the implementation of the `AccountImpl::balance()` method cannot be used. Using multiple inheritance will not work because both interfaces define a `balance()` method. Therefore, we need to inherit `AccountImpl`'s implementation of the `balance()` method, as well as export the new `Checking` implementation. The tie mechanism allows us to do this as shown in Code sample 12.5 and Code sample 12.6.

Note Implementation inheritance can also be accomplished by compiling your IDL with the `_virtual_impl_inh` flag of the `idl2cpp` compiler.

Looking at the CheckingImpl class

If you derive the `CheckingImpl` class from the `AccountImpl` class, you will be able to use the `AccountImpl::balance()` method that you implemented earlier. You must use the `_tie_checking` template class to allow the `CheckingImpl` object to handle client requests for the `checking::deposit()`.

The `CheckingImpl` class inherits from `AccountImpl` and provides an implementation for the new `deposit()` method.

Code sample 12.5 CheckingImpl class

```
class CheckingImpl : public AccountImpl
{
      public:

            void deposit(CORBA::Float amount)
            {
                  _balance += amount;
            }
};
```

Changing the server to enable implementation inheritance

Code sample 12.6 shows the modifications to the **account_srvr.C** file required to use the `_tie_checking` class. Here are the steps the server implements.

1. Initialize the ORB and BOA.
2. Instantiate a `CheckingImpl`.
3. Instantiate a `_tie_checking` class and initialize it with `CheckingImpl`.
4. Export the `_tie_checking` object.
5. Wait for requests.

Code sample 12.6 New main routine that enables implementation inheritance

```
int main(int argc, char* const* argv)
{
      try {
            // Initialize the ORB and BOA
            CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
            CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

            // Create a new checking object.
            CheckingImpl account;

            // Create the checking _tie object, initialized with the
            // checking object.
            _tie_checking<CheckingImpl> tieCheckingServer(account,
                                          "Jack B. Quick");

            // Export the newly created _tie object.
            boa->obj_is_ready(&tieCheckingServer);
            cout << "Checking object is ready." << endl;
```

```
            // Wait for incoming requests
            boa->impl_is_ready();

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

Chapter

13

Using interface repositories

An interface repository (IR) contains descriptions of CORBA object interfaces. The data in an IR is the same as in IDL files—descriptions of modules, interfaces, operations, and parameters—but it is organized for runtime access by clients. A client can browse an interface repository (perhaps serving as an online reference tool for developers) or can look up the interface of any object for which it has a reference (perhaps in preparation for invoking the object with the Dynamic Invocation Interface).

Reading this chapter will enable you to create an interface repository and access it with VisiBroker utilities or with your own code. It includes the following major sections:

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 Chapter 19, “Discovering object instances 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.

Clients that use interface repositories may also use the Dynamic Invocation Interface (DII) described in Chapter 14, "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 parameters—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 another type and is used with the dynamic invocation interface. See "Passing type safely with the Any class" on page 14-10 for information about the `any` type.

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()` method defined for all objects, you must have at least one interface repository server running so the ORB can look up the interface

in the IR. If no interface repository is available, or if the IR that the ORB binds to has not been loaded with an interface definition for the object, `_get_interface()` raises an exception.

Creating and viewing an interface repository with irep

The VisiBroker interface repository server is called `irep`, and is located in the **bin** directory. The `irep` program provides both a command-line and a graphical user interface to an IR. You can register `irep` with the Object Activation Daemon (described in Chapter 6, "Activating objects and implementations") as you would any object implementation. The `oadutil` tool requires the object ID—for example, `IDL:org.omg/CORBA/Repository:1.0` (as opposed to an interface name such as `CORBA::Repository`).

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:

Syntax

```
irep [-console] IRname [file.idl]
```

The `irep` arguments are defined in the following table.

Table 13.1 irep arguments

Argument	Description
`-console`	Directs the program to present a command line interface; otherwise it presents a graphical interface.
`IRname`	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. Use `irep`'s `save` command to save the IR to an IDL file if you want to be able to re-create the IR.

The following example shows how an interface repository named `TestIR` can be created from a file called **Bank.idl**.

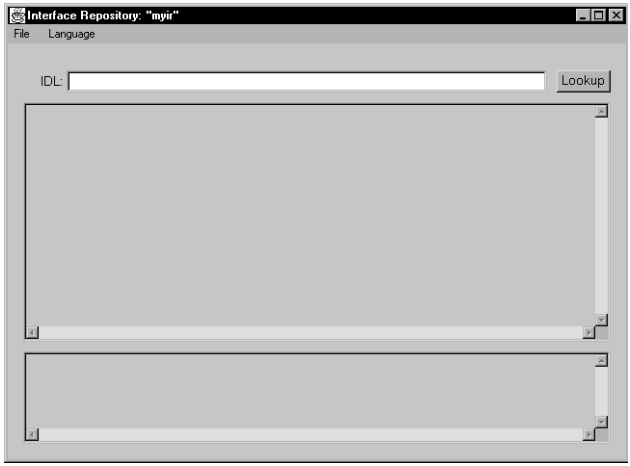
Example

```
irep TestIR Bank.idl
```

Viewing the contents of the interface repository

When the `irep` program starts, it prints the new interface repository object's IOR, and then displays the window shown below. If the `-console` option has been specified, a command-line prompt is displayed instead. To see the definition of an item in the IR, enter its Repository ID or its fully qualified IDL name in the Name text box, then click the Lookup button.

Figure 13.1 irep graphical interface (Windows 95)



The `irep` graphical interface has a File menu and a Language menu. The File menu's items are described in the following table.

Table 13.2 irep File menu

Menu item	Action
Load	Loads the file named by the user in the file browser dialog that appears when this item is chosen.
Lookup	Displays the named item or all items. Enter an item name in the dialog that appears, using either a Repository ID or a fully qualified IDL name.
Save	Writes the contents of the interface repository as the IDL file specified when `irep` was started, unless a Save As command has been issued, in which case the Save As file name is used.
Save As	Writes the contents of the interface repository as the IDL file specified in the file browser dialog which appears when this item is chosen.
Exit	Terminates `irep`.

The Language menu items change the appearance of the displayed data as shown in the following table:

Table 13.3 irep Language menu

Menu item	Action
IDL	Displays IR contents in IDL.
C++	Displays IR contents in C++.
Java	Displays IR contents in Java.

The irep command-line interface provides these commands:

Table 13.4 irep commands

Command		Action
h	(help)	Displays the commands recognized by the program.
l [name]	(lookup)	Displays the named IR element or all elements if no name is supplied. The name can be a fully qualified IDL name, or a Repository ID.
s	(save)	Updates the IDL file specified when irep was started, or the file created earlier with an a command.
a {file.idl}	(save as)	Writes the contents of the interface repository to a new IDL file.
r {file.idl}	(read)	Reads the named IDL file, adding its elements to the IR. If any element in the IDL file matches an element that is already in the IR, the program rejects the file. Matching is based on fully qualified IDL name or Repository ID.
q	(quit)	Exits the program.

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 follows.

Syntax

```
idl2ir [-ir IRname] [-replace] file.idl
```

The idl2ir arguments perform the following actions:

Table 13.5 Arguments for the idl2ir utility

Argument	Interpretation
-ir *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.
-replace	Directs the program to replace interface repository items with matching items in **file.idl**. If -replace is not specified, matching items causes the program to reject everything in the IDL file. If -replace is specified, non-matching items (based on fully qualified IDL names or Repository IDs) are rejected.
file.idl	Specifies the IDL file containing additions or replacements for the interface repository.

The following example shows how the TestIR interface repository would be updated with definitions from the **Bank.idl** file.

Example

```
idl2ir -ir 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.

Understanding the structure of the interface repository

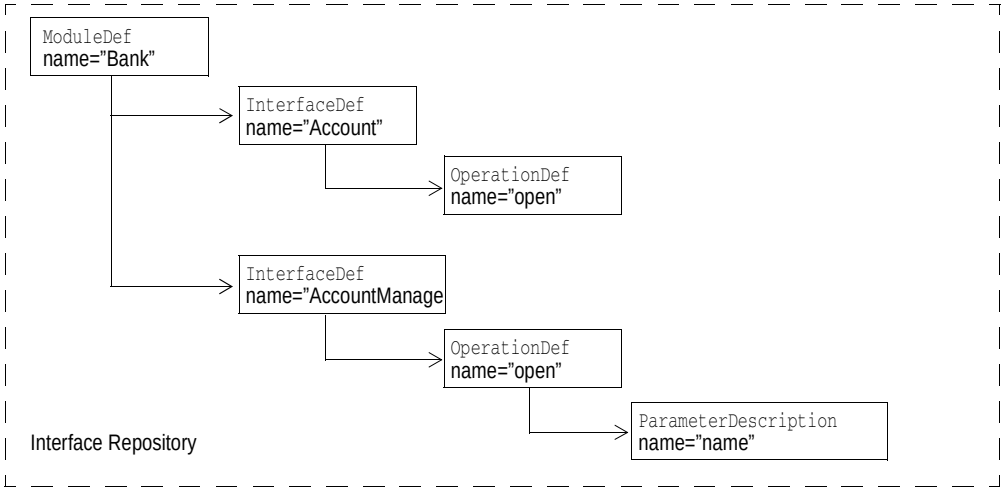
An interface repository organizes the objects it contains into a hierarchy that corresponds to the way interfaces are defined in an IDL specification. Some objects in the interface repository contain other objects, just as an IDL module definition might contain several interface definitions. Consider how the example IDL file shown in IDL sample 13.1 would translate to a hierarchy of objects in an interface repository.

IDL sample 13.1 Bank.idl file

```
// Bank.idl

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

Figure 13.2 Interface repository object hierarchy for the Bank.idl specification



Identifying objects in the interface repository

The following table shows the objects that are provided to identify and classify interface repository objects.

Table 13.6 Objects used to identify and classify interface repository objects

Item	Description
name	A character string that corresponds to the identifier assigned in an IDL specification to a module, interface, operation, and so forth. An identifier is not necessarily unique.
id	A character string that uniquely identifies an IRObject. A RepositoryID contains three components, separated by ":" 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 "/" characters. The first identifier is typically a unique prefix.
def_kind	An enumeration that defines values which represent all the possible types of interface repository objects.

Types of objects that can be stored in the interface repository

Table 13.7 summarizes the objects that can be contained in an interface repository. Most of these objects correspond to IDL syntax elements. A StructDef, for example, contains the same information as an IDL struct declaration, an InterfaceDef contains the same information as an IDL interface declaration, all the way down to a PrimitiveDef which contains the same information as an IDL primitive (boolean, long, and so forth.) declaration.

Table 13.7 Objects that can be stored in the interface repository

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.
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.

Table 13.7 Objects that can be stored in the interface repository (continued)

Object type	Description
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 Table 13.7). Table 13.8 summarizes these widely inherited interfaces.

Table 13.8 Interfaces inherited by many IR objects

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
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()—Describe an object.

Accessing an interface repository

Your client program can use an interface repository's ORB interface to obtain information about the objects it contains. Your client program can bind to the Repository and then invoke the methods shown in Code sample 13.1.

Code sample 13.1 Repository class

```
class CORBA {

      class Repository : public Container {
            ...
            CORBA::Contained_ptr lookup_id(const char * search_id);
            CORBA::PrimitiveDef_ptr get_primitive(CORBA::PrimitiveKind kind);
            CORBA::StringDef_ptr create_string(CORBA::ULong bound);
            CORBA::SequenceDef_ptr create_sequence(CORBA::ULong bound,
                  CORBA::IDLType_ptr element_type);
```

```
            CORBA::ArrayDef_ptr create_array(CORBA::ULong length,
                  CORBA::IDLType_ptr element_type);
            ...
        };
        ...
    };
```

Note A program that uses an interface repository must be compiled with the `-D_VIS_INCLUDE_IR` flag.

Example programs

This section describes a simple IR client that looks up a user-specified interface in an IR and prints its operations (methods) and attributes.

The **PrintIR** program's command-line argument is the fully qualified name of an IDL interface, such as `Bank::Account`. After initializing the ORB and binding to an IR, the program looks up the interface name in the IR. If the repository `lookup()` method fails, the program notifies the user that the IR contains no such interface. If it succeeds, the program narrows the generic object returned by `lookup()` to an IR `InterfaceDef`. From the `InterfaceDef`, the program obtains a `FullInterfaceDescription` object. Using the `FullInterfaceDescription`, the program iterates first through its operations and then through its attributes, printing each operation or attribute's name as it is encountered.

Code sample 13.2 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);
            InterfaceDef_var intDef = CORBA::InterfaceDef::_narrow(contained);
            if (intDef != CORBA::InterfaceDef::_nil()) {
               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;
}
```

Chapter

14

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.

This chapter describes the DII in these sections:

What is the Dynamic Invocation Interface?

The Dynamic Invocation Interface (DII) enables a client program to invoke a method on a CORBA object whose type was unknown at the time the client was written. The DII contrasts with the default static invocation, which requires that the client source code include a compiler-generated stub for each type of CORBA object that the client intends to invoke. In other words, a client that uses static invocation declares in advance the types of objects it will invoke. A client that uses the DII makes no such declaration because its programmer doesn't know what kinds of objects will be invoked. The advantage of the DII is flexibility—it can be used to write generic

clients that can invoke any object, including objects whose interfaces did not exist when the client was compiled. The DII has two disadvantages:

- It is more difficult to program (in essence, your code must do the work of a stub).
- Invocations take longer because more work is done at runtime.

The DII is purely a client interface—static and dynamic invocations are identical from an object implementation's point of view.

You can use the DII to build clients like these:

- Bridges or adapters between script environments and CORBA objects. For example, a script calls your bridge, passing object and method identifiers and parameter values. Your bridge constructs and issues a dynamic request, receives the result, and returns it to the scripting environment. Such a bridge could not use static invocation because its developer could not know in advance what kinds of objects the script environment would want to invoke.
- Generic object testers. For example, a client takes an arbitrary object identifier, looks up its interface in the interface repository (see Chapter 13, "Using interface repositories"), and then invokes each of its methods with artificial argument values. Again, this style of generic tester could not be built with static invocation.

Note Clients *must* pass valid arguments in DII requests. Failure to do so can produce unpredictable results, including server crashes. Although it is possible to dynamically type-check parameter values with the interface repository, it is expensive. For best performance, ensure that the code (for example, script) that invokes a DII-using client can be trusted to pass valid arguments.

Introducing the main DII concepts

The dynamic invocation interface is actually distributed among a handful of CORBA interfaces. Furthermore, the DII frequently offers more than one way to accomplish a task—the trade-off being programming simplicity versus performance in special situations. As a result, DII is one of the more difficult CORBA facilities to grasp. This section is a starting point, a high-level description of the main ideas. Details, including code examples, are provided later in the chapter.

To use the DII you need to understand these concepts, starting from the most general:

- `Request` objects
- `Any` and `Typecode` objects
- Request sending options
- Reply receiving options

Using request objects

A `Request` object represents one invocation of one method on one CORBA object. If you want to invoke two methods on the same CORBA object, or the same method on two different objects, you need two `Request` objects. To invoke a method you first need an object reference representing the CORBA object—the target reference. Using the target reference, you create a `Request`, populate it with arguments, send the

Request, wait for the reply, and obtain the result from the Request. When you have obtained the result from a Request, you free its memory in the usual C++ way.

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 ORBs do not contain type information. By specifying a place holder result type in the Request, you give the ORB the information it needs to properly extract the result from the reply message sent by the target object. Similarly, if the method you are invoking can raise user exceptions, you must add place holder exceptions to the Request before sending it.

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 Anys. 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). A Typecode can be simple or recursive. Simple TypeCode constants for simple IDL types, such as long and Object, are built into the header files produced by the idl2cpp compiler. TypeCodes for IDL constructed types, such as structs and typedefs, have to be constructed. Such TypeCodes can be recursive because the types they describe can be recursive. Consider a struct consisting of a long and a string. The TypeCode for the struct contains a TypeCode for the long and a TypeCode for the string. The idl2cpp compiler will generate TypeCodes for the constructed types in an IDL file if the

compiler is invoked with the -type_code_info option. However, if you are using the DII, you need to obtain TypeCodes at runtime. You can get a TypeCode at runtime from an interface repository (see Chapter 13, "Using interface repositories") or by asking the ORB to create one by invoking ORB::create_struct_tc() or ORB::create_exception_tc().

If you use the _create_request() method, you need to put the Any-encapsulated target method arguments in another special object called an NVList. No matter how you create a Request, its result is encoded as an NVList. Everything said about arguments in this paragraph applies to results as well. NV stands for named value, and an NVList consists of a count and number of items, each of which has a name, a value, and a flag. The name is the argument name, the value is the Any encapsulating the argument, and the flag denotes the argument's IDL mode (for example, in or out).

Options for sending requests

Once you've created and populated a Request with arguments, a result type, and exception types, you send it to the target object. There are several ways to send a Request,

- The simplest is to call the Request::invoke() method, which blocks until the reply message is received.
- More complex, but not blocking, is the Request::send_deferred() method. This is an alternative to using threads for parallelism. For many operating systems the send_deferred() method is more economical than spawning a thread.
- If your motivation for using the send_deferred() method is to invoke multiple target objects in parallel, you can use the ORB object's send_multiple_requests_deferred() method instead. It takes a sequence of Request objects.
- Use the Request::send_oneway() method if, and only if, the target method has been defined in IDL as oneway.
- You can invoke multiple oneway methods in parallel with the ORB's send_multiple_requests_oneway() method.

Options for receiving replies

If you send a Request by calling its invoke() method, there is only one way to get the result—use the Request object's env() method to test for an exception, and if none, extract the NVList from the Request with its result() method. If you used the send_oneway() method then there is no result. If you used the send_deferred() method, you can periodically check for completion by calling the Request's poll_response() method which returns a code indicating whether the reply has been received. If, after polling for a while, you want to block waiting for completion of a deferred send, use the Request's get_response() method.

If you have sent Requests with the send_multiple_requests_deferred() method, you can find out if a particular Request is complete by invoking that Request's get_response() method. To learn when any outstanding Request is complete, use the ORB's get_next_response() method. To do the same thing without risking blocking, use the ORB's poll_next_response() method.

Steps for invoking object operations dynamically

To summarize, here are the steps that a client follows when using the DII,

1. Make sure the `-type_code_info` option is passed to the `idl2cpp` compiler so that type codes are generated for IDL interfaces and types. See the *VisiBroker for C++ Reference* for a complete description of the `idl2cpp` tool.
2. Obtain a generic reference to the target object you wish to use.
3. Create a `Request` object for the target object.
4. Initialize the request parameters and the result to be returned.
5. Invoke the request and wait for the results.
6. Retrieve the results.

Location of example programs for using the DII

Several example programs that illustrate the use of the DII are included in the **examples/acctdii** directory of the VisiBroker distribution. These example programs will be used to illustrate DII concepts in this chapter. 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.
Code sample 14.1 shows how your client program can use the `bind()` method offered by the ORB object to bind to any object by specifying its name. This method returns a generic `CORBA::Object`.

Code sample 14.1 Obtaining a generic object reference

```
...
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 the 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::_request()` method or `Object::_create_request()` method are used to create a `Request` object.

Request class

Code sample 14.2 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. The initialization of the remaining properties is covered in "Creating and initializing a request" on page 14-6.

Code sample 14.2 CORBA::Request class

```
class Request {
   public:
            CORBA::Object_ptr target() const;
            const char* operation() const;
            CORBA::NVList_ptr arguments();
            CORBA::NamedValue_ptr result();
            CORBA::Environment_ptr env();
            void ctx(CORBA::Context_ptr ctx);
            CORBA::Context_ptr ctx() const;

            CORBA::Status invoke();
            CORBA::Status send_oneway();
            CORBA::Status send_deferred();
            CORBA::Status get_response();
            CORBA::Status poll_response();
            ...
      };
};
```

Ways to create and initialize a DII request

Once you have issued a bind to an object and obtained an object reference, you can use one of two methods for creating a `Request` object. Code sample 14.3 shows the methods offered by the `CORBA::Object` class.

Code sample 14.3 Three methods for creating a Request object

```
class Object {
   ...
   CORBA::Request_ptr _request(Identifier operation);
   CORBA::Status _create_request(
      CORBA::Context_ptr            ctx,
      const char *      operation,
      CORBA::NVList_ptr arg_list,
      CORBA::NamedValue_ptr         result,
      CORBA::Request_ptr         request,
      CORBA::Flags         req_flags);
   CORBA::Status                    _create_request(
      CORBA::Context_ptr            ctx,
      const char *      operation,
      CORBA::NVList_ptr arg_list,
      CORBA::NamedValue_ptr         result,
      CORBA::ExceptionList_ptr eList,
      CORBA::ContextList_ptr ctxList,
      CORBA::Request_out         request,
      CORBA::Flags         req_flags);
   ...
};
```

Using the create_request method

You can use the `_create_request()` method to create a `Request` object, initialize the `Context`, the operation name, the argument list to be passed, and the result. Optionally, you can set the `ContextList` for the request, which corresponds to the attributes defined in the request's IDL. The request parameter points to the `Request` object that was created for this operation.

Using the _request method

Code sample 14.4 shows the use of the `_request()` method to create a `Request` object, specifying only the operation name. After creating `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 `CORBA::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. If your client program specifies the ORBbackdii option when the ORB is initialized, then the client will have ownership of the memory associated with these items. See Appendix A, "Using command-line options," of the *VisiBroker for C++ Reference* for complete information on the ORBbackdii option.

Code sample 14.4 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.

Code sample 14.5 Context class

```
class Context {
      public:
            const char *context_name() const;
            CORBA::Context_ptr parent();
            CORBA::Status create_child(
                  const char       *name,
                  CORBA::Context_ptr&);
            CORBA::Status set_one_value(
                  const char       *name,
                  const CORBA::Any&);
            CORBA::Status set_values(CORBA::NVList_ptr);
```

```
            CORBA::Status delete_values(const char *name);
            CORBA::Status get_values(
                    const char        *start_scope,
                    CORBA::Flags,
                    const char        *name,
                    CORBA::NVList_ptr&) const;
};
```

Setting arguments for the request

The arguments for a Request are represented with a `NVList` object, which stores name-value pairs as `NamedValue` objects. You can use the `arguments()` method to obtain a pointer to this list. This pointer can then be used to set the names and values of each of the arguments.

Note Always initialize the arguments before sending a `Request`. Failure to do so will result in marshalling errors and may even cause the server to abort.

Implementing a list of arguments with the NVList

This class implements a list of `NamedValue` objects that represent the arguments for a method invocation. Methods are provided for adding, removing, and querying the objects in the list.

Code sample 14.6 NVList class

```
class NVList {
      public:
            ...
            CORBA::Long count() const;
            CORBA::NamedValue_ptr add(Flags);
            CORBA::NamedValue_ptr add_item(
                        const char *name,
                        CORBA::Flags flags);
            CORBA::NamedValue_ptr add_value(
                        const char *name,
                        const CORBA::Any *any,
                        CORBA::Flags flags);
            CORBA::NamedValue_ptr add_item_consume(
                        char *name,
                        CORBA::Flags flags);
            CORBA::NamedValue_ptr add_value_consume(
                        char *name,
                        CORBA::Any *any,
                        CORBA::Flags flags);
            CORBA::NamedValue_ptr item(CORBA::Long index);
            CORBA::Status remove(CORBA::Long index);
            ...
};
```

Setting input and output arguments with the NamedValue Class

This class implements a name-value pair that represents both input and output arguments for a method invocation request. The `NamedValue` class is also used to

represent the result of a request that is returned to the client program. The `name` property is simply a character string and the `value` property is represented by an `Any` class.

Code sample 14.7 NamedValue class

```
class NamedValue {
      public:
            const char *name() const;
            CORBA::Any *value() const;
            CORBA::Flags flags() const;
};
```

The following table describes the methods in the `NamedValue` class.

Table 14.1 NamedValue methods

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 "Passing type safely with the Any class" on page 14-10.
`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 ORB should make a copy of the argument and leave the caller's memory intact. Flags are —`ARG_IN` —`ARG_OUT` —`ARG_INOUT` —`IN_COPY_VALUE`

Passing type safely with the Any class

This class is used to hold an IDL-specified type so that it may be passed in a type-safe manner. Objects of this class have a pointer to a `TypeCode` that defines the contained object's type and a pointer to the contained object. Methods are provided to construct, copy, and release an object as well as initialize and query the object's value and type. In addition, streaming operators are provided to write the object to a stream.

Code sample 14.8 Any class

```
class Any
{
      public:
            Any();
            Any(const CORBA::Any&);
            Any(TypeCode_ptr tc, void *value,
                  CORBA::Boolean release=0);
            ~Any();

            Any& operator=(const CORBA::Any&);
            // Overloaded operators for all data types
            void operator<<=(CORBA::Short);
```

```
        void operator<<=(CORBA::UShort);
        void operator<<=(CORBA::Long);
        void operator<<=(CORBA::ULong);
        ...
        CORBA::TypeCode_ptr type();
        const void *value() const;
        static CORBA::Any_ptr _nil();
        static CORBA::Any_ptr _duplicate(CORBA::Any *ptr);
        static      void   _release(CORBA::Any *ptr);

        // Streaming operators to write Anys to stdout, etc.
        VISostream& operator<<(VISostream& strm, const CORBA::Any& any);
        VISostream& operator<<(VISostream& strm, const CORBA::Any_ptr any);
        VISistream& operator>>(VISistream& strm, CORBA::Any& any);
        VISistream& operator>>(VISistream& strm, CORBA::Any_ptr any);
        ...
}
```

Representing argument or attribute types with the TypeCode class

This class is used by the Interface Repository and the IDL compiler to represent the type of arguments or attributes. `TypeCode` objects are also used in a `Request` to specify an argument's type, in conjunction with the `Any` class. `TypeCode` objects have a kind and parameter list property.

The following table shows the kinds and parameters for the `TypeCode` objects.

Table 14.2 TypeCode kinds and parameters

Kind	Parameter list
tk_null	None
tk_void	None
tk_short	None
tk_long	None
tk_longlong	None
tk_ushort	None
tk_ulong	None
tk_ulonglong	None
tk_float	None
tk_double	None
tk_longdouble	None
tk_boolean	None
tk_char	None
tk_wchar	None
tk_wstring	None
tk_octet	None
tk_any	None
tk_TypeCode	None

Table 14.2 TypeCode kinds and parameters (continued)

Kind	Parameter list
tk_Principal	None
tk_objref	interface_id
tk_struct	struct-name, {member[1], TypeCode[1]}, {member[n], TypeCode[n]}
tk_union	union-name, switch TypeCode,{label-value[1], member-name[1], TypeCode[1]}, {label[1]-value[n], member-name[n], TypeCode[n]}
tk_enum	{enum-name, enum-id[1], enum-id[2], ... enum-id[n]}
tk_string	maxlen-integer
tk_sequence	TypeCode, maxlen
tk_array	TypeCode, length

The following is a list of the TypeCode constants for IDL data types. All of the TypeCode constants have a data type of TypeCode_ptr.

- _tc_null
- _tc_void
- _tc_short
- _tc_long
- _tc_longlong
- _tc_ushort
- _tc_ulong
- _tc_ulonglong
- _tc_float
- _tc_double
- _tc_longdouble
- _tc_boolean
- _tc_char
- _tc_wchar
- _tc_wstring
- _tc_octet
- _tc_Any
- _tc_TypeCode
- _tc_Principal
- _tc_Object
- _tc_string
- _tc_NamedValue

Sending DII requests and receiving results

The Request class, shown in Code sample 14.2 on page 14-6, 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 `result()` method returns a pointer to a `NamedValue` object that represents the return value.

Code sample 14.9 Sending a request with invoke()

```
try {
      req->invoke();
      CORBA::Environment_ptr env = req->env();
      if (env->exception())
            cout << "Exception occurred" << endl;
      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;
}

// Print out the results
cout << "The balance in " << name << "'s account is $";
cout << acct_balance << "." << endl;
```

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. Code sample 14.10 shows how these methods are used.

Code sample 14.10 Using the send_deferred() and poll_response() methods to send a deferred DII request

```
...
// Send the request
try {
      req->send_deferred();
      cout << "Send deferred call is made..." << endl;
} catch(const CORBA::Exception& excep) {
      cout << "Error while sending request" << endl;
      cout << excep << endl;
}

// Poll for response
try {
      while (!req->poll_response())
```

```
        {
              cout << "  Waiting for response..." << endl;
              sleep(1000);  // Wait one second between polls
        }
  } catch(const CORBA::Exception& excep) {
        cout << "Failure while polling for response" << endl;
        cout << excep << endl;
  }

  try {
        req->get_response();
        CORBA::Environment_ptr env = req->env();
        if (env->exception())
              cout << "Exception occurred" << endl;
        else {
              // Get the return value;
              acct_balance = *(CORBA::Float *)resultAny->value();
        }
  ...
```

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 `RequestSeq`, defined in the `CORBA::ORB` class and shown in Code sample 14.12. A sequence of requests can be sent using the ORB methods `send_multiple_requests_oneway()` or `send_multiple_requests_deferred()`. If the sequence of requests is sent as oneway requests, no response is expected from the server to any of the requests.
Code sample 14.11 shows how two requests are created and then used to create a sequence of requests. The sequence is then sent using the `send_multiple_requests_deferred()` method.

Code sample 14.11 Sending multiple deferred requests with 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 ORB method `poll_next_response()` can be used to determine if a response has been received from the server. This method returns 1 if there is at least one response available. This method returns 0 if there are no responses available.

The ORB method `get_next_response()` can be used to receive a response. If no response is available, this method will block until a response is received. If you do not wish your client program to block, use the `poll_next_response()` method to first determine when a response is available and then use the `get_next_response()` method to receive the result.

Code sample 14.12 ORB methods for sending multiple requests and receiving the results

```
class CORBA {
      class ORB {
            ...
```

```
            typedefsequence <Request_ptr> RequestSeq;

            Status send_multiple_requests_oneway(const RequestSeq &);
            Status 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 Chapter 13, "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.

The example

- 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.

Code sample 14.13 Using the IR and the DII

```
// 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;
  }
```

Chapter

15

Dynamically creating object implementations

This chapter describes how object servers can dynamically create object implementations at run time to service client requests. It includes the following major sections:

What is the Dynamic Skeleton Interface?

The Dynamic Skeleton Interface (DSI) provides a mechanism for creating an object implementation that does not inherit from a generated skeleton interface. Normally, an object implementation is derived from a skeleton class generated by the `idl2cpp` compiler. The DSI allows an object to register itself with the ORB, receive operation requests from a client, process the requests, and return the results to the client without inheriting from a skeleton class generated by the `idl2cpp` compiler.

Note From the perspective of a client program, an object implemented with the DSI behaves just like any other ORB object. Clients do not need to provide any special handling to communicate with an object implementation that uses the DSI.

The ORB presents client operation requests to a DSI object implementation by using the object's `invoke()` method and passing a `ServerRequest` object. The object implementation is responsible for determining the operation being requested, interpreting the arguments associated with the request, invoking the appropriate internal method or methods to fulfill the request, and returning the appropriate values.

Implementing objects with the DSI requires more manual programming activity than using the normal language mapping provided by object skeletons. Nevertheless, an object implemented with the DSI can be very useful in providing inter-protocol bridging.

Steps for creating object implementations dynamically

To create object implementations dynamically using the DSI, follow these steps:

1. Design your object implementation so that it is derived from the `CORBA::DynamicImplementation` abstract class instead of deriving your object implementation from a skeleton object.
2. Declare and implement the `invoke()` method, which the ORB will use to dispatch client requests to your object.
3. Register your object implementation with the BOA in the normal way, using the `BOA::obj_is_ready()` method.
4. Define the `_VIS_INCLUDE_DSI` flag when compiling your program.

Location of an example program for using the DSI

An example program that illustrates the use of the DSI is included in the **examples/dsi** directory of the VisiBroker distribution. This example is used to illustrate DSI concepts in this chapter. The example uses the **bank.idl** file, shown in IDL sample 15.1. Compile this example with the `_VIS_INCLUDE_DSI` flag defined for your compiler.

IDL sample 15.1 bank.idl file used in the DSI example

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

DynamicImplementation class

To use the DSI, object implementations should be derived from the `DynamicImplementation` base class shown in Code sample 15.1. This class offers several constructors and the `invoke()` method, which you must implement.

Code sample 15.1 Portion of the DynamicImplementation base class

```
class CORBA::DynamicImplementation : public CORBA::Object {
   ...
   virtual void invoke(CORBA::ServerRequest_ptr request) = 0;
   ...
   protected:
   CORBA::DynamicImplementation(const char *interface_name,
         const char *object_name =                    NULL,
         const char *repository_id = NULL);

   CORBA::DynamicImplementation( const InterfaceNameSequence& interfaces,
         const RepositoryIdSequence& ids,
         const char *object_name=NULL);

   virtual ~CORBA::DynamicImplementation();
   ...
};
```

Example of designing objects for dynamic requests

Code sample 15.2 shows the declaration of the `DSIAccount` object that is to be implemented with the DSI. It is derived from the `DynamicImplementation` class, which declares the `invoke()` method. The ORB will call the `invoke()` method to pass client operation requests to the implementation in the form of `ServerRequest` objects.

Code sample 15.2 DSI:Account class from the dynamic_account_srvr.C file

```
class DSIAccount : public CORBA::DynamicImplementation
{
   private:
         CORBA::Float _balance;

         void invoke(CORBA::ServerRequest_ptr req);
         CORBA::Float balance();

   public:
         DSIAccount(float balance, const char *object_name=NULL);
         ~DSIAccount() {}
};
```

The `DSI:Account` class constructs itself as shown in Code sample 15.3.

Code sample 15.3 Construction of the DSI:Account class

```
DSIAccount::DSIAccount(float balance, const char *object_name):
   CORBA_DynamicImplementation("Bank::Account", object_name,
                               "IDL:Bank/Account:1.0") {
...
}
```

Code sample 15.4 shows the declaration of the DSIAccountManager object that is to be implemented with the DSI. It is derived from the DynamicImplementation class, which declares the invoke() method. The ORB will call the invoke() method to pass client operation requests to the implementation in the form of ServerRequest objects.

Code sample 15.4 DSI:AccountManager class from the dynamic_account_srvr.C file

```
class DSIAccountManager : public CORBA::DynamicImplementation
{
   private:
      void invoke(CORBA::ServerRequest_ptr req);
      DSIAccount* open(const char* name);
   public:
         DSIAccountManager(const char *object_name);
         ~DSIAccountManager() {}
};
```

The DSI:AccountManager class constructs itself as shown in Code sample 15.5.

Code sample 15.5 Construction of the DSI:AccountManager class

```
DSIAccountManager::DSIAccountManager(const char *object_name):
   CORBA_DynamicImplementation("Bank::AccountManager", object_name,
                               "IDL:Bank/AccountManager:1.0") {
...
}
```

Looking at the ServerRequest class

A ServerRequest object is passed as a parameter to an object implementation's invoke() method. The ServerRequest object represents the operation request and provides methods for obtaining the name of the requested operation, the parameter list, and the context. It also provides methods for setting the result to be returned to the caller and for reflecting exceptions.

Code sample 15.6 ServerRequest class

```
class CORBA::ServerRequest
{
   public:
   const char* op_name() const { return _operation; }
   CORBA::Context_ptr ctx() {
      ...
   }
   void params(CORBA::NVList_ptr);
   void result(CORBA::Any_ptr);
   void exception(CORBA::Any_ptr exception);
   ...
};
```

Note All arguments passed into the params(), results(), or exception() methods become owned by the ORB. The memory for these arguments will be released by the ORB—you should not release them.

Implementing the account object

The DSIAccount object in our example offers only one method, so the processing done by its 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 DSIAccount 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::result() method.

Code sample 15.7 DSIAccount::invoke() method

```
void DSIAccount::invoke(CORBA::ServerRequest_ptr req)
{
   cout << "DSIAccount::invoke(" << req->op_name() << ")"
            << endl;

   // ensure that the operation name is correct
   if (strcmp(req->op_name(), "balance") != 0) {
         throw CORBA::BAD_OPERATION();
   }

   // According to CORBA V2.0, 18.4.2:
   // Implementations must always set the ServerRequest's params,
   // so set here to an empty NVList
   CORBA::NVList_var params = new CORBA::NVList();
   req->params(params);

   // no parameters, so simply invoke the balance operation
   CORBA::Float account_balance = balance();
   CORBA::Any_ptr result = new CORBA::Any();
   result->operator<<= account_balance;
   req->result(result);
}
```

Implementing the AccountManager object

Like the DSIAccount object, the DSIAccountManager offers only one method. However, the AccountManager.open() method does accept an account name parameter. This makes the processing done by the invoke() method a little more complicated.
Code sample 15.8 shows the implementation of the DSIAccountManager::invoke() method.

The method first checks to see that the requested operation has the name "open." If the name does not match, a BAD_OPERATION exception is raised. If the DSIAccountManager 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.

Processing input parameters

Here are the steps the AccountManager::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::param() 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 Chapter 14, "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::params() method is invoked to obtain the values of all of the parameters in the list.

Note After invoking the params() method, the NVList will be owned by the ORB. This means that if an object implementation modifies an ARG_INOUT parameter in the NVList, the change will automatically be apparent to the ORB. This NVList should not be released by the caller.

An alternative to constructing the NVList for the input arguments is to use the CORBA::ORB::create_operation_list() method. This method accepts an OperationDef and returns an NVList object, completely initialized with all the necessary Any objects. The appropriate OperationDef object may be obtained from the interface repository, described in Chapter 13, "Using interface repositories."

Setting the return value

After invoking the ServerRequest::params() 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 DSIAccount object, which is returned to the caller by invoking the ServerRequest::result() method.

Code sample 15.8 DSIAccountManager::invoke() method

```
void DSIAccountManager::invoke(CORBA::ServerRequest_ptr req)
{
cout << "DSIAccountManager::invoke(" <<req->op_name() <<")" <<endl;
  // ensure that the operation name is correct
  if (strcmp(req->op_name(), "open") != 0) {
    throw CORBA::BAD_OPERATION();
  }
```

```
    // create an empty parameter list
    CORBA::NVList_ptr params = new CORBA::NVList();

    // create an Any for the account name parameter
    CORBA::Any nameAny;
    // set the Any's type to be a string
    nameAny <<= (const char*) NULL;
    // add the Any to the parameter list
    params->add_value("name", nameAny, CORBA::ARG_IN);
    // obtain the parameter values from the request
    req->params(params);

    // pull out the account name from the Any.
    // Note that the revisions to the C++ mapping
    // no longer give return ownership to the caller
    // (with back. compat. the caller _is_ responsible
    // for releasing the return values)
    CORBA::NamedValue_var name_nv;
    CORBA::Any_var name_any;
    if (_back_compat_dsi) {
        name_nv = params->item(0);
        name_any = name_nv->value();
    } else {


        name_nv = CORBA::NamedValue::_duplicate(params->item(0));
        name_any = CORBA::Any::_duplicate(name_nv->value());
    }
    char *name_string = (char *) NULL;
    if (! (name_any->operator>>= name_string )) {
      throw CORBA::BAD_PARAM();
    }
    CORBA::String_var name(name_string);

    // make the call to "open"
    DSIAccount* account = open(name);

    // put the result into the request
    CORBA::Any_ptr result = new CORBA::Any();
    result <<= account;
    req->result(result);
}
```

Looking at the main routine

The implementation of the `main` routine, shown in Code sample 15.9, is almost identical to the original example introduced in Chapter 4, “Quick start for development with VisiBroker for C++.”

Code sample 15.9 main routine implementation

```
int main(int argc, char* const* argv)
{
   try {
            // Initialize the ORB and BOA
            CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
            CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

            // Create a new account object.
            DSIAccountManager manager("Bank Manager");

            // Export the newly created object.
            boa->obj_is_ready(&manager);
            cout << "Account object is ready." << endl;

            // Wait for incoming requests
            boa->impl_is_ready();


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

Chapter

16

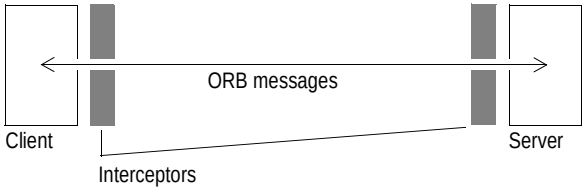
Instrumenting and modifying the ORB with interceptors

An interceptor is a powerful feature that exposes under-the-cover communication between clients and servers. Interceptors are extensions that the VisiBroker ORB invokes at specific points in its processing. Using interceptors, you can view communications between clients and servers, and modify these communications if you wish, effectively altering the behavior of the VisiBroker ORB.

This chapter describes the common uses of interceptors, walks through an interceptor example, provides a complete reference to the interceptor API, and describes some advanced features such as interceptor factories and chaining interceptors. This chapter includes these major topics:

What are interceptors?

Figure 16.1 How interceptors work



At its simplest usage, the interceptor is useful for tracing through code. Because you can see the messages being sent between clients and servers, you can determine exactly how the ORB is processing requests.

If you are building a more sophisticated application such as a monitoring tool or security layer, interceptors give you the information and control you need to enable these lower level applications. For example, you could develop an application that monitors the activity of various servers and performs load balancing. The OMG defines two types of interceptors—message interceptors and request interceptors—and VisiBroker supports both.

You can write interceptors to perform the following actions:

- Impose access control on objects and methods, granting or denying access based on the identity of the requestor.
- Write log data for analysis of performance and usage patterns.
- Add to, or alter the data passed between clients and servers, for example, add transaction information or encrypt a credit card number.
- Trace the routing of invocations from clients to servers. Potentially, this tracing could extend to all the servers where a client invocation is forwarded.
- Dynamically select the server that receives a client invocation, possibly choosing it based on load information.
- Provide an alternative location service that is invoked when the VisiBroker `bind()` method fails to find a server.

The power of interceptors can be applied in numerous ways, giving you the flexibility to extend the VisiBroker ORB to best suit your purposes. Interceptors can be used with both the BOA and the Secure BOA.

Note Use object wrappers, described in Chapter 22, “Using object wrappers,” if you want to intercept an operation request before it is marshalled on the client-side or if you want to intercept an operation request before it is process on the server-side.

Prerequisites for using interceptors

At the most basic level, interceptors are useful informational tools for tracing your application. Inprise provides you with an example interceptor that you can use for this purpose. The explanation of this example interceptor, in "Example interceptor" on page 16-4, should provide you with enough information to write your own interceptors that view low-level information.

If you would like to modify the example interceptor, or produce your own interceptor, Inprise recommends that you be proficient with system-level programming. Interceptors work below the client and server application code, and therefore lack the comprehensive error checking of higher-level code. As a result, the effects of errors can be both serious and difficult to diagnose. Some interceptor methods—such as those that change the messages sent between clients and servers by the ORB—provide more potential for damage than others. Prudent developers will begin with simple interceptors that log data; and then, after those are working, tackle interceptors that alter ORB, client, or server behavior.

To write an interceptor, Visigenic recommends that you be familiar with Chapter 12, "General Inter-ORB Protocol," of the CORBA 2.0 specification—especially IORs (interoperable object references) and GIOP message formats (including headers and marshal buffers, and request, reply, and locate messages). The specification is available from the OMG web site at http://www.omg.org/corba/corbaiiop.htm.

Interceptor components

Interceptor developers derive classes from one or more of the following base interceptor API classes which are defined and implemented by the VisiBroker ORB:

- **VISBindInterceptor**. The VisiBroker ORB invokes the methods in this class as it locates objects either through `bind()` or `rebind()`.
- **VISClientInterceptor**. The VisiBroker ORB invokes the methods in this class as it prepares and sends object invocation requests, and as it receives replies to these requests.
- **VISServerInterceptor**. The VisiBroker ORB invokes the methods in this class as it processes ordinary (object invocation) requests and locate requests.

Some interceptor API methods have only constant arguments, and are therefore just notices from the ORB that something has happened. Other interceptor API methods have arguments that can be changed or that return a result; and therefore give you the opportunity to alter data at the ORB level, including message contents and IORs. Use caution with methods that alter ORB communications.

Only some interceptors are allowed to throw exceptions, as shown in Table 16.5, Table 16.6, and Table 16.7.

Example interceptor

VisiBroker includes an example interceptor that you can use to trace through your applications. The example interceptor uses all of the interceptor API methods listed in "Interceptor API reference" on page 16-6 so that you can see how these methods are used, and when they are invoked.

The code sample in this section only shows the code for the server-side interceptor; however, the code for the client-side interceptor is very similar. To see the complete code for the example interceptor, review the **vinter.h** header file in the **/examples/inter** directory. Each interceptor API method for the `VISServerInterceptor` class is bold in the following code excerpt for quick reference.

Code sample 16.1 Implementation for server-side interceptors

```
class SampleServerInterceptor : public VISServerInterceptor {
public:
   virtual IOP::IOR *locate(CORBA::ULong /* req_id */,
         CORBA_OctetSequence */* object_key */,
         VISClosure&/* closure */) {
     interceptPrint("ServINT", "locate");
     return 0;
   }
   // Called if locate succeeded.
   virtual void locate_succeeded(CORBA::ULong /* req_id */,
         VISClosure&/* closure */) {
     interceptPrint("ServINT", "locate_succeeded");
   }
   // Called if locate forwarded. If ior is changed, that IOR
   // is forwarded to client.
   virtual void locate_forwarded(CORBA::ULong /* req_id */,
         IOP::IOR&/* forward_ior */,
         VISClosure&/* closure */) {
     interceptPrint("ServINT", "locate_forwarded");
   }
   // Called if locate failed. If "non null" ior is returned that
   // IOR is forwarded to client
   virtual IOP::IOR *locate_failed(CORBA::ULong /* req_id */,
         CORBA_OctetSequence */*object_key  */,
         VISClosure&/* closure */) {
     interceptPrint("ServINT", "locate_failed");
     return 0;
   }
   // Called when a request message is received from client
   // If return value is "not null", that buffer is passed onto
   // target. It also possible to change the hdr and forward the
   // request to a different operation or to a different Object.
   virtual CORBA_MarshalInBuffer *receive_request(
         GIOP::RequestHeader& /* hdr */,
         CORBA::Object *& target/* target */,
         CORBA_MarshalInBuffer */* buf */,
         VISClosure&/* closure */) {
     interceptPrint("ServINT", "receive_request");
     return 0;
```

```
    }
    // Function called when the reply header is being prepared
    // NOTE: request_failed can be called
    virtual void prepare_reply(
          const GIOP::RequestHeader&/* hdr */,
          GIOP::ReplyHeader&/* reply hdr */,
          CORBA::Object_ptr/* target */,
          VISClosure&/* closure */) {
      interceptPrint("ServINT", "prepare_reply");
    }
    // Function called when reply is being sent.
    // If return value is "not null", that value is sent to client.
    // The env passed will contain any user exceptions, if they were
    // thrown.
    virtual CORBA_MarshalOutBuffer *send_reply(
          const GIOP::RequestHeader&/* reqHdr */,
          const GIOP::ReplyHeader& /* hdr */,
          CORBA::Object_ptr/* target */,
          CORBA_MarshalOutBuffer*/* buf */,
          CORBA::Environment&/* env */,
          VISClosure&/* closure */) {
      interceptPrint("ServINT", "send_reply");
      return 0;
    }
    // Function called if send failed.
    virtual void send_reply_failed(
          const GIOP::RequestHeader&/* reqHdr */,
          const GIOP::ReplyHeader& /* hdr */,
          CORBA::Object_ptr/* target */,
          const CORBA_Exception&/* excep */,
          VISClosure&/* closure */) {
      interceptPrint("ServINT", "send_reply_failed");
    }
    // Function called if reply sent successfully
    virtual void request_completed(const GIOP::RequestHeader& /*reqHdr*/,
          CORBA::Object_ptr /* target */,
               VISClosure&/* closure */) {
      interceptPrint("ServINT", "request_completed");
    }
    // Function called when exception occurred
    virtual void exception_occurred(const GIOP::RequestHeader& hdr,
          CORBA::Environment& env,
          VISClosure&/* closure */) {
      interceptPrint("ServINT", "exception_occurred");
    }
    // Called when the connection is forcibly being shutdown by
    // the orb (Adapter) since the connection limit has been
    // reached
    virtual void shutdown(VISServerInterceptor::ShutdownReason reason) {
      interceptPrint("ServINT", "shutdown");
    }
};
```

The results of executing the complete example interceptor (**vinter.h**) are shown in the following table. The execution by the client and server is listed in sequence. Results prefaced with `BindINT` refer to methods in the `VISBindInterceptor` class, those prefaced with `ClntINT` refer to methods in the `VISClientInterceptor` class, and those prefaced with `ServINT` refer to methods in the `VISServerInterceptor` class.

Table 16.1 Results of executing the complete example interceptor

Client	Server

	`Installing Debugging Interceptors`
	`BindINT: bind`
	`BindINT: bind_failed`
	`Account object is ready.`

`Installing Debugging Interceptors BindINT: bind`	
	`ServINT: locate` `ServINT: locate_forwarded`

`BindINT: bind_succeeded`
`ClntINT: prepare_request`
`ClntINT: send_request_succeeded`	

	`ServINT: receive_request`
	`ServINT: prepare_reply` `ServINT: request_completed`
`ClntINT: receive_reply` `The balance in Default Name's account is $168.38.`	

By observing the output from the interceptors, you can see much of what VisiBroker is doing behind the scenes. In this example, the `account_server` attempts to bind to the Object Activation Daemon (OAD). Since the OAD is not running, the `bind()` call fails and the server proceeds. The client binds to the account object, and then calls the `balance()` method. This request is received by the server, processed, and results are returned to the client. The client prints the results.

As shown through the example code and results, the interceptors for both the client and server are installed when the process starts. Information about registering an interceptor is covered in “Registering interceptors with the VisiBroker ORB” on page 16-14.

Interceptor API reference

The following sections list the methods for each interceptor class (`VISBindInterceptor`, `VISClientInterceptor`, and `VISServerInterceptor`), and provide short descriptions of their functionality. The methods are located in **include/vinter.h** for further reference.

Modifying arguments in the interceptor methods

Some of the methods described in the following sections allow modification of their arguments, as indicated in the table descriptions of each method. However, modification of these arguments should be done with caution.

Specifically, when modifying either the `RequestHeader` or the `ReplyHeader`, a service context can be added or removed. Also for `RequestHeader`, the `Principal` can be modified.

Additionally, modifications to `MarshalInBuffer` and `MarshalOutBuffer` must be carefully managed to return these buffers in the manner expected by the ORB. The easiest way to modify these buffers involves changing them in place—that is, not allocating a new buffer. Also, it is required that the offsets to these buffers remain at the same place after the interceptor is called—that is, any reads must be rewound before returning from the interceptor.

VISBindInterceptor class

By writing a bind interceptor, you can supply an alternative object location service that is invoked unequivocally or only when the default service fails. You can also use a bind interceptor to record how long binding takes.

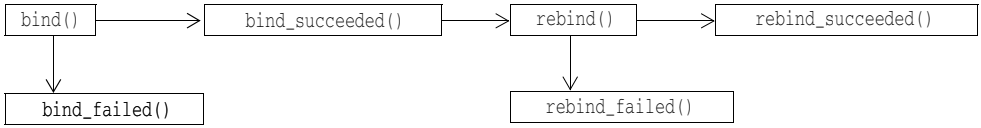
Table 16.2 VISBindInterceptor class

Method	Behavior
`bind()`	The ORB invokes this method when it is about to bind to a server object. The method can supply an IOR for binding to the object—effectively determining the object to which the interceptor will bind. You can also use this method to start a timer for measuring binding time. See "Passing information between your interceptors" on page 16-15 for more details. Modification of the IOR is allowed.
`bind_succeeded()`	The ORB invokes this method when it has completed the `bind()` successfully. It is a notification that an interceptor could use, for example, to stop a timer that its `bind()` method started. This method is for notification only.
`bind_failed()`	The ORB invokes this method if its bind attempt did not succeed. This method can be used to note the failure or attempt to bind the object by its own means, returning an IOR for the ORB to use. Modification of the IOR is allowed.
`rebind()`	If rebinding is enabled, the ORB invokes this method when it is about to rebind. Typically, the `rebind()` method is called when an object reference is no longer valid. For example, the `rebind()` method is called when a call is made on an object that is no longer available. The method can override the default `rebind()` method behavior, returning an IOR for the ORB to use. Modification of the IOR is allowed.
`rebind_succeeded()`	The ORB invokes this method if a rebind succeeds—it is a notice. This method is for notification only.
`rebind_failed()`	The ORB invokes this method if a rebind failed. This method can do its own rebind and return an IOR for a server object. Modification of the IOR is allowed.

Note If more than one bind interceptor is installed, the `bind()` and `rebind()` methods are called in the order they are installed in the interceptor chain. All others are called in reverse order. See "Using more than one interceptor" on page 16-12 for more information about chaining interceptors.

Figure 16.2 shows the order in which the VISBindInterceptor methods are called.

Figure 16.2 Order in which VISBindInterceptor methods are called.



All VISBindInterceptor methods are allowed to throw exceptions. Doing so causes the ORB to call the interceptor exception_occurred() method. Note that the exception_occurred() method is always called on the same interceptor that throws the exception. Exceptions thrown in the exception_occurred() method will override the original exception.

VISClientInterceptor class

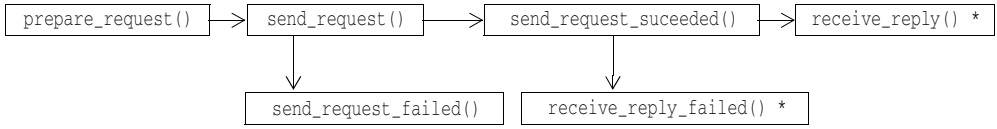
A client interceptor's methods, if implemented, are invoked at key points in the ORB's processing of client requests.

Table 16.3 VISClientInterceptor class

Method	Behavior
prepare_request()	The ORB invokes this method when it is preparing a request. The method can alter or add to the request's header. Modification of the RequestHeader is allowed.
send_request()	The ORB invokes this method before it sends a request. The method can change the request's marshal buffer, perhaps encrypting a parameter. The method might also start a timer or log request data. Modification of the MarshalOutBuffer is allowed.
send_request_failed()	The ORB invokes this method if it was unable to send the request, likely due to a communication problem. The nature of the error is described in the Exception parameter. This method is for notification only.
send_request_succeeded()	The ORB invokes this method if it was able to send the request. This method is for notification only.
receive_reply()	The ORB invokes this method when it receives a reply to a request. The method can change the marshal buffer received from the server. For example, receive_reply() could decrypt the marshal buffer. Modification of the MarshalInBuffer is allowed.
receive_reply_failed()	The ORB invokes this method if the connection is lost during receipt of a reply, or if a reply is not received within the time specified by the client in the bind() method options. This method is for notification only.
exception_occurred()	See "Using more than one interceptor" on page 16-12.

Figure 16.3 shows the order in which the VISClientInterceptor methods are called.

Figure 16.3 Order in which VISClientInterceptor methods are called



Note In Figure 16.3, the asterisks mark the methods that are not relevant for oneway calls.

All `VISClientInterceptor` methods are allowed to throw exceptions. Doing so causes the ORB to call the interceptor `exception_occurred()` method. Note that the `exception_occurred()` method is always called on the same interceptor that throws the exception. Exceptions thrown in the `exception_occurred()` method will override the original exception.

VISServerInterceptor class

Server interceptor methods, if implemented, are invoked at key points in the ORB's processing of object invocation and locate requests.

Table 16.4 VISServerInterceptor class

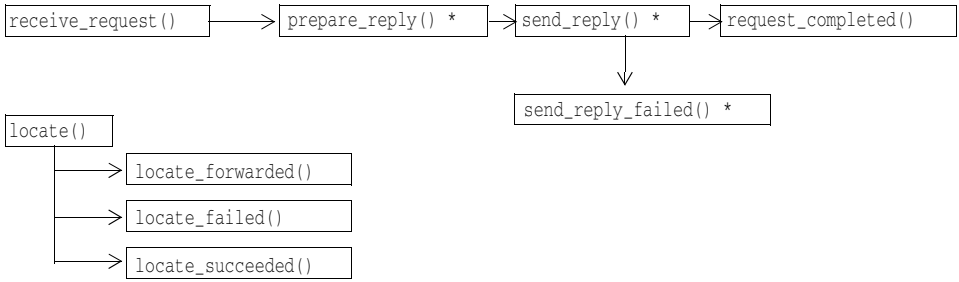
Method	Behavior
`receive_request()`	The ORB invokes this method when it receives a request message for the server. The method can alter the incoming message—for example, it can decrypt arguments. It can also redirect the message to a different operation on the same object or to a different object. Modification of the `MarshalInBuffer` and `RequestHeader` is allowed.
`prepare_reply()`	The ORB invokes this method when it is about to prepare a reply message. The method can change the reply header.
`send_reply()`	The ORB invokes this method when it is about to send a reply. This is a notice, and can be used, for example, to stop a timer set when the corresponding request arrived. Modification of the `MarshalOutBuffer` is allowed.
`send_reply_failed()`	The ORB invokes this method if it was unable to send a reply, likely due to a communication problem. This method is for notification only.
`request_completed()`	The ORB invokes this method if it sent the reply successfully. This method is for notification only.
`locate()`	In addition to invocation messages, servers may receive locate request messages generated by a client's ORB. For example, servers may receive locate requests as the result of a client's `bind()` call. A server's ORB invokes this method when one arrives and returns an IOR to direct the server ORB to return a status of `OBJECT_FORWARD` in the reply. In other words, the method can force locate requests to be forwarded to a server of its choosing. Modification of the IOR is allowed.
`locate_succeeded()`	The ORB invokes this method when a locate message succeeds, in other words when the ORB is about to reply to a locate request with a status of `OBJECT_HERE`. VisiBroker does not normally use this interceptor point due to some optimizations. This method is for notification only.

Table 16.4 VISServerInterceptor class (continued)

Method	Behavior
`locate_failed()`	The ORB invokes this method if the server no longer implements the object. By returning an IOR, the method can force the request to another server, as noted above in the `locate()` description. This method is for notification only.
`locate_forwarded()`	The ORB invokes this method just before it sends a locate forwarded reply. By overriding this method, an interceptor can learn that an interceptor on the chain (possibly itself) has caused a locate message to be forwarded. In other words, if any interceptor's `locate()` method changes the IOR, the ORB invokes this method. Modification of the IOR is allowed.
`exception_occurred()`	See "Using more than one interceptor" on page 16-12.
`shutdown()`	The ORB invokes this method when the connection terminates. This could be caused by a client terminating or the connection limit being reached. This method is for notification only.

Figure 16.4 shows the order in which `VISServerInterceptor` methods are called.

Figure 16.4 Order in which VISServerInterceptor methods are called



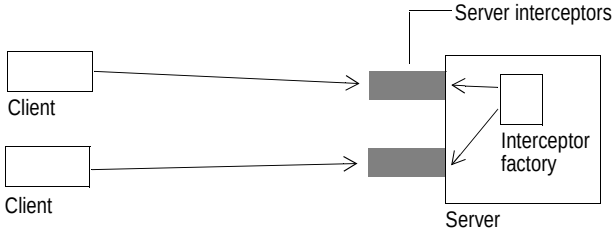
Note In Figure 16.4, the asterisks mark the methods that are not relevant for oneway calls.

Only the `receive_request()` and `send_reply()` methods are allowed to throw exceptions. Any exceptions that are thrown in the server, including those thrown in the implementation code, cause the `exception_occurred()` method to fire. All exceptions are propagated back to the client, which is distinct from the bind or client interceptors. Note that the `exception_occurred()` method is always called on the same interceptor that throws the exception. Exceptions thrown in the `exception_occurred()` method will override the original exception.

Creating interceptor instances with factories

You create an instance of a client or server interceptor indirectly using *factories*. A factory is a class that uses the `create()` method to establish instances of interceptors. For server interceptors, instances of the interceptor are created per client connection as shown by the following diagram:

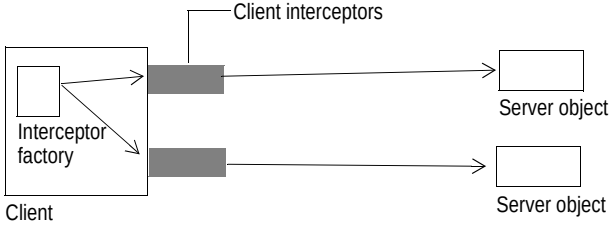
Figure 16.5 Creating interceptor instances with factories



Note You can set up your code so that only one server interceptor is created to process all client connections. The example interceptor uses only one server interceptor, as shown by the code in the following Code sample 16.2.

For client interceptors, instances of the interceptor are created per server object requested by the client, regardless of the location of the server object. As with server interceptors, you can create one client interceptor to handle all server object requests. The following figure shows a client factory producing instances of the client interceptor for two server objects.

Figure 16.6 Producing instances of the client interceptor for two server objects



Note No factories are required for bind interceptors.

The following code excerpt shows how the example interceptor creates a factory for the server interceptor. The code is very similar for the client interceptor factory.

Code sample 16.2 Factory creation in the example interceptor

```
class SampleServerInterceptorFactory : public VISServerInterceptorFactory {
private:
  static SampleServerInterceptorFactory _instance;
public:
  SampleServerInterceptorFactory() {
    _server = (SampleServerInterceptor*) NULL;
  }
  static SampleServerInterceptorFactory *instance() {
    return &_instance;
  }
  // Returns a server interceptor for every connection to the server
  VISServerInterceptor *create(int fd, const IOP::TaggedProfile& p) {
    if (_server == (SampleServerInterceptor*) NULL) {
      _server = new SampleServerInterceptor();
    }
```

```
    return VISServerInterceptor::_duplicate(_server);
  }
private:
  SampleServerInterceptor *_server;
};
```

As shown, you provide a server factory implementation by deriving from the `VISServerInterceptorFactory` class, and override the `create()` method to return an instance of your derivation of the `VISServerInterceptor` class. For client factories, you provide a factory implementation by deriving from the `VISClientInterceptorFactory` class, and override the `create()` method to return an instance of your derivation of the `VISClientInterceptor` class.

An additional step is taken to implement the factory when you register the interceptor with the VisiBroker ORB, as shown in the following code excerpt:

Code sample 16.3 Implementing the factory

```
SampleServerInterceptorFactory *serverFac =
   new SampleServerInterceptorFactory();
```

The process for registering interceptors is fully described in "Registering interceptors with the VisiBroker ORB" on page 16-14. You do not need to release or free interceptors—the VisiBroker ORB cleans them up when their process, object, or connection goes away.

Using more than one interceptor

To use more than one interceptor at a time—either per object (for client interceptors), or per connection (for server interceptors)—you can *chain* interceptors. The VisiBroker ORB chains interceptor instances in the order that their classes are registered (see "Registering interceptors with the VisiBroker ORB" on page 16-14). The ORB invokes an interceptor method by passing down the chain, invoking the method on one interceptor after another. For example, if three client interceptors are registered, and all three implement the `VISClientInterceptor::send_request()` method, the VisiBroker ORB invokes the three `send_request()` methods in the order the interceptors were registered.

Note Even if you are only using one interceptor, you must still use a chaining statement.

The following code excerpt shows how a chain of server interceptors is created. Chains of bind or client interceptors are created in a similar fashion.

```
VISChainServerInterceptorFactory::add(serverFac);
```

If more than one interceptor has been installed through the interceptor chain, and an exception is raised in one of the interceptors, the remaining interceptors will not fire. The `exception_occurred()` method will be called on the interceptors in the chain as indicated by the firing orders shown in Table 16.5, Table 16.6 on page 16-13, and Table 16.7 on page 16-13.

For example, suppose there are five interceptors on a chain. In the process of invoking these interceptors' `prepare_request()` methods, number three raises a CORBA exception. The VisiBroker ORB will not invoke the `prepare_request()`

methods of interceptors four and five, and instead will invoke the `exception_occurred()` method in interceptors one, two, and three, and raise an exception to the client. The `exception_occurred()` methods of interceptors one and two should then clean up any state that depends on execution of the request; for example, if a timer has been started, it should be reset.

The following tables show the order in which methods fire for chained interceptors, as well as the `exception_occurred()` behavior for these methods. When "Forward" is used, this means methods fire for interceptor A, then interceptor B, and so forth. When "Backward" is used, this means the reverse is true—methods fire for interceptor C, then interceptor B, then interceptor A.

Table 16.5 Order methods fire and exception behavior for VISBindInterceptor methods

Method	Order	exception_occurred() behavior
`bind()`	Forward	Called for all bind interceptors fired in the chain.
`bind_succeeded()`	Backward	Called for every interceptor in the chain.
`bind_failed()`	Backward	Called for every interceptor in the chain.
`rebind()`	Forward	Called for all rebind interceptors fired in the chain.
`rebind_succeeded()`	Backward	Called for every interceptor in the chain.
`rebind_failed()`	Backward	Called for every interceptor in the chain.

Table 16.6 Order methods fire and exception behavior for VISClientInterceptor methods

Method	Order	exception_occurred() behavior
`prepare_request()`	Forward	Called for all client interceptors fired so far in the chain.
`send_request()`	Forward	Called for every interceptor in the chain.
`send_request_succeeded()`	Backward	Called for every interceptor in the chain.
`send_request_failed()`	Backward	Called for all interceptors that have not fired.
`receive_reply()`	Backward	Called for all interceptors that have not fired.
`receive_reply_failed()`	Backward	Called for all interceptors that have not fired.

Table 16.7 Order methods fire and exception behavior for VISServerInterceptor methods

Method	Order	exception_occurred() behavior
`locate()`	Forward	Exceptions are ignored.
`locate_succeeded()`	Backward	Exceptions are ignored.
`locate_failed()`	Backward	Exceptions are ignored.
`locate_forwarded()`	Backward	Exceptions are ignored.
`receive_request()`	Forward	Called for all server interceptors that have fired up to this point. The exception is returned to the client. Interceptors that have not fired will get returned to the chain upon the next request.
`prepare_reply()`	Backward	Exceptions are ignored.
`send_reply()`	Backward	Called for every interceptor in the chain.
`send_reply_failed()`	Backward	Exceptions are ignored.

Table 16.7 Order methods fire and exception behavior for VISServerInterceptor methods (continued)

Method	Order	exception_occurred() behavior
request_completed()	Backward	Exceptions are ignored.
shutdown()	Forward	Exceptions are ignored.

Registering interceptors with the VisiBroker ORB

To register an interceptor with the VisiBroker ORB, you must create a class that initializes an instance of the interceptor. When the process invokes the ORB_init() or BOA_init() methods, the ORB checks for the existence of an initializer. If an initializer exists, the ORB ensures that the appropriate initialization methods are called. For installing the interceptors, the initialization method creates instances of the interceptor factories and adds them to the chain. Code sample 16.4 shows the server-relevant parts of the initialization file.

Code sample 16.4 Registering interceptors with the VisiBroker ORB

```
#include "vinit.h"
#include "vinter.h"

static CORBA::Boolean _debugInited = 0;
// Class initialized when the ORB is initialized
class SampleInitializer : public VISInit {
  static SampleInitializer _instance;
  void init(int &argc, char *const *argv) {
    if (!_debugInited) {
      cout << "Installing Sample Interceptors" << endl;
      // Install server interceptor
      SampleServerInterceptorFactory *serverFac =
            new SampleServerInterceptorFactory();
      // Associate factory with server chain so ORB can
      // call create() on the factory
      VISChainServerIntercepFactory::add(serverFac);
      // Install client interceptor
      SampleClientInterceptorFactory *clientFac =
            new SampleClientInterceptorFactory();
      // Associate factory with client chain so ORB can
      // call create() on the factory
      VISChainClientIntercepFactory::add(clientFac);
      // Install bind interceptor
      // Factory not needed since process uses chain for its lifetime
      VISChainBindInterceptor::add(new SampleBindInterceptor());
      _debugInited = 1;
    }
  }
  // Called when VisiBroker process calls ORB_init()
  void ORB_init(int &argc, char *const *argv, CORBA::ORB_ptr orb) {
    init(argc, argv);
  }
```

```
  // Called when VisiBroker process calls BOA_init()
  void BOA_init(int &argc, char *const *argv, CORBA::BOA_ptr boa) {
    init(argc, argv);
  }
};
```

Note The `VISInit` class provides a general purpose initialization method that can also be used for non-interceptor initialization.

Passing information between your interceptors

The interceptors allow the user to propagate information between the methods using the `VISClosure` class. A reference to a `VISClosure` object is available from every method. At any time, a user can set the `data` or `managedData` fields for the object, and the remaining methods will get the new object. `VISClosure` objects are specific to one interceptor in a given chain—they are not propagated between different interceptors in a chain. The `VISClosure` class appears as follows:

Code sample 16.5 VISClosure() class

```
struct VISClosure {
   CORBA::ULong            id;
   void                    *data;
   VISClosureData          *managedData;
};
```

The `data` member of the `VISClosure` class requires the user to perform memory management. The `managedData` member allows the ORB to handle releasing the memory. To make use of this, you need to derive your data implementation from the `VISClosureData` class. The destructor will automatically get called when appropriate. See the Timer interceptor example in the **examples/inter** directory for an example of using the `VISClosure` class.

Chapter

17

Customizing remote object invocations using smart stubs

A VisiBroker stub is a C++ object in a client that represents a remote object and its methods. You can customize a stub to intervene in every client invocation to a remote object. This capability is useful for a variety of purposes including load balancing, logging, and caching. The developer of smart stubs decides what actions a smart stub takes when it is invoked.

This chapter includes these major topics:

What is a smart stub?

A VisiBroker stub is a C++ object in a client that represents a remote object and its methods. By default, the VisiBroker IDL compiler generates stubs for remote objects. When a client invokes a stub method, the stub invokes the corresponding method on the remote object.

You can customize stubs to behave differently than the stubs that VisiBroker generates by default. Because a smart stub intervenes in every client invocation of a remote object, it can be used for a variety of purposes, including

- **Caching**: The smart stub attempts to return method results from a cache rather than invoking the remote operation.

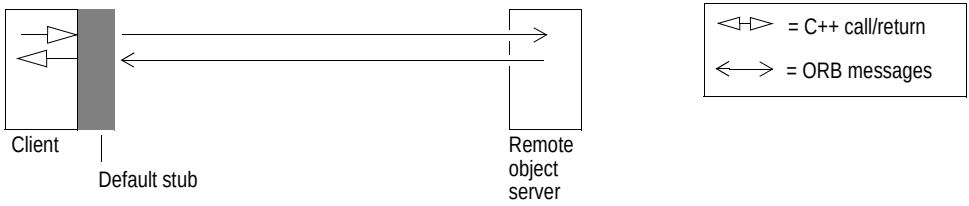
- **Logging**: The smart stub records the method that has been invoked, then invokes the method on the remote object.
- **Load Balancing**: The smart stub invokes the operation on the least-loaded of several equivalent remote objects.

Smart stubs are not perfectly transparent to client developers. Although remote object invocation is identical with default stubs and smart stubs, smart stubs must be explicitly instantiated whereas default stubs are created automatically.

How do smart stubs work?

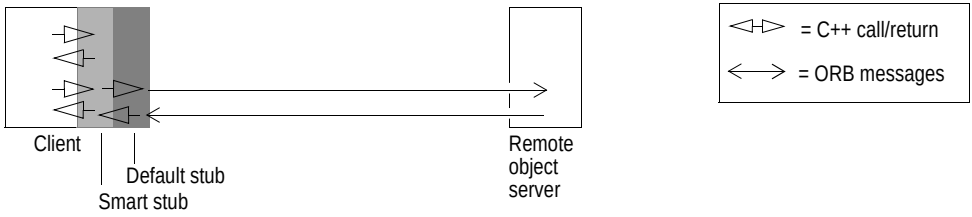
To invoke a method on a remote object, a client makes a local C++ method invocation. This local implementation is a proxy for the remote implementation. The stub class is generated by the IDL compiler when the remote object's interface definition is compiled. The stub takes care of many details required to invoke a remote object: marshaling arguments, invoking the remote object, and unmarshaling results. In other words, by representing a remote object, a stub makes a remote object appear to be local to the invoking client.

Figure 17.1 Client invokes a default stub which invokes a remote object



A smart stub is interposed between the client and the default stub. The client invokes the smart stub. Then the smart stub can, but does not have to, invoke the default stub to invoke the remote object. A common use for a smart stub is to improve performance by substituting caching for some remote invocations. An example of a caching smart stub is provided later in this chapter.

Figure 17.2 Client invokes a smart stub which may invoke the default stub



Notice that smart stubs (and default stubs, for that matter) are invisible to object servers—stubs are purely client-side facilities.

Prerequisites for writing and using smart stubs

To write a smart stub you need to know how to write a derived class that overrides some methods of its base class. A smart stub is a subclass of the default stub generated by the IDL compiler. Smart stubs need to be thread-safe if their clients are multithreaded and client threads can invoke the same remote operation concurrently.

To use a smart stub in a client, you need only include the stub implementation and add initialization code that installs it. Binding to remote objects and invoking their methods is identical whether a smart stub or a default stub represents the remote object.

Smart stub components

To develop a smart stub, you inherit one class and implement another.

default stub	This class is generated by the IDL compiler, and the smart stub is derived from it. The default stub has the same name as is defined in the IDL for the remote object.
smart stub	This is the class that implements the smart stub, inheriting from the default stub. If the smart stub overrides a method in the remote object, it has the same method name as the original method in the remote object. Any remote object methods that are not overridden are handled by the default stub. In addition to defining a subset of the remote object's methods, the smart stub class must provide a constructor and a factory method. The ORB calls the factory method to create an instance of the smart stub.

To use a smart stub, you include its class in your client, and install the class by creating an instance of the `VISSmartStub` class. The `VISSmartStub` class is provided with the VisiBroker ORB and declared in the **vsstub.h** file. You install the smart stub after initializing the ORB but before binding to an object represented by the smart stub.

Example caching smart stub

To see how a smart stub can be used to speed performance by caching results, consider the following IDL, which defines a `Dictionary` object. A `Dictionary` object is a collection of word/definition pairs, in which the word is called a key.

IDL sample 17.1 IDL for the Dictionary interface

```
// Smart Stub example
interface Dictionary {
  typedef string KeyType;
  typedef string DefinitionType;
  typedef sequence<string> KeyListType;
```

```
    struct EntryType {
      KeyType key;
      DefinitionType defn;
    };
    typedef sequence<EntryType> EntryListType;

    // List all keys in the dictionary
    KeyListType allKeys();

    // Lookup a definition in the dictionary based on a key
    boolean lookup(in KeyType key, out DefinitionType defn);
  };
```

The `Dictionary` interface defines two methods:

- The `lookup()` method takes a word (`key`) and returns either the word's definition (`defn`) and 1, or, if the word is not in the dictionary, the method returns 0.
- The `allKeys()` method returns a list of the words in the dictionary.

In this example, the code for the smart stub implementation and the client that uses it are in the same file. A smart stub can also be implemented in a separate file so different clients can include it.

The smart stub class `SmartDictionary` is derived from the default stub class `Dictionary` that is generated by compiling the `Dictionary` IDL file. Making the smart stub a subclass of the default stub ensures that any methods not overridden by the smart stub are handled by the default stub.

The first method in the `SmartDictionary` class implements the smart stub constructor, which is invoked by the factory method. To register itself properly with the ORB, the constructor is required to invoke the `CORBA::Object`'s constructor first. It may then perform any other initialization—in this case, the constructor sets the new smart stub's cache length to 0.

The second method, `smartFactory()`, is the factory method. The factory method creates an instance of the smart stub object. The method pointer is registered with the ORB when the instance of `VISSmartStub` is created as shown in Code sample 17.2. The ORB invokes the factory method when it needs to create a stub object—that is, when the client code binds to an object of the type represented by the smart stub. The factory method may be given any name, but the client must pass the method to the constructor of the `VISSmartStub` class. Although the factory method's name can be different, the rest of its signature must be the same as the example's `smartFactory()` method.

The third method, `lookup()`, overrides the corresponding method in the `Dictionary` default stub class. There are no VisiBroker-imposed requirements on overriding methods, though developers should be cautious about changing the semantics of a remote object method.

From the client's perspective, the `SmartDictionary` class behaves like the remote object except that, for some key values, it is faster. Notice, however, that the client will *not* see a change in the definition of a word in the remote dictionary if the old definition has been cached by the smart stub. A real-world smart stub might implement a more sophisticated cache that is informed of remote object updates or retires old cache entries.

The `SmartDictionary` class does not override the `allKeys()` method. Therefore, client invocations of that method are handled by the `Dictionary` class, the default stub. When you write a smart stub, you can override the default stub class methods that you want to implement yourself, leaving the others to the default stub.

Code sample 17.1 SmartDictionary smart stub implementation

```
// dict_clnt.C
#include <iostream.h>
#include "dict_c.hh"
#include "vsstub.h"
#include "vport.h"

// Derive from the generated stub
class SmartDictionary : public Dictionary
{
   public:
  // Must call the constructor for CORBA::Object
  SmartDictionary(const char *object_name=NULL)
    : CORBA_Object(object_name, 1) {
      _cache.length(0);
  }
  // Factory method used so ORB will create this local
  // stub instead of remote object stub
  static CORBA::Object *_smartFactory() {
    return new SmartDictionary();
  }

  // Method we wish to be "smart" about
  CORBA::Boolean lookup(const char* key, DefinitionType& defn) {
    // Lookup key in cache
    for (int i=0; i < _cache.length(); i++) {
      if (!vstricmp(key, _cache[i].key)) {
   defn = CORBA::strdup(_cache[i].defn);
   cout << "<cache>  ";
   return 1;
      }
    }

    // Not found in cache so must look up remote
    cout << "<remote> ";
    if (Dictionary::lookup(key, defn)) {
      // Add definition for this key in local cache
      _cache.length(_cache.length() + 1);
      _cache[_cache.length()-1].key = (const char *) key;
      _cache[_cache.length()-1].defn = (const char *) defn;
      return 1;
    }
    return 0;
  }
  private:
    EntryListType _cache;
};
```

A client that uses a smart stub is almost identical to one that uses a default stub. The client initializes the ORB as usual. The only code that is required to use a smart stub is the creation of an instance of the `VISSmartStub` class.

The constructor takes three arguments:

- The ORB (obtained from the earlier ORB initialization).
- The `SmartDictionary`'s interface repository ID, which identifies the smart stub class to be installed.
- The `SmartDictionary` class's factory method, which the ORB invokes to create an instance of the smart stub when the client binds to a `Dictionary` object.

By creating a `VISSmartStub` instance, the ORB installs the smart stub for the `Dictionary` interface's repository ID. If the `VISSmartStub` instance is deleted, the smart stub is removed from the ORB. The body of the client would be identical if it did not install the `SmartDictionary` smart stub.

Code sample 17.2 Client that uses the SmartDictionary smart stub

```
int main(int argc, char* const* argv)
{
  CORBA::Boolean done = 0;
  try {
    // Initialize the ORB.
    CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);

    // Create local stub instance before binding to an object
    VISSmartStub sstub(orb, SmartDictionary::_desc()->repository_id(),
             SmartDictionary::_smartFactory);

    // Bind returns smart dictionary object
    Dictionary_var dict = Dictionary::_bind();
    CORBA::Object_ptr obj = (CORBA::Object_ptr) dict;

    // Print all keys that are in the dictionary
    cout << "Dictionary has these keys:" << endl;
    Dictionary::KeyListType_var keys = dict->allKeys();
    for (int i=0; i < keys->length(); i++) {
      cout << "  " << (*keys)[i] << endl;
    }
    cout << "\n";

    while (!done) {
         char key[256];
         char *defn;
         cout << "Enter key to lookup definition ('quit' to exit): ";
         cin >> key;
         cout << "\n";
         if (!vstricmp("quit", key)) {
            done = 1;
            break;
         }
// Call to lookup will call smart dictionary object, which may,
      // in turn, make a call to a remote object
```

```
      if (dict->lookup(key, defn))
            cout << key << " - " << defn << endl;
      else
            cout << "Not Found" << endl;
      cout << "\n";
    }
  } catch(const CORBA::Exception& e) {
    cerr << e << endl;
    return(1);
  }
  return(0);
}
```

Chapter

18

Enabling object persistence using the Object Database Activator

This chapter describes how VisiBroker may be integrated with a databases to provide support for persistent objects, and will enable you to use the ObjectStore database, or another database product, to implement persistent objects. It includes the following major sections:

What is object persistence?

Unlike C++ objects, most distributed objects are persistent. This means they must maintain their state long after the program that creates them terminates. To be

persistent, an object's state must be stored in a non-volatile datastore—for example, a database or file system.

VisiBroker allows you to implement objects that persist beyond the life of the implementation server by saving the objects to a text file, database, or object database. When an object implementation is activated (using the `Activator` class as described in Chapter 6, "Activating objects and implementations"), the object is initialized from the storage device and when the object is deactivated, its state is saved to the storage device.

What is the difference between tie and ptie?

The `idl2cpp` compiler may be used to generate `tie` or persistent `tie` template classes. The `tie` template class, as discussed in Chapter 12, "Using the tie mechanism: An alternative to inheritance," provides object servers with a *delegator implementation* class that inherits from `CORBA::Object`. The delegator implementation does not provide any semantics of its own. It simply delegates every request it receives to the real implementation class, which can be implemented separately.

The persistent tie template class (`ptie`) provides a delegator implementation as well—except its delegator implementation is designed to work with databases. The delegator implementation does not provide any semantics of its own. It simply delegates every client request that is received to the real implementation object whose state is stored in the database.

Prerequisites for using the Object Database Activator

To write the Object Database Activator, Inprise recommends that you be familiar with how servers are activated (see "Automatically activating servers with the Object Activation Daemon" on page 6-5), how the `tie` class is used (see Chapter 12, "Using the tie mechanism: An alternative to inheritance"), and how interceptors work (see Chapter 16, "Instrumenting and modifying the ORB with interceptors").

Steps for implementing a persistent object

You can use the following steps to create a server that implements persistent objects. Each of these steps will be described using an example program that uses the ObjectStore database, but the steps are similar if other object databases are used.

1. Create or obtain the IDL file that contains the interfaces you wish to implement as persistent objects.
2. Use the `idl2cpp` compiler to generate persistent `tie` template classes from the IDL file.

3. Use the persistent tie template classes generated to derive your own implementation template classes, providing the methods and member data that your object database requires.
4. Create the persistent object implementation classes that meet the requirements of your storage device.
5. If your object database requires you to use transaction semantics, define an interceptor with `pre_method()` and `post_method()` implementations to handle the *begin* and *commit* semantics. Interceptors are described in Chapter 16, "Instrumenting and modifying the ORB with interceptors."
6. Implement `Activator` objects for your server's objects, if required. The use of the `Activator` class is described in the section "Deferring object activation until a client request" on page 6-12.
7. Write the server's `main` routine, incorporating the method invocations that your object database requires.

Note The following example is written specifically for the Object Design ObjectStore database. The example may require extensive modifications to be used with other database products.

Example of using the Object Database Activator

The following example is a version of the `Bank` example that uses the persistent tie mechanism with the ObjectStore database. The following files make up the example client and server programs:

- **bank.idl**—Defines the interface for the `AccountManager` and `Account` object.
- **server.C**—Implements the server `main()` routine and is described in "Implementing the server" on page 18-12.
- **impl.h**—Declares the `AccountImpl` and `AccountManagerImpl` classes.
- **impl.C**—Implements the `AccountImpl` and `AccountManagerImpl` classes.
- **handler.h**—Declares the `BankImplHandler`, derived from `ImplEventHandler`.
- **handler.C**—Implements the event handler for transaction semantic support, described in "Handling transaction semantics" on page 18-9.
- **activate.h**—Declares the `AccountActivator` class, derived from the `Activator` class.
- **activate.C**—Implements the `Activator` for `Account` objects, described in "Creating a service activator for persistent objects" on page 18-10.
- **tieobj.h**—Implements the persistent tie template classes that link the persistent object implementations to the ORB objects offered by the server.
- **schema.C**—Contains ObjectStore-specific code.
- **bankclnt.C**—Implements a client program that adds accounts.

- **listaccts.C**—Implements a client program that lists all `Account` objects and invokes operations on them.
- **Makefile**—Builds the client and server programs.

Looking at the bank interfaces

The bank example defines an `AccountManager` object that contains and manages multiple `Account` objects. Client programs bind to the `AccountManager` object and then either create a new `Account` object or obtain an object reference to an existing `Account` object. Once a client program has a reference to an `Account` object, it can query the account's name and current balance. The IDL specification shown in IDL sample 18.1 shows the `AccountManager` and `Account` interfaces.

IDL sample 18.1 Account and AccountManager interfaces

```
// Bank.idl

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

      typedef sequence<Account> AccountList;

      interface AccountManager {
            void open(in string name, in float balance);
            AccountList GetAccountList();
      };
};
```

Generating persistent tie template classes

You can generate the persistent tie template classes needed for implementing persistent objects by using the command shown in IDL sample 18.2.

IDL sample 18.2 Generating the persistent tie templates

```
prompt> idl2cpp -ptie -no_tie bank.idl
```

The `-ptie` option causes the compiler to generate persistent tie template classes for all of the interfaces defined in the IDL file. In this case, the `-no_tie` option is also used to suppress the generation of the regular tie template classes. Although not strictly necessary, suppressing `tie` generation may reduce compile time. After you run the `idl2cpp` compiler, the following files will have been generated.

- **bank_c.cpp**
- **bank_c.hh**
- **bank_s.cpp**
- **bank_s.hh**

The **bank_s.hh** file contains two templates—Bank_ptie_Account, shown in Code sample 18.1, and Bank_ptie_AccountManager, shown in Code sample 18.2. Use these template classes to derive your own template classes which will, in turn, be used to create *delegator implementations* for persistent versions of the Account and AccountManager objects.

Bank_ptie_Account template

The following code sample shows what the idl2cpp compiler produces for the Bank_ptie_Account in the **bank_s.hh** file.

You will use this template class to derive your own template class PAccountImpl described later in this chapter.

Code sample 18.1 Bank_ptie_Account template

```
template <class T>
class Bank_ptie_Account : public Bank::Account
{
      protected:
               Bank_ptie_Account(const char *obj_name=(char*)NULL) :
                        Bank::Account(obj_name) {
                  _object_name(obj_name);
               }
               Bank_ptie_Account(const char *service_name,
                        const CORBA::ReferenceData& id) {
                  _service(service_name, id);
               }
      public:
               virtual ~Bank_ptie_Account() {}
               char * name() {
                     return _get_impl()->name();
               }
               CORBA::Float balance() {
                     return _get_impl()->balance();
               }

      protected:
               virtual T *_get_impl() =0;
};
```

Bank_ptie_AccountManager template

The following code sample shows what the idl2cpp compiler produces for the Bank_ptie_AccountManager in the **bank_s.hh** file.

You will use this template class to derive your own template class PAccountImpl described later in this chapter.

Code sample 18.2 Bank_ptie_AccountManager template

```
template <class T>
class Bank_ptie_AccountManager : public Bank::AccountManager
{
      protected:
               Bank_ptie_AccountManager(const char *obj_name=(char*)NULL) :
                        Bank::AccountManager(obj_name) {
                  _object_name(obj_name);
               }
               Bank_ptie_AccountManager(const char *service_name,
                        const CORBA::ReferenceData& id) {
                  _service(service_name, id);
               }
      public:
               virtual ~Bank_ptie_AccountManager() {}
               void open(const char * name, CORBA::Float balance) {
                     _get_impl()->open(name, balance);
               }
               Bank::AccountList * GetAccountList() {
                     return _get_impl()->GetAccountList();
               }
      protected:
               virtual T *_get_impl() =0;
};
```

Integrating an object database using template features

The persistent tie template classes generated by the `idl2cpp` compiler contain two important features that allow you to easily integrate with an object database. The first is a constructor that accepts a `service_name` and a `ReferenceData` parameter. The `service_name` parameter can be used to represent the name of the `Activator` associated with the object implementation. The `ReferenceData` parameter can be used to store a database reference to the persistent object. See "Automatically activating servers with the Object Activation Daemon" on page 6-5 for more information.

The second feature is the pure virtual `_get_impl()` method, which returns a pointer to the persistent object. You provide the implementation for this method, which is used in the delegation of all client requests to the persistent implementation.

Deriving template classes to use your object database

After generating the persistent tie templates, you are now ready to derive your own template classes that use your object database. Code sample 18.3 and Code sample 18.4 show the `PAccountImpl` and `PAccountManagerImpl` template classes created for the bank example.

The templates provide a private data member named `os_Reference`, an ObjectStore object reference that is used to implement the `_get_impl()` method. Both templates also define a constructor that accepts a reference to the implementation object as well as the object name.

The constructor for PAccountImpl must provide a means for the AccountActivator (described in "Creating a service activator for persistent objects" on page 18-10) to establish the correct association between the PAccountImpl object and the persistent AccountImpl object. To accomplish this, the PAccountImpl constructor converts the input ObjectStore reference ref to a string and then uses it to create a ReferenceData object named id. Next, the _service() method is used to associate the AccountActivator object with the ObjectStore reference contained in the id parameter.

Code sample 18.3 shows the PAccountImpl class from the **tieobj.h** file.

Code sample 18.3 PAccountImpl template class

```
template <class T>
class PAccountImpl : public Bank_ptie_Account<T>
{
      public :
               PAccountImpl (T& ref, const char* obj_name) :
                     Bank_ptie_Account <T> (obj_name), _ref (&ref)
               {
                     char* osref = _ref.dump();
                     CORBA::ReferenceData id(strlen (osref)+1,
                                    strlen (osref) + 1,
                                    (CORBA::Octet*) osref, 1);
                     _service("AccountActivator", id);
               }

               T* _get_impl () {
                     return (T*) _ref;
               }

      private :
               os_Reference <T> _ref;
};
```

Code sample 18.4 shows the PAccountManagerImpl class from the **tieobj.h** file.

Code sample 18.4 PAccountManagerImpl template class

```
template <class T>
class PAccountManagerImpl : public Bank_ptie_AccountManager<T>
{
      public:
               PAccountManagerImpl (T& ref, const char* obj_name) :
                        Bank_ptie_AccountManager <T>(obj_name), _ref (&ref) {}

               T *_get_impl() {
                     return (T*) _ref;
               }

      private :
               os_Reference <T> _ref;
};
```

Creating persistent object classes

The **impl.h** and **impl.C** files contain the definitions and implementations of the `AccountImpl` and `AccountManagerImpl` classes. Instances of these classes will be stored in the ObjectStore database. ObjectStore requires each of these classes to define a static `get_os_typespec()` method. If you are using a different database for your persistent objects, you should check the documentation for your database to see what is required for your implementation.

Code sample 18.5 shows the `AccountImpl` class definition.

Code sample 18.5 AccountImpl class

```
class AccountImpl
{
      public:
               AccountImpl(const char* author, float balance);
               virtual ~AccountImpl() {}
               static os_typespec* get_os_typespec();
               char* name () const;
               float balance () const;
      private:
               String _name;
               float _balance;
};
```

Code sample 18.6 shows the `AccountManagerImpl` class definition.

Code sample 18.6 AccountManagerImpl class

```
class AccountManagerImpl
{
      public:
               AccountManagerImpl (const char* name = "");
               virtual ~AccountManagerImpl() {}
               static os_typespec* get_os_typespec();
               char* name() const { return _name.value(); }

               void open (const char* name, float balance);

#ifndef _SCHEMA_ // because AccountList is an IDL-generated class
               Bank::AccountList* GetAccountList() const;
#endif
      private:
               String _name;
               os_List<AccountImpl*>& _account_list;
};
```

Note For ObjectStore users, code generated by `idl2cpp` is not to be compiled by the ObjectStore scheme generator. Because the schema generator will not see `idl2cpp` generated code, all references to generated types (such as `Bank::AccountList`) are made invisible using the preprocessor definition `_SCHEMA_`.

Handling transaction semantics

ObjectStore requires that every access to a persistent object be made within the context of a transaction. The example program uses interceptors to support the required transaction semantics. See Chapter 16, “Instrumenting and modifying the ORB with interceptors,” for details.

The example server implementation registers an interceptor. Whenever a client request for a persistent object is received, the interceptor’s `pre_method()` method is used to *begin* the transaction and the `post_method()` method is used to *commit* the transaction.

Code sample 18.7 Handling transaction semantics using interceptors

```
...
int main(int argc, char * const *argv)
{
      // initialize ORB
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      CORBA::ORB_prt boa = orb->BOA_init(argc, argv);

      // setup event handlers
      VIS_EXT::HandlerRegistry_ptr regHandle;
      regHandle = VIS_EXT::HandlerRegistry::instance();

      // our event handler to start/end transactions
      BankImplHandler VHandler;
      try {
            regHandle->reg_glob_impl_handler(&VHandler);
      }
      catch (const VIS_EXT::HandlerExists& excep) {
            cout << "Handler exists " << endl;
      }
      ...
```

The `BankImplHandler` class, shown in Code sample 18.8 defines an ObjectStore transaction pointer, `_trans`, as a private data member. This pointer is used to *begin* and *commit* a transaction.

Code sample 18.8 BankImplHandler class

```
class BankImplHandler : public VIS_EXT::ImplEventHandler
{
      public :
            void pre_method (const VIS_EXT::ConnectionInfo& ref,
                     CORBA::Principal_ptr, const char*, CORBA::Object_ptr);

            void post_method (const VIS_EXT::ConnectionInfo&,
                     CORBA::Principal_ptr, const char*, CORBA::Object_ptr);
         private :
            os_transaction* _trans;
};
```

Code sample 18.9 shows the implementation of the `pre_method()` and `post_method()` for use with the ObjectStore database. The `pre_method()` creates an ObjectStore transaction

and initializes the BankImplHandler::_trans pointer. The post_method() commits and then deletes the transaction pointed to by the _trans pointer.

Code sample 18.9 Using the pre_method() and post_method() implementations with the ObjectStore database

```
void BankImplHandler::pre_method (const VIS_EXT::ConnectionInfo&,
         CORBA::Principal_ptr, const char*, CORBA::Object_ptr)
{
      OS_ESTABLISH_FAULT_HANDLER
      // start local transaction
      _trans = os_transaction::begin (os_transaction::update);
      OS_END_FAULT_HANDLER
}
...

void BankImplHandler::post_method (const VIS_EXT::ConnectionInfo&,
         CORBA::Principal_ptr, const char*, CORBA::Object_ptr)
{
      OS_ESTABLISH_FAULT_HANDLER
      // end local transaction
      os_transaction::commit();
      delete _trans;
      OS_END_FAULT_HANDLER
}
```

Creating a service activator for persistent objects

The bank example design must allow a client program to use an Account object reference without first binding to an AccountManager object. This situation might arise if a client obtains an Account object reference, converts it to a string, and then uses it later without first binding to the AccountManager object. Instead of automatically loading all Account objects from the database when the server starts up, an Activator can be created to load an Account object only upon client request.

Code sample 18.10 shows the AccountActivator class, derived from the CORBA_Activator class. Code sample 18.11 shows the implementation of the activate() and deactivate() methods.

Code sample 18.10 AccountActivator class

```
class AccountActivator : public CORBA_Activator
{
      public :
            virtual CORBA::Object_ptr activate(
                     CORBA::ImplementationDef_ptr impl);
            virtual void deactivate(CORBA::Object_ptr,
                     CORBA::ImplementationDef_ptr impl);
};
```

activate() and deactivate() methods

The `activate()` method must create a new `PAccountImpl` object initialized with a previously created persistent object. To do this, the `activate()` method retrieves the `ReferenceData` parameter from the implementation definition input parameter. The `ReferenceData` parameter contains an ObjectStore reference, stored as a string, to the previously created persistent object. The `ReferenceData` parameter is converted from a string back into an ObjectStore reference and is then used to initialize a newly created `PAccountImpl` object.

Code sample 18.11 Using the activate() and deactivate() methods with service activation of persistent objects

```
CORBA::Object_ptr AccountActivator::activate(CORBA::ImplementationDef_ptr
            impl)
{
   os_Reference <AccountImpl> accountRef;
   PAccountImpl<AccountImpl> *accountObj;

   OS_ESTABLISH_FAULT_HANDLER;
   os_transaction::begin(os_transaction::update);

   CORBA::ReferenceData_var ref(impl->id());

   // assign the dump string to an os reference
   accountRef.load((char*) ref->data(), dbPtr);

   // make sure you resolve the os reference before using it!
   accountObj = new PAccountImpl <AccountImpl> (*(accountRef.resolve()), "");

   boa->obj_is_ready(accountObj);
   CORBA::Object::_duplicate(accountObj);

   os_transaction::commit();
   OS_END_FAULT_HANDLER;

   return (accountObj);
}
...

void AccountActivator::deactivate (CORBA::Object_ptr obj,
         CORBA::ImplementationDef_ptr impl)
{
   cout << "Deactivate called " << endl;
   obj->_release ();
}
```

Implementing the server

The example bank server, implemented in the **server.C** file, performs the following tasks:

1. Initializes the ORB and sets up the interceptor (shown in Code sample 18.7 on page 18–9).
2. Performs the ObjectStore initialization and opens database (shown in Code sample 18.12).
3. Begins a transaction and then searches for the "Bank" database root. If the root is not found, it creates a root labelled "Bank." This root is then set to return a newly created AccountManager object.

The following code sample shows the `ObjectStore` initialization.

Code sample 18.12 Portion of the main routine showing the ObjectStore initialization

```
...
// ObjectStore setup
objectstore::initialize();
os_collection::initialize();

OS_ESTABLISH_FAULT_HANDLER;

// setup the database file name and open it
cout << "Creating database now ...." << endl;
dbPtr = os_database::open ("Bank.db", 0, 0664);

// start a database transaction and look for the bank object
os_transaction::begin(os_transaction::update);

os_database_root *a_root;
a_root = dbPtr->find_root("Bank");
AccountManagerImpl* acctMgrImpl = 0;

if (!a_root) {
      cout << "Creating root now .... " << endl;
      a_root = dbPtr->create_root("Bank");

      acctMgrImpl = new (dbPtr, AccountManagerImpl::get_os_typespec())
                AccountManagerImpl();
      a_root->set_value(acctMgrImpl, AccountManagerImpl::get_os_typespec());
}

cout << "Found database root .... " << endl;
// get the Bank::AccountManager object from the database
acctMgrImpl = (AccountManagerImpl*) a_root->get_value
         (AccountManagerImpl::get_os_typespec());
...
```

After completing the initial tasks as shown in Code sample 18.12, the example bank server

1. Activates the `AccountActivator` object.
2. Creates the `PAccountManagerImpl` object and initializes it with the `acctMgrImpl` object from the ObjectStore database.
3. Registering the ORB object.

4 Commits the transaction.

The following code sample shows the creation and activation of the `PAccountManagerImpl` object.

Code sample 18.13 Portion of the main routine showing the creation and activation of the PAccountManagerImpl object

```
    ...
    boa->impl_is_ready("Activator", new AccountActivator(), 0);

    // create an orb object (tie) to the persistent AccountManager object
    PAccountManagerImpl<AccountManagerImpl> tieServer
                (*acctMgrImpl, "Bank Manager");
    boa->obj_is_ready(&tieServer);

    // end of transaction, changes are committed to the database
    os_transaction::commit();

    // server is ready to handle requests
    cout << "Press any key to exit " << endl;
    getchar();

    dbPtr->close();
    OS_END_FAULT_HANDLER;
    return 0;
}
```

Running the example

Before attempting to run the example server or any of the clients, be sure that your environment is properly set up and that a Smart Agent process is running within your network. See the *VisiBroker for C++ Installation and Administration Guide* for instructions on starting a Smart Agent.

Starting the server

You can start the server by using the following command:

```
prompt> bankserver
```

Running the client

You can use the **bankclnt** program to add an account with the name "John" and a balance of $400 by using the following command:

```
prompt> bankclnt John 400
```

Note The client program simply accesses the remote object using its object reference. The server is responsible for loading the object from the persistent store in response to the client request. The persistent storage is transparent to the client.

Chapter

19

Discovering object instances 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.

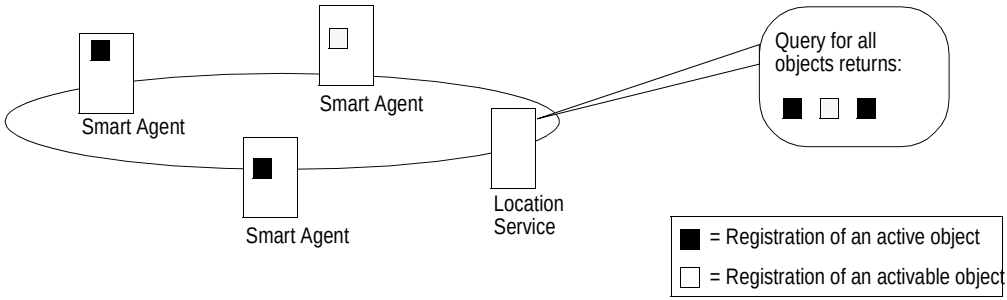
This chapter includes the following major sections:

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,* or a list of the instances it knows along with information it knows about the instances. When queried by the Location Service, a Smart Agent forwards the query to the other Smart Agents, and aggregates their replies in the result it returns to the Location Service. In this way, the Location Service "sees" all the instances of an object to which a client can bind. Smart Agents only know about globally scoped instances that are *accessible*. An *accessible* instance is either an *active* object—an object whose server is running and

has invoked the `obj_is_ready()` method for the object—or an *activable* object—an object that has been registered with an Object Activation Daemon (OAD). The following diagram illustrates this concept.

Figure 19.1 Location Service uses Smart Agents to find instances of objects



Note A server specifies an instance's scope when it creates the instance, as described in Chapter 6, "Activating objects and implementations." Only globally-scoped instances are registered with Smart Agents.

The Location Service can make use of the information the Smart Agent keeps about each object instance. For each object instance, the Location Service maintains information encapsulated in the structure `ObjLocation::Desc` shown in IDL sample 19.1.

IDL sample 19.1 IDL for the Desc structure

```
struct Desc {
    Object ref;
    IIOP::ProfileBody 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:

- An *object reference*, or a handle for invoking the object.
- The host name of the Smart Agent with which the instance is registered.
- A *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*, or the name given to the object by its server.
- An activable flag which differentiates between instances that can be activated by an OAD, and instances that are manually started.
- The host name and port of the instance's server. This information is only meaningful if the object reference supports IIOP. Host names are returned as strings in the instance description.

The Location Service is useful for purposes such as load balancing. 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. See Chapter 16, "Instrumenting and modifying the ORB with interceptors," for more information about writing interceptors.

Prerequisites for using the Location Service

To understand the material in this chapter you should be familiar with object scoping and the Basic Object Adaptor (BOA). See Chapter 6, "Activating objects and implementations," for more information on the BOA.

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 are a notification mechanism by which clients of the Location Service can be notified of changes to the availability of instances.

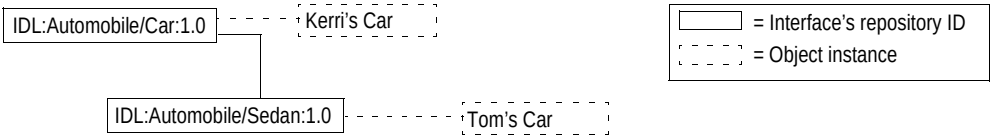
What is the Location Service agent?

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*. Figure 19.2 illustrates the use of interface repository IDs and instance names given the following example IDL:

Example

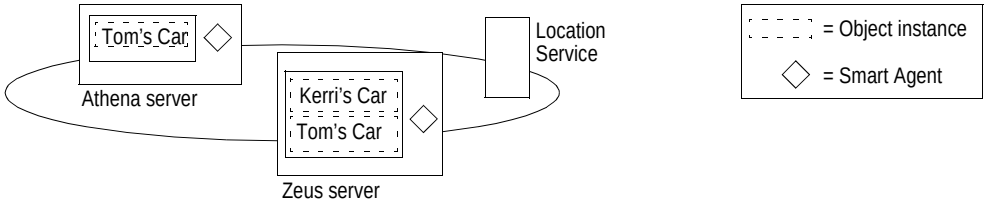
```
module Automobile {
         interface Car{...};
         interface Sedan:Car {...};
       }
```

Figure 19.2 Use of interface repository IDs and instance names



Given the example in Figure 19.2, the following diagram visually depicts Smart Agents on a network with references to instances of Car. In this example, there are three instances: one instance of Kerri's Car and two replicas of Tom's Car.

Figure 19.3 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 names of all hosts running Smart Agents

Using the `HostnameSeq all_agent_locations()` method, you can find out which servers are hosting VisiBroker Smart Agents. In the example shown in Figure 19.3, this method would return the names 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 all_repository_ids()` method. In the example shown in Figure 19.3 on page 19-4, 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 Figure 19.2 on page 19-3, 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:

Table 19.1 Obtaining references to instances of an interface

Method	Description
`ObjSeq all_instances(in string repository_id)`	Use this method to return object references to instances of the interface.
`DescSeq all_instance_descs(in string repository_id)`	Use this method to return an instance description for instances of the interface.

In the example shown in Figure 19.3 on page 19-4, a call to either method with the request `IDL:Automobile/Car:1.0` would return three instances of the Car interface: Tom's Car on Athena, Tom's Car on Zeus, and Kerri's Car. The Tom's Car instance is returned twice because there are occurrences of it with two different Smart Agents.

Obtaining references to like-named instances of an interface

Using one of the following methods, you can query VisiBroker Smart Agents on a network to return all occurrences of a particular instance name.

Table 19.2 References to like-named instances of an interface

Method	Description
`ObjSeq all_replica(in string repository_id, in string instance_name)`	Use this method to return object references to like-named instances of the interface.
`DescSeq all_replica_descs(in string repository_id, in string instance_name)`	Use this method to return an instance description for like-named instances of the interface.

In the example shown in Figure 19.3 on page 19-4, a call to either method specifying the repository ID `IDL:Automobile/Sedan:1.0` and instance name Tom's Car would return two instances because there are occurrences of it with two different Smart Agents.

What is a trigger?

A trigger is essentially a callback mechanism that lets you determine changes to the availability of a specified instance. It is an asynchronous alternative to polling an `Agent`, and is typically used to recover after the connection to an object has been lost. Whereas queries can be employed in many ways, triggers are special-purpose.

Looking at trigger methods

The trigger methods in the `Agent` class are described in the following table:

Table 19.3 Trigger methods

Methods	Description
`void reg_trigger(in TriggerDesc desc, in TriggerHandler handler)`	Use this method to register a trigger handler.
`void unreg_trigger(in TriggerDesc desc, in TriggerHandler 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 table:

Table 19.4 TriggerHandler interface method

Method	Description
`void impl_is_ready(in Desc desc)`	This method is called by the Location Service when an instance matching the `desc` becomes accessible.
`void impl_is_down(in 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 _sk_TriggerHandler class, and implementing its impl_is_ready() and impl_is_down() methods. To register a trigger with the Location Service, you use the reg_trigger() method in the Agent interface. This method requires that you provide a description of the instance you want to monitor, and the TriggerHandler object you want invoked when the availability of the instance changes. The instance description (TriggerDesc) can contain combinations of the following instance information: repository ID, instance name, and host name. The more instance information you provide, the more particular your specification of the instance.

IDL sample 19.2 IDL for TriggerDesc

```
struct TriggerDesc {
    string repository_id;
    string instance_name;
    string host_name;
  };
```

Note The field of the TriggerDesc will be ignored if any of the string fields are set to the empty string (""). The default for each field value is the empty string.

For example, a TriggerDesc containing only a repository ID matches any instance of the interface. Looking back to our example in Figure 19.3 on page 19-4, a trigger for any instance of IDL:Automobile/Car:1.0 would occur when one of the following instances becomes available or unavailable: Tom's Car on Athena, Tom's Car on Zeus, or Kerri's Car. Adding an instance name of "Tom's Car" to the TriggerDesc tightens the specification so that the trigger only occurs when the availability of one of the two "Tom's Car" instances changes. Finally, adding a host name of Athena refines the trigger further so that it only occurs when the instance 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:

IDL sample 19.3 Account example interface definition

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

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:Account:1.0`.

Code sample 19.1 Finding all instances satisfying the Account interface

```
#include <locate_c.hh>

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_ptr the_agent = ObjLocation::Agent::_narrow(obj);

    // Query the Location Service for all instances of
    // the Account interface
    ObjLocation::ObjSeq_var accountRefs;
    accountRefs = the_agent->all_instances("IDL:Account:1.0");
    cout << "Obtained "<< accountRefs->length()
    << "Account objects" << endl;

   // Convert the references to strings and display
   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 everything known to Smart Agents

The following code sample shows how to find everything known to Smart Agents. It does this by invoking the `all_repository_ids()` method to obtain all known interfaces. Then it invokes the `all_instances_descs()` method for each interface to obtain the instance descriptions.

Code sample 19.2 Finding everything known to a Smart Agent

```
#include "locate_c.hh"

int main(int argc, char** argv)
{
  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;

    // Find all Repository Ids
    cout << "Obtaining all registered Repository Ids..." << endl;
    ObjLocation::RepositoryIdSeq_var repIds =
      the_agent->all_repository_ids();
    cout << "Agent returned " << repIds->length()
      << " Repository Ids" << 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 = " << descriptors[j].instance_name << endl;
         cout << "\tHost Name     = " << descriptors[j].iiop_locator.host<<endl;
         cout << "\tBOA Port      = " << descriptors[j].iiop_locator.port<<endl;
         cout << "\tActivable     = " << (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 two sections provide code samples that show how a trigger is implemented and registered.

Implementing a trigger handler

The following code sample implements 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.

Code sample 19.3 Implementing a trigger handler

```
#include <locate_s.hh>
// Instances of this class will be called back by the Agent when the
// event for which it is registered happens.

class TriggerHandlerImpl : public _sk_ObjLocation::_sk_TriggerHandler
{
public:
  TriggerHandlerImpl(ObjLocation::Agent_ptr agent,
           const ObjLocation::TriggerDesc& initial_desc)
    : _agent(ObjLocation::Agent::_duplicate(agent)),
      _initial_desc(initial_desc) {}

  void impl_is_ready(const ObjLocation::Desc& desc) {
    notification(desc, 1);
  }
  void impl_is_down(const ObjLocation::Desc& desc) {
    notification(desc, 0);
  }
private:
  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 << "\tBOA Port      = " << desc.iiop_locator.port << 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;
};
```

Registering the trigger handler

The following code sample registers a trigger. The main program creates a `TriggerHandlerImpl` that looks for any instance of the `Account` interface.

Code sample 19.4 Registering a trigger handler

```
// This code creates a TriggerHandlerImpl and registers it.
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_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);

   // Create a trigger description to notify us about
   // OSAgent changes with respect to Account objects
   ObjLocation::TriggerDesc desc;
   desc.repository_id = (const char*) "IDL:Account:1.0";
   desc.instance_name = (const char*) "";
   desc.host_name = (const char*) "";
```

```
    // Create and register the TriggerHandlerImpl
    ObjLocation::TriggerHandler_ptr 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;
}
```

Chapter

20

Event loop integration

This chapter describes how to integrate VisiBroker's event loop processing with other event-driven systems in a single threaded model. The information in this chapter only pertains to single threaded applications. It includes the following major sections:

Overview

When your single threaded server invokes the `BOA::impl_is_ready()` method, an event loop is entered that waits for the arrival of requests from client programs. Your object implementation may also need to interact with another event-driven system. In a multithreaded environment, you can solve this problem by simply using two threads—one thread that waits for VisiBroker events, and another thread that services events relevant to the other event mechanism.

This chapter is broken into several sections. The first two sections deal with integrating VisiBroker in a Windows programming environment. The third section covers integration with XWindows. The last section discusses use of the `Dispatcher` class, which can be customized for use with other event-driven systems.

Building single-threaded servers on Windows

VisiBroker provides a `WDispatcher` class that you can use to integrate VisiBroker events with Windows message events. The `WDispatcher` must be instantiated before any ORB object implementations are instantiated and before ORB or BOA methods are invoked. When you instantiate the `WDispatcher` object, you must pass it the window handle.

Note There are significant advantages to building a multithreaded server rather than integrating the ORB with the Windows event loop. For more information, see "Building multithreaded servers on Windows" on page 20-3.

Using the Win32 API

Code sample 20.1 shows how to use the `WDispatcher` class to integrate with the Windows event loop.

Code sample 20.1 Using the WDispatcher class with the Windows event loop

```
#include "wdisp.h"
...
// Windows main entry point
WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR,
               int nCmdShow)
{
   static char szAppName[] = "Account";
   HWND hwnd;
   MSG msg;
   WNDCLASS wndclass;


   // Initialize wndclass
   ...

   hwnd = CreateWindow(szAppName, "AccountServer", WS_OVERLAPPEDWINDOW,
               CW_USEDEFAULT, CW_USEDEFAULT, 200, 200,
               NULL, NULL, hInstance, NULL);

   WDispatcher *winDispatcher = new WDispatcher(hwnd);
   CORBA::ORB_var orb = CORBA::ORB_init(__argc, __argv);
   CORBA::BOA_var orb = orb->BOA_init(__argc, __argv);

   AccountImpl server("Jack B. Quick");

   boa->obj_is_ready(&server);

   ShowWindow(hwnd, nCmdShow);
   UpdateWindow(hwnd);

   // Enter message loop
   while(GetMessage(&msy, NULL, 0, 0) ) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
   }
   return msg.wParam;
}
```

Using Microsoft Foundation Classes

You may also use the `WDispatcher` class when developing client programs with the Microsoft Foundation Classes (MFC). When you derive your program class from the Microsoft `CWinApp` class, you need to provide an `InitInstance()` method. The `WDispatcher` object should be instantiated in the `InitInstance()` method, prior to any ORB-related activity.

Code sample 20.2 Using the WDispatcher class with MFC-based applications

```
#include <afxwin.h>
#include <dispatch/wdisp.h>
...
// Application class
class AccountClientApp : public CWinApp
{


   public:
            BOOL InitInstance();
};

BOOL AccountClientApp::InitInstance()
{
      m_pMainWnd = new MainWindow;
      m_pMainWnd->showWindow(m_nCmdShow);
      m_pMainWnd->UpdateWindow();

      WDispatcher *winDispatcher = new WDispatcher(m_pMainWnd);
      CORBA::ORB_var orb = CORBA::ORB_init(__argc, __argv);
      CORBA::BOA_var boa = orb->BOA_init(__argc, __argv);
      AccountImpl* server = new AccountImpl("Jack B. Quick");
      boa->obj_is_ready(server);
      return 1;
}

...
```

Building multithreaded servers on Windows

Building multithreaded servers using VisiBroker for C++ is a straightforward process. There are significant advantages to building a multithreaded server rather than integrating the ORB with the Windows event loop. These advantages are

- VisiBroker for C++ automatically manages threads so servers can handle multiple client requests. On multiprocessor systems running Windows NT, servers automatically distribute work among processors. In contrast, a server implemented with VisiBroker functions integrated into the Windows event loop, can process only a single client request at a time.
- If a single worker thread should block, the server can continue processing other requests. In contrast, a server implemented with VisiBroker functions integrated

into the Windows event loop may block if any request fails, because at that point, there is no way for the blocked thread to process other requests.

You may build multithreaded servers either directly on the Win32 API, or using MFC. The code examples following show how to initialize the ORB when using either the Win32 API or MFC.

Using the Win32 API

The following code example shows how to initialize the ORB for a multithreaded Win32 server without using MFC.

Code sample 20.3 Windows object server that does not use the WDispatcher class

```
// Windows main entry point
WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR,
            int nCmdShow) {
   static char szAppName[] = "Account";
   HWND hwnd;
   MSG msg;
   WNDCLASS wndclass;
   // Initialize wndclass
   ...
   hwnd = CreateWindow(szAppName, "AccountServer", WS_OVERLAPPEDWINDOW,
               CW_USEDEFAULT, CW_USEDEFAULT, 200, 200,
               NULL, NULL, hInstance, NULL);
   CORBA::ORB_var orb = CORBA::ORB_init(__argc, __argv);
   CORBA::BOA_var boa = orb->BOA_init(__argc, __argv);
   AccountImpl server("Jack B. Quick");
   boa->obj_is_ready(&server);

   ShowWindow(hwnd, nCmdShow);
   UpdateWindow(hwnd);
   // Enter message loop
   while(GetMessage(&msy, NULL, 0, 0) ) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
   }
   return msg.wParam;
}
```

Code sample 20.3 is almost identical to the `WDispatcher` example, Code sample 20.1 on page 20-2, except it does not create a `WDispatcher` object. The event loop handles normal Windows messages as usual. Client operation requests, however, do not flow through the event loop. Instead, VisiBroker will automatically assign a worker thread when a request arrives. The threads managed by VisiBroker are completely independent of the Windows event loop.

Using MFC

Code sample 20.4 shows how to create a multithreaded server using MFC. The `InitInstance()` method provides all ORB initialization. After initializing the ORB and BOA, the initialization code creates the `AccountImpl` object.

Whether the server is built directly on the Win32 API or using MFC, the multi-threaded VisiBroker library listens for incoming requests and creates worker threads to handle each request. The application does not need to perform other thread-specific coding.

Code sample 20.4 Multithreaded MFC server

```
#include <stdafx.h>
...
// Application class
class AccountClientApp : public CWinApp {
      public:
            BOOL InitInstance();
};
BOOL AccountClientApp::InitInstance() {
   m_pMainWnd = new MainWindow;
   m_pMainWnd->showWindow(m_nCmdShow);
   m_pMainWnd->UpdateWindow();
   CORBA::ORB_var orb = CORBA::ORB_init(__argc, __argv);
   CORBA::BOA_var boa = orb->BOA_init(__argc, __argv);
   AccountImpl* server = new AccountImpl("Jack B. Quick");
   boa->obj_is_ready(server);
   return 1;
}
...
```

Implementing a complex Windows user interface

A multithreaded server can implement a complex Windows user interface either directly on the Win32 API or using MFC.

A key point in building a multithreaded server with an MFC-based Windows user interface is that only certain threads may do user interface updates. Within an MFC application, either the main application thread or a `CWinThread`-derived class that the application created may do user interface updates. These restrictions are in place because MFC threads contain thread-local storage that is important for updating user interfaces. Performing a user interface update from a non-MFC thread causes errors because the system does not have the required local storage within the thread.

Because VisiBroker for C++ creates a worker thread for each incoming connection which are not MFC threads, the worker threads cannot perform user interface updates directly.

To interface between VisiBroker threads and MFC threads, the VisiBroker thread can post an invalidate message to the window to update. The message may contain either the needed information for update or the two threads may use a common object or data structure to pass information.

For example, a server needs to update the user interface when it handles a request. The object implementation that handles the request updates a shared data structure that contains the request count. It then posts a `WM_PAINT` message to the window to paint. In an MFC-based server, the worker thread can invoke the MFC function `CWnd::Invalidate` to post the `WM_PAINT` message.

In either case, the window's painting code accesses the common data structure containing the needed counter and repaints the window. Because the window updates in response to a message, the painting occurs within the window's own thread.

Integrating VisiBroker events with other environments

To integrate your program with another system's event loop, you need to derive your own class from the `Dispatcher` class. The methods of your new class need to be implemented using the methods and interfaces provided by the event handling mechanism with which you are integrating. The details of the implementation will depend on the event system with which VisiBroker is being integrated.
Code sample 20.5 shows how you might create your own `Dispatcher` class.

Code sample 20.5 Deriving your own Dispatcher class from Dispatcher

```
class MyDispatcher : public Dispatcher {
   public:
            MyDispatcher();
            virtual ~MyDispatcher();
            virtual void link(int fd, DispatcherMask, IOHandler*);
            virtual IOHandler* handler(int fd, DispatcherMask) const;
            virtual void unlink(int fd);
            virtual void startTimer(long sec, long usec, IOHandler *);
            virtual void stopTimer(IOHandler *);
            virtual iv_boolean setReady(int, DispatcherMask)
                        { return 0; } // No need to implement
            virtual void dispatch();
            virtual iv_boolean dispatch(long&, long& )
                        { return 0; } // No need to implement
            virtual iv_boolean dispatch(timeval *val);
            ...
};
```

Building single-threaded servers on XWindows systems

Note This implementation is for single-threaded servers only.

U VisiBroker provides an `XDispatcher` class that you can use to integrate your program with the XWindows `XtMainLoop`. The `XDispatcher` class registers the file descriptors it uses for its connections with the Xt event loop and installs the appropriate event handlers. The result is that the Xt event loop receives and dispatches events for both XWindow and VisiBroker events. When an event occurs on one of VisiBroker's file

descriptors, the Xt event loop will invoke the appropriate VisiBroker method to process the data.

Code sample 20.6 shows how you might use the XDispatcher class in your object implementation. Applications that use the XDispatcher class should link with the library **libxdispatch.a** in addition to all the other appropriate VisiBroker libraries.

Code sample 20.6 Using the XDispatcher class

```
#include "xdisp.h"
int main(int argc, char * const *argv) {
   // Instantiate XDispatcher before invoking any VisiBroker methods.
   XDispatcher xdisp;

   // Initialize ORB and BOA.
   CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
   CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);
   ...
   // Enter the event loop for processing both ORB and Xt events.
   // Either XtMainLoop or boa::impl_is_ready may be invoked here to enter
   // this loop.
   XtMainLoop();
   //boa->impl_is_ready();
   AccountImpl* server=newAccountImpl("Jack B. Quick");
   boa->obj_is_ready(server);
         ...
}
```

Detecting events on several file descriptors with the Dispatcher class

The Dispatcher class is designed to detect events on several file descriptors and dispatch those events to the appropriate handler. The Dispatcher maintains three lists of file descriptors—one list for reading data, one list for writing data, and one for exceptions. You can use the link() method to add a file descriptor to one of the Dispatcher class' lists and define the IOHandler object to be invoked to handle events on that file descriptor. You can find the include file for the Dispatcher class in **include/dispatch/dispatch.h**.

Note The Dispatcher class is useful for single-threaded applications only.

An application should have only one instance of the Dispatcher class. The static instance() method is provided to allow access to that Dispatcher. Implementations of customized dispatchers should implement this method to return a reference to their own constructed Dispatcher.

Code sample 20.7 Dispatcher class

```
class Dispatcher {
   public:
         enum DispatcherMask {
               ReadMask,
               WriteMask,
               ExceptMask
         };
```

```
            Dispatcher();
            virtual ~Dispatcher();
            virtual void link(int fd, DispatcherMask, IOHandler*);
            virtual IOHandler handler(int fd, DispatcherMask) const;
            virtual void unlink(int fd);

            virtual void startTimer(long sec, long usec, IOHandler*);
            virtual void stopTimer(IOHandler*);

            virtual iv_boolean dispatch(long& sec, long& usec);
            virtual iv_boolean dispatch(timeval *);
            virtual iv_boolean dispatch();
            static Dispatcher& instance();
            ...
};
```

Adding file descriptors

When using the link() method to add a file descriptor to the `Dispatcher`, you specify the file descriptor, the `DispatcherMask`, and a pointer to an `IOHandler` object. The `DispatcherMask` value determines whether the file descriptor is added to the read, write, or exception event list.

When an event occurs on the file descriptor, the `Dispatcher` will invoke the appropriate `IOHandler` method to service the event. The `IOHandler` object provides methods for reading data from or writing data to a file descriptor as well as for handling exceptions and expired timers. If an `IOHandler` method returns a negative value indicating it encountered an error, the `Dispatcher` will automatically unlink the `IOHandler` from its file descriptor.

Note You must make multiple invocations of the link method if you want a particular file descriptor to be placed in more than one of the dispatcher's lists. You cannot logically "or" the `DispatcherMask` values together when invoking the `link()` method.

You can use the `handler()` method to return the `IOHandler` object defined for a particular file descriptor and `DispatcherMask` combination.

Setting event timers

You can set an interval timer for a particular `IOHandler` object by invoking the `startTimer()` method. This method lets you specify a time interval in a combination of seconds and microseconds. When the interval expires, the `IOHandler` object's `timerExpired()` method is invoked. The `Dispatcher` method `stopTimer()` can be invoked to stop a timer.

Note Timers are not periodic. Once they expire, you must set them again if you want to time another interval.

Watching for specific events with the dispatch() method

One form of the `dispatch()` method accepts no arguments and blocks indefinitely or until an event occurs on one of its file descriptors or a timer expires. If a file descriptor event occurs, the appropriate `IOHandler` method is invoked before the `dispatch()` method returns.

The other two forms of the `dispatch()` method accept a time interval specification. If the time interval specified is 0, the `Dispatcher` will return immediately after checking all the file descriptors and timers. If the time interval is greater than 0, the `Dispatcher` will block until an event occurs on one of the file descriptors or until the time interval expires. The `dispatch()` method returns 1 if an event on a file descriptor caused the return. This method returns 0 if an expired timer caused the `dispatch()` method to return.

Removing file descriptors

The `unlink()` method removes the specified file descriptor from all lists maintained by the `Dispatcher`.

Handling events for a specific file descriptor with the IOHandler class

You derive your own class from the `IOHandler` class to handle events on a particular file descriptor. You associate your `IOHandler` object with a file descriptor, using the `Dispatcher` object's `link()` method.

You can find the include file for this `IOHandler` in the following location: **include/dispatch/iohandl.h**.

Code sample 20.8 IOHandler class

```
class IOHandler {
   protected:
            IOHandler();
   public:
            virtual ~IOHandler();
            virtual int inputReady(int fd);
            virtual int outputReady(int fd);
            virtual int exceptionRaised(int fd);
            virtual void timerExpired(long sec, long usec);
};
```

Implementing the IOHandler methods

You must provide implementations for the `IOHandler` methods that you want to handle for your file descriptor. Table 20.1 describes each of the `IOHandler` methods.

Table 20.1 IOHandler class methods

Method name	Description
`inputReady()`	Called when the `Dispatcher` detects that data is ready to be read from the file descriptor associated with this handler.
`outputReady()`	Called when the `Dispatcher` detects that the file descriptor associated with this handler is ready to accept more data.
`exceptionRaised()`	Called when the `Dispatcher` detects that an I/O exception has occurred on the file descriptor associated with this handler.
`timer_expired()`	Called when the `Dispatcher` is notified that an interval timer for this handler has expired.

Table 20.2 shows the return code conventions that the `Dispatcher` class assumes your methods will follow.

Table 20.2 Return code conventions for IOHandler methods

Return value	Meaning
–1 or negative value	The method encountered an error and does not want to handle any more events.
0	The method has completed successfully and currently has no more work to do.
1 or a positive value	The method has completed successfully, but has more data to read or write. The `Dispatcher` will keep invoking this method, after checking all other file descriptors, until this method returns 0 or a negative value.

Creating an IOHandler

To create your own `IOHandler`, simply derive your own class and implement those methods you intend to use. Code sample 20.9 shows an example `IOHandler`-derived class.

Code sample 20.9 Example IOHandler-derived class

```
#include "iohandle.h"
...

class MyHandler : public IOHandler {
   public:
            MyHandler();
            virtual ~MyHandler();
            virtual int inputReady(int fd) {
                  // read from file using fd
                  ...
                  if(done) {
                        return(0);
                  } else if(more_left_to_read) {
                        return(1);
                  } else if(failure) {
                        return(-1);
                  }
            }
         ...
};
```

Code sample 20.10 shows how you might instantiate your handler and link it to a file descriptor. In this example, when an input event occurs on `myfd` the `Dispatcher` will invoke the `my_handler::inputReady()` method to handle the event.

Code sample 20.10 Instantiating and linking a handler to a file descriptor

```
...
MyHandler my_handler;
Dispatcher &disp = Dispatcher::instance();
disp.link(myfd, Dispatcher::ReadMask, my_handler);
...
```

Chapter

21

Dynamically managed types

This chapter describes the DynAny feature of VisiBroker, which allows you to construct and interpret data types at runtime. It includes the following major sections:

Overview

The DynAny interface provides a way for client applications to dynamically create basic and constructed data types at runtime. It also allows server applications to interpret and extract information from an Any object, even if the type it contains was not known to the server at compile-time. The use of the DynAny interface enables you to build powerful client and server applications that create and interpret data types at runtime.

Example client and server applications that illustrate the use of DynAny are included in the dynany directory of the examples directory in the VisiBroker distribution. These example programs will be used to illustrate DynAny concepts in this chapter.

DynAny types

A DynAny object has an associated value that may either be a basic data type (such as boolean, int, or float) or a constructed data type. The DynAny interface, described in

detail in the *VisiBroker for C++ Reference*, provides methods for determining the type of the contained data as well as for setting and extracting the value of primitive data types.

Constructed data types are represented by the following interfaces, which are all derived from DynAny. Each of these interfaces provides its own set of methods that are appropriate for setting and extracting the values it contains.

Table 21.1 Interfaces derived from DynAny that represent constructed data types

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.

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.

Creating a DynAny

The ORB class provides a set of static methods that you can use to create a DynAny, or one if its derived types. These ORB methods are described in detail in the *VisiBroker for C++ Reference*.

You can use the ORB::create_dyn_any method to create a DynAny and initialize its value and TypeCode from an existing Any object. Alternatively, you can use the ORB::create_basic_dyn_any method to create a DynAny value for a basic data type.

Note The TypeCode of a DynAny is specified when the object is created and cannot be changed during the life of the object.

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 InvalidValue exception to be raised.

The DynAny::get_<type> methods allow you to access the value contained in a DynAny object, where <type> is boolean, octet, char, and so on. Any attempt to access a value from a DynAny component which does not match the TypeCode defined for the DynAny will cause a TypeMismatch exception to be raised.

The DynAny interface also provide methods for copying, assigning, and converting to or from an Any object. The sample programs, described later in this chapter, provide examples of how to use some of these methods. The *VisiBroker for C++ Reference* provides a complete description of these methods.

Constructed data types

The following types are derived from the DynAny interface and are used to represent constructed data types. These interfaces, and the methods they offer, all described in the *VisiBroker for C++ Reference*.

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	Resents 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.

DynEnum

This interface represents a single enumeration constant. Methods are provided for setting and obtaining the value as a string or as an integral value.

You create an DynEnum object by invoking the ORB::create_dyn_enum method.

DynStruct

This interface represents a dynamically constructed struct type. The members of the structure can be retrieved or set using a sequence of NameValuePair objects. Each

NameValuePair object contains the member's name and an Any containing the member's Type and value.

You may use the rewind, next, current_component, and seek methods to traverse the members in the structure. Methods are provided for setting and obtaining the structure's members.

You create an DynStruct object by invoking the ORB::create_dyn_struct method.

DynUnion

This interface represents a union and contains two components. The first component represents the discriminator and the second represents the member value.

You may use the rewind, next, current_component, and seek methods to traverse the components. Methods are provided for setting and obtaining the union's discriminator and member value.

You create an DynUnion object by invoking the ORB::create_dyn_union method.

DynSequence and DynArray

A DynSequence or DynArray represents a sequence of basic or constructed data types without the need of generating a separate DynAny object for each component in the sequence or array. The number of components in a DynSequence may be changed, while the number of components in a DynArray is fixed.

You may use the rewind, next, current_component, and seek methods to traverse the members in a DynArray or DynSequence.

You use the ORB::create_dyn_sequence to create a DynSequence object. Use the ORB::create_dyn_array to create a DynArray object.

Example IDL

Code sample 21.2 shows the IDL used in the example client and server applications. The StructType structure contains two basic data types and an enumeration value. The PrinterManager interface is used to display the contents of an Any without any static information about the data type it contains.

Code sample 21.1 IDL for the DynAny example clients

```
// Printer.idl

module Printer {
  enum EnumType {first, second, third, fourth};
  struct StructType {
    string str;
    EnumType e;
    float fl;
  };
```

```
  interface PrinterManager {
    void printAny(in any info);
    void shutdown();
  };
};
```

Example client application

Code sample 21.2 shows a client application that can be found in the dynany directory of the examples directory in the VisiBroker distribution. The client application uses the DynStruct interface to dynamically create an StructType structure.

The DynStruct interface uses a sequence of NameValuePair objects to represent the structure members and their corresponding values. Each name-value pair consists of a string containing the structure member's name and an Any object containing the structure member's value.

After initializing the ORB in the usual manner and binding to an PrintManager object, the client performs these steps:

1. Create an empty DynStruct with the appropriate type.
2. Create a sequence of NameValuePair objects that will contain the structure members.
3. Create and initialize Any objects for each of the structure member's values.
4. Initialize each NameValuePair with the appropriate member name and value.
5. Initialize the DynStruct object with the NameValuePair sequence.
6. Invoke the PrinterManager::printAny method, passing the DynStruct converted to a regular Any.

Note You must use the DynAny::to_any method to convert a DynAny object, or one of its derived types, to an Any before passing it as a parameter on an operation request.

Code sample 21.2 Example client application that uses DynStruct

```
int main(int argc, char **argv)
{
   try {
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
      Printer::PrinterManager_var account =
         Printer::PrinterManager::_bind("PrinterManager");

      // create a Dynamic Any Struct to put our info into
      CORBA::DynStruct_var info =
            orb->create_dyn_struct(Printer::_tc_StructType_get());

      // Use the NameValuePair to insert the members into the DynStruct
      CORBA::NameValuePairSeq sequence(3);
      sequence.length(3);

      sequence[(CORBA::ULong)0].id = CORBA::string_dup("str");
      sequence[(CORBA::ULong)0].value <<= "Jack";
```

```
        sequence[(CORBA::ULong)1].id = CORBA::string_dup("e");
        sequence[(CORBA::ULong)1].value <<= Printer::second;

        sequence[(CORBA::ULong)2].id = CORBA::string_dup("fl");
        sequence[(CORBA::ULong)2].value <<= (CORBA::Float)864.50;

        // put the sequence into our DynStruct
        info->set_members(sequence);
        // convert to an Any and send it to the server to be
        // displayed.
        CORBA::Any_var info_any = info->to_any();
        account->printAny(info_any);
    }
    catch (CORBA::UserException& user) {
        cerr << "User exception: " << user << endl;
        return 1;
    }
    . . .
    return 0;
}
```

Example server application

Code sample 21.3 shows a server application that can be found in the dynany directory of the examples directory in the VisiBroker distribution. The server application performs these steps.

1. Initialize the ORB and the BOA.
2. Create a PrintManager object.
3. Activate the PrintManager object.
4. Print a message and wait for incoming operation requests.

Code sample 21.3 Example server application

```
int main(int argc, char **argv)
{
     try {
    cout << "Contacting orb" << endl;
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    cout << "Initializing boa" << endl;
    CORBA::BOA_var boa = orb->BOA_init(argc, argv);

    cout << "Making PrinterManager" << endl;
    Printer::PrinterManager_var myImpl =
             new PrinterManagerImpl(orb, "PrinterManager");
    boa->obj_is_ready(myImpl);

    cout << "PrinterManager is ready." << endl;
    boa->impl_is_ready();
     }
```

```
    catch (CORBA::UserException& user) {
        cerr << "User exception: " << user << endl;
        return 1;
    }
    . . .
    return 0;
}
```

Code sample 21.4 shows how the `PrinterManager` implementation follows these steps in using a `DynAny` to process the `Any` object, without any compile-time knowledge of the type the `Any` contains.

1. Create a `DynAny` object, initializing it with the received `Any`.
2. Perform a `switch` on the `DynAny` object's type.
3. If the `DynAny` contains a basic data type, simply print out the value.
4. If the `DynAny` contains an `Any` type, create a `DynAny` for it, determine it's contents, and then print out the value.
5. If the `DynAny` contains an `enum`, create a `DynEnum` for it and then print out the string value.
6. If the `DynAny` contains a union, create a `DynUnion` for it and then print out the union's discriminator and the member.
7. If the `DynAny` contains a `struct`, array, or sequence, traverse through the contained components and print out each value.

Code sample 21.4 Implementation of PrinterManager showing the use of DynAny types to process a received Any object

```
PrinterManagerImpl(CORBA::ORB_ptr orb,
            const char* name = 0) : _sk_PrinterManager(name)
 { _orb = CORBA::ORB::_duplicate(orb); }

void printAny(const CORBA::Any& info) {
   try {
    CORBA::DynAny_var dynany = _orb->create_dyn_any(info);
        display(dynany);
   }
   . . .
}

void display(CORBA::DynAny* value) {
   CORBA::TypeCode_var tc = value->type();
   switch (tc->kind()) {
      case CORBA::tk_null :
      case CORBA::tk_void :
         break;
      case CORBA::tk_short :
         cout << value->get_short() << endl;
         break;
      . . .
```

```
            case CORBA::tk_any : {
               CORBA::Any_var any = value->get_any();
               CORBA::DynAny_var newdyn = _orb->create_dyn_any(*any);
               display(newdyn);
               }
               break;
            . . .
            case CORBA::tk_enum : {
               CORBA::DynEnum_var enumdyn = CORBA::DynEnum::_narrow(value);
               CORBA::String_var enumval = enumdyn->value_as_string();
               cout << enumval << endl;
               }
               break;
            case CORBA::tk_union : {
               CORBA::DynUnion_var uniondyn = CORBA::DynUnion::_narrow(value);
               CORBA::DynAny_var discrim = uniondyn->discriminator();
               CORBA::DynAny_var member = uniondyn->member();
               display(discrim);
               display(member);
               }
               break;
            case CORBA::tk_struct :
            case CORBA::tk_array :
            case CORBA::tk_sequence : {
               value->rewind();
               do {
                  CORBA::DynAny_var member = value->current_component();
                  display(member);
               } while (value->next());
               }
               break;
            . . .
            default :
               cout << "Invalid type" << endl;
            }
         }
      . . .

   }
```

Chapter

22

Using object wrappers

This chapter describes the object wrapper feature of VisiBroker, which allows your applications to be notified or to trap an operation request for an object. It includes the following major sections:

Overview

VisiBroker's object wrapper feature allows you to define methods that are to be invoked when a client application invokes a method on a bound object or when a server application receives an operation request. Unlike the interceptor feature described in Chapter 16, which is invoked at the ORB level, object wrappers are invoked before an operation request has been marshalled. In fact, you can design object wrappers to return results without the operation request having ever been marshalled, sent across the network, or actually presented to the object implementation.

Object wrappers may be installed on just the client-side, just the server-side, or they may be installed in both the client and server applications.

Here are a few examples of how you might use object wrappers in your application,

- Log information about the operation requests issued by a client or received by a server.

- Measure the time required for operation requests to complete.
- Cache the results of frequently issued operation requests so results can be immediately returned, without actually contacting the object implementation each time.

Note Externalizing a reference to an object for which object wrappers have been installed, using the `ORB::object_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 un-typed. You can mix the use of both of these types of wrappers within a single client or server application. Table 22.1 summarizes the important distinctions between these two types of object wrappers.

Table 22.1 Comparison of features for typed and un-typed object wrappers

Features	Typed	Un-typed
Receives all arguments that are to be passed to the stub.	Yes	No
Can return control to the caller without actually invoking the next wrapper, the stub, or the object implementation.	Yes	No
Will be invoked for all operation requests for all objects.	No	Yes

Special idl2cpp requirements

Whenever you plan to use typed or un-typed object wrappers, you must ensure that you use the `-obj_wrapper` option with the `idl2cpp` compiler when you generate the code for your applications. This will result in the generation of

- An object wrapper base class for each of your interfaces
- The `_this` method, generated for each skeleton classes, which is used on the server side when creating an object implementation that will use object wrappers.

Example applications

The `objwrap` directory, located in the `examples` directory, contains example client and server applications that will be used to illustrate object wrapper concepts in this chapter.

Un-typed object wrappers

Un-typed object wrappers allow you to define methods that are to be invoked before an operation request is processed, after an operation request is processed, or both.

Un-typed wrappers can be installed for client or server applications and you can also install multiple versions.

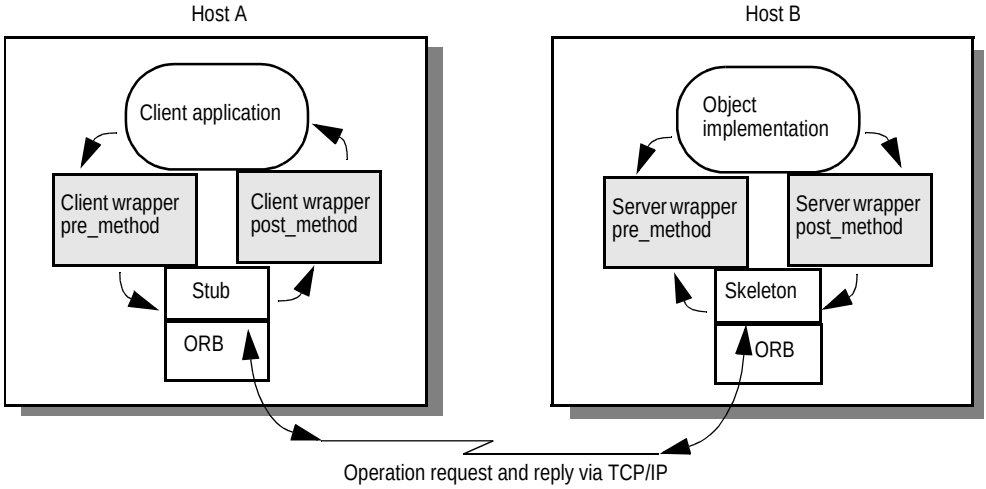
You may also mix the use of both typed and un-typed object wrappers within the same client or server application. For information on typed wrappers, see page 22–9.

By default, un-typed object wrappers have a global scope and will be invoked for any operation request. You can design un-typed wrappers so that they have no effect for operation requests on object types in which you are not interested.

Note Unlike typed object wrappers, un-typed wrapper methods do not receive the arguments that the stub or object implementation would receive nor can they prevent the invocation of the stub or object implementation.

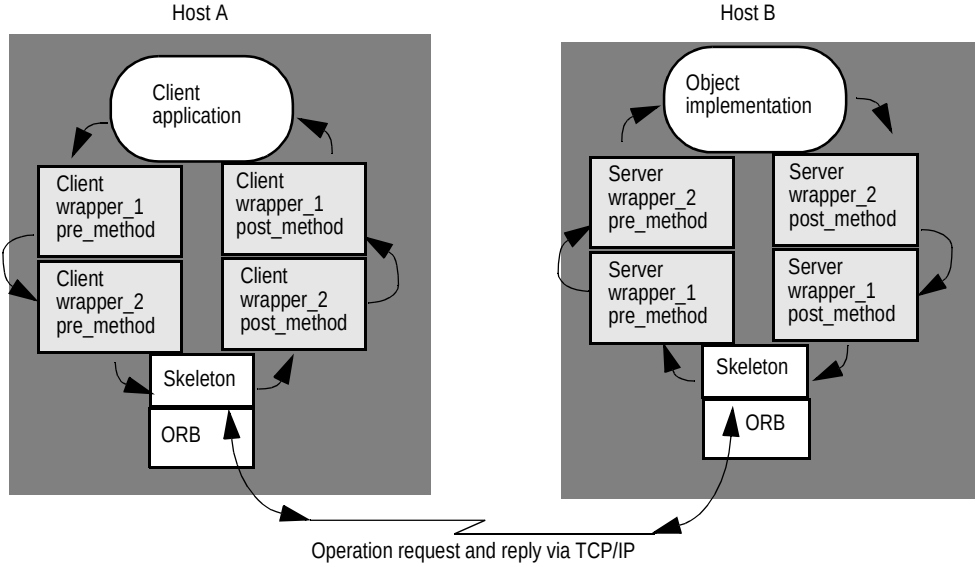
Figure 22.1 shows how an un-typed object wrapper's `pre_method` is invoked before the client stub method and how the `post_method` is invoked afterward. It also shows the calling sequence on the server-side with respect to the object implementation.

Figure 22.1 Single un-typed object wrapper installed for a client and a server application



Using multiple, un-typed object wrappers

Figure 22.2 Multiple un-typed object wrappers installed for a client and a server application



Order of pre_method invocation

When a client invokes a method on a bound object, each un-typed object wrapper `pre_method` will receive control before the client's stub routine is invoked. When a server receives an operation request, each un-typed object wrapper `pre_method` will be invoked before the object implementation receives control. In both cases, the first `pre_method` to receive control will be the one belonging to the object wrapper that was *registered first*.

Order of post_method invocation

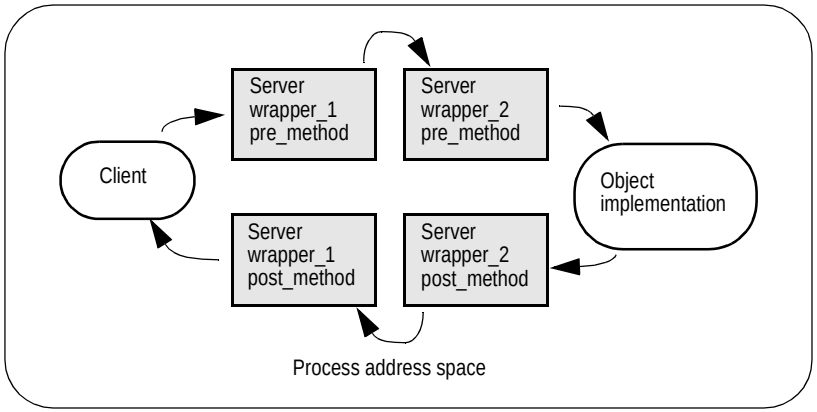
When a server's object implementation completes its processing, each `post_method` will be invoked before the reply is sent to the client. When a client receives a reply to an operation request, each `post_method` will be invoked before control is returned to the client. In both cases, In both cases, the first `post_method` to receive control will be the one belonging to the object wrapper that was *registered last*.

Note If you choose to use both typed and un-typed object wrappers, see "Combined use of un-typed and typed object wrappers" on page 22-15 for information on the invocation order.

Un-typed object wrappers with co-located client and servers

When the client and server are both packaged in the same process only the server-side `pre_method` and `post_method`s will receive control. Figure 22.3 illustrates the invocation order.

Figure 22.3 Un-typed object wrapper invocation order with co-located client and server objects



Using un-typed object wrappers

You must use the following steps when using un-typed object wrappers. Each step is discussed, in turn.

1. Identify the interface, or interfaces, for which you want to create a un-typed object wrapper.
2. Generate the code from your IDL specification using the `idl2cpp` compiler using the `-obj_wrapper` option.
3. Create an implementation for your un-typed object wrapper factory, derived from the `VISObjectWrapper::UntypedObjectWrapperFactory` class.
4. Create an implementation for your un-typed object wrapper, derived from the `VISObjectWrapper::UntypedObjectWrapper` class.
5. Modify your application to create your un-typed object wrapper factory.

Implementing an un-typed object wrapper factory

The **timewrap.h** file, part of the `objwrap` sample applications, illustrates how to define an un-typed object wrapper factory that is derived from the `VISObjectWrapper::-UntypedObjectWrapperFactory`. Your factory's `create` method will be invoked to create an un-typed object wrapper whenever a client binds to an object or a server creates an object implementation. The `create` method receives the target object, which allows

you to design your factory to not create an un-typed object wrapper for those object types you wish to ignore.

Code sample 22.1 shows the TimingObjectWrapperFactory, which is used to create an un-typed object wrapper that displays timing information for method calls. Notice the addition of the key parameter to the TimingObjectWrapperFactory constructor. This parameter is also used by the service initializer to identify the wrapper.

Code sample 22.1 TimingObjectWrapperFactory implementation from the timewrap.h file

```
class TimingObjectWrapperFactory
  : public VISObjectWrapper::UntypedObjectWrapperFactory
{
   public:
      TimingObjectWrapperFactory(VISObjectWrapper::Location loc,
            const char* key)
         : VISObjectWrapper::UntypedObjectWrapperFactory(loc),
            _key(key) {}

      //
      // ObjectWrapperFactory operations
      //

      VISObjectWrapper::UntypedObjectWrapper_ptr create(
               CORBA::Object_ptr target) {
         if (_owrap == NULL) {
               _owrap = new TimingObjectWrapper(_key);
         }
         return VISObjectWrapper::UntypedObjectWrapper::_duplicate(_owrap);
      }

   private:
      CORBA::String_var _key;
      VISObjectWrapper::UntypedObjectWrapper_var _owrap;
};
```

Implementing an un-typed object wrapper

Code sample 22.2 shows the implementation of the TimingObjectWrapper, also defined in the **timewrap.h** file. Your un-typed wrapper must be derived from the VISObjectWrapper::UntypedObjectWrapper class and you must provide an implementation for both the pre_method or post_method methods in your un-typed object wrapper.

Once your factory has been installed, either automatically by the factory's constructor or manually by invoking the VISObjectWrapper::ChainUntypedObject-Wrapper::add method, an un-typed object wrapper object will be created automatically whenever your client binds to an object or when your server creates an object implementation.

The pre_method shown in Code sample 22.2, 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 that has elapsed since the pre_method was called and print the elapsed time.

Code sample 22.2 TimingObjectWrapper implementation

```
void pre_method(const char* operation,
      CORBA::Object_ptr target,
      VISClosure& closure)
{
   cout << "*Timing: [" << flush;
   if (_key) {
         cout << _key << flush;
   } else {
         cout << "<no key>" << flush;
   }
   cout << "] pre_method" << endl;
   TimerBegin(closure, operation);
}

void post_method(const char* operation,
      CORBA::Object_ptr target,
      CORBA::Environment& env,
      VISClosure& closure)
{
   cout << "*Timing: [" << flush;
   if (_key) {
         cout << _key << flush;
   } else {
         cout << "<no key>" << flush;
   }
   cout << "] post_method " ;
   TimerDelta(closure, operation);
}
```

pre_method and post_method parameters

Both the pre_method and post_method receive these parameters:

Table 22.2 Common arguments for the pre_method and post_method methods

Parameter	Description
operation	The name of the operation that was requested on the target object.
target	The target object.
closure	An area where data can be save across method invocations for this wrapper.

The post_method also receives an Environment parameter, which can be used to reflect to the user any exceptions that might occur.

Creating and registering un-typed object wrapper factories

An un-typed object wrapper factory is automatically added to the chain of un-typed wrappers whenever it is created with the base class constructor that accepts a

location. If you use the default void constructor for the base class to create your object wrapper factory, you must register the factory using the `VISObjectWrapper::Chain-UntypedObjectWrapperFactory::add` method, described in the *VisiBroker for C++ Reference*.

On the client side, un-typed object wrapper factories must be created and registered before any objects are bound. On the server side, un-typed object wrapper factories must be created and registered before any implementation objects are created.

Code sample 22.3 shows a portion of the sample file **untypedClient.C**, which shows the creation, with automatic registration, of two un-typed object wrapper factories for a client. The factories are created after the ORB has been initialized, but before the client binds to any objects.

Code sample 22.3 Creating and registering two client-side, un-typed object wrapper factories

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

         // install untyped object wrappers
         TimingObjectWrapperFactory timingfact(VISObjectWrapper::Client,
                  "timeclient");
         TraceObjectWrapperFactory tracingfact(VISObjectWrapper::Client,
                  "traceclient");

         // Locate an account manager.
         Bank::AccountManager_var manager(
         Bank::AccountManager::_bind("BankManager"));
         . . .
```

Code sample 22.4 shows the sample file **untypedServer.C**, which shows the creation and registration of an un-typed object wrapper factories for a server. The factories are created after the ORB is initialized, but before any object implementations are created.

Notice that when the `AccountManager` object is created, the `_this` method is used, which will return the first object wrapper in the chain. When activating an object implementation or invoking a method it offers, the server must use the `_this` method. Failing to use the `_this` method will circumvent any object wrappers that may be registered.

Code sample 22.4 Registering a server-side, un-typed object wrapper factory

```
#include "bankimpl.h"
#include "tracewrap.h"

int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB.
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      // Initialize the BOA.
      CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);
      // install untyped object wrappers
      TraceObjectWrapperFactory tracingfact(VISObjectWrapper::Server,
         "traceserver");
```

```
        // Create the account manager object.
        Bank::AccountManager_var manager(
              (new AccountManager("BankManager"))->_this());
        // reference count is now two. One for the new, and one that is caused
        // by calling _this, which always duplicates the reference.
        // So we need to release the extra reference
        CORBA::release(manager);

        // Export the newly create object.
        boa->obj_is_ready(manager);
        cout << manager << " is ready." << endl;

        // Wait for incoming requests
        boa->impl_is_ready();
    }
        catch(const CORBA::Exception& e) {
        cerr << e << endl;
        return 1;
    }
    return 0;
}
```

Removing un-typed object wrappers

The `VISObjectWrapper::ChainUntypedObjectWrapperFactory::remove` method can be used to remove an un-typed object wrapper factory from a client or server application. You must specify a location when removing a factory. This means that if you have added a factory with a location of `VISObjectWrapper::Both`, you can selectively remove it from the `Client` location, the `Server` location, or both.

Note Removing one or more object wrapper factories from a client will not affect objects of that class that are already bound by the client. Only subsequently bound objects will be affected. Removing object wrapper factories from a server will not affect object implementations that have already been created. Only subsequently created and object implementation will be affected.

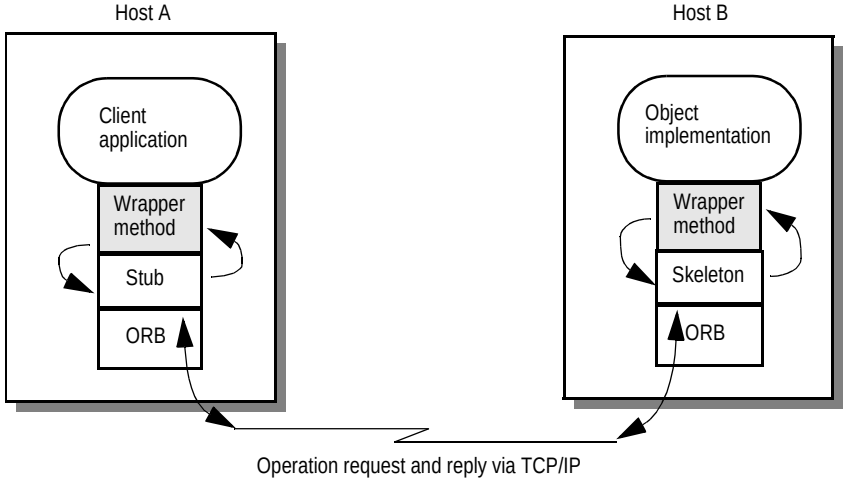
Typed object wrappers

When you implement a typed object wrapper for a particular class, you define the processing that is to take place when a method is invoked on a bound object. Figure 22.4 shows how an object wrapper method on the client is invoked before the client stub class method and how an object wrapper on the server-side is invoked before the server's implementation method.

Note Your typed object wrapper implementation is not required to implement all methods offered by the object it is wrapping.

You may also mix the use of both typed and un-typed object wrappers within the same client or server application. For more information, see "Combined use of un-typed and typed object wrappers" on page 22-15.

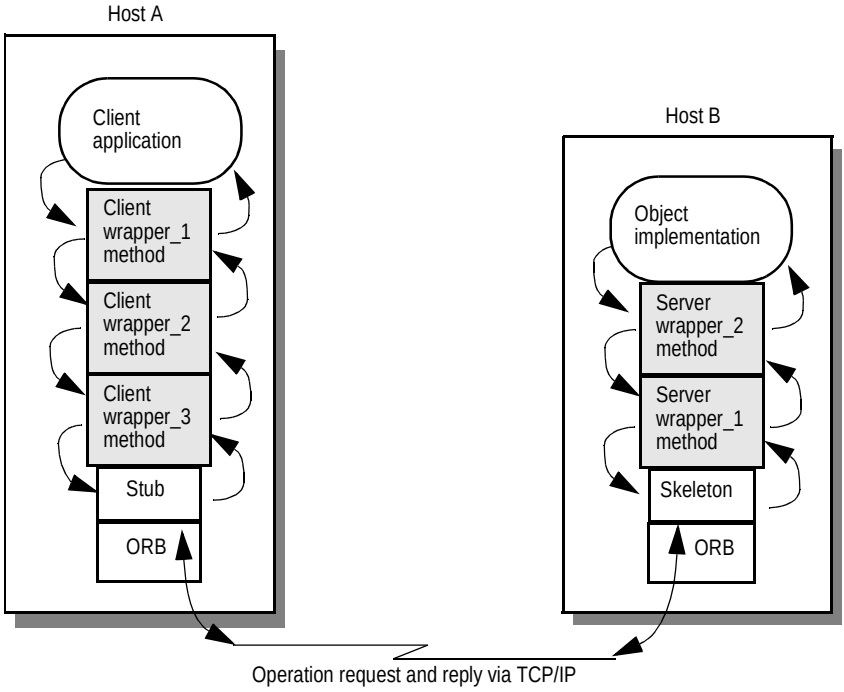
Figure 22.4 Single typed object wrapper registered for a client and a server application



Using multiple, typed object wrappers

You may implement and register more than one typed object wrapper for a particular class of object, as shown in Figure 22.5. On the client side, the first object wrapper registered was `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 was `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 skeleton.

Figure 22.5 Multiple, typed object wrappers registered for a client and a server application



Order of invocation

The methods for a typed object wrapper that are register for a particular class will receive all of the arguments that are normally passed to the stub method on the client side or to skeleton on the server side. Each object wrapper method can pass control to the next wrapper method in the chain by invoking the parent class' method, `<interface_name>ObjectWrapper::<method_name>`. If an object wrapper wishes to return control without calling the next wrapper method in the chain, it can `return` with the appropriate return value.

A typed object wrapper method's ability to return control to the previous method in the chain allows you to create a wrapper method that never invokes a client stub or object implementation. For example, you can create an object wrapper method that caches the results of a frequently requested operation. In this scenario, the first invocation of a method on the bound object results in an operation request being sent to the object implementation. As control flows back through the object wrapper method, the result is then stored. On subsequent invocations of the same method, the object wrapper method can simply return the cached result without actually issuing the operation request to the object implementation.

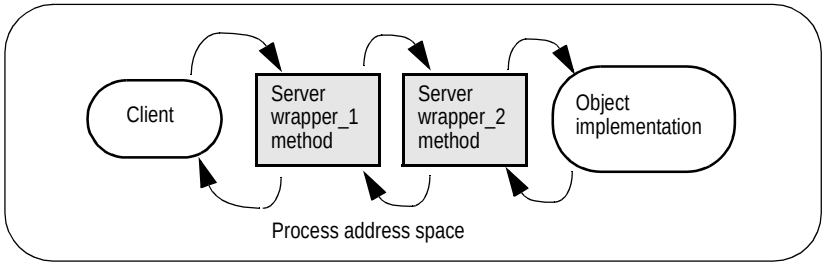
If you choose to use both typed and un-typed object wrappers, see "Combined use of un-typed and typed object wrappers" on page 22-15 for information on the invocation order.

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 received control will belong to the first server-side object wrapper that was installed. Figure 22.6 illustrates the invocation order.

Note Client object wrappers never receive control in a co-located client and server application.

Figure 22.6 Typed object wrapper invocation order with co-located client and server objects



Using typed object wrappers

You must use the following steps when using typed object wrappers. Each step is discussed in turn.

1. Identify the interface, or interfaces, for which you want to create a typed object wrapper.
2. Generate the code from your IDL specification using the `idl2cpp` compiler using the `-obj_wrapper` option.
3. Derive your typed object wrapper class from the `<interface_name>ObjectWrapper` class generated by the `idl2cpp` compiler and provide an implementation of those methods you wish to wrap.
4. Modify your application to register the typed object wrapper.

Implementing typed object wrappers

You derive typed object wrappers from the `<interface_name>ObjectWrapper` class that is generated by the `idl2cpp` compiler. Code sample 22.5 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. Invoke the `_inited` method to see if this method has been invoked before.
2. If this is the first invocation, invoked the next `balance` method on the next object in the chain, save the result in `_balance`, set `_inited` to `true`, and return the value.

3 If this method has been invoked before, simply return the cached value.

Code sample 22.5 Portion of the CachingAccountObjectWrapper implementation

```
class CachingAccountObjectWrapper : public Bank::AccountObjectWrapper
{
   public:
      CachingAccountObjectWrapper() : _inited((CORBA::Boolean)0) {}

      CORBA::Float balance() {
         cout << "+ CachingAccountObjectWrapper: before calling balance"
            << endl;
         if (! _inited) {
               _balance = Bank::AccountObjectWrapper::balance();
               _inited = 1;
         } else {
               cout << "+ CachingAccountObjectWrapper:returning cached copy"
                  << 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. Code sample 22.6 shows a portion of the `typed` **Client.C** file that creates and registers a typed object wrapper.

Code sample 22.6 Creating and registering a client-side, typed object wrapper

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

         // Install typed object wrapper for Account
         Bank::AccountObjectWrapper::add( orb,
                  CachingAccountObjectWrapper::factory,
                  VISObjectWrapper::Client);

         // Locate an account manager.
         Bank::AccountManager_var manager(
         Bank::AccountManager::_bind("BankManager"));
         . . .
```

The ORB keeps track of any object wrappers that have been registered for it on the client-side. When a client invokes the `_bind` method to bind to an object of that type, the necessary object wrappers will be created. If a client binds to more than one instance of a particular class of object, each instance will have its own set of wrappers.

Registering typed object wrappers for a server

As with a client application, a typed object wrapper is registered on the server-side by invoking the <interface_name>::add method. Server-side, typed object wrappers must be registered after the ORB_init method has been called, but before any object implementations are created or activated. Code sample 22.7 shows a portion of the **typedServer.C** file that installs a typed object wrapper.

Code sample 22.7 Registering a server-side, typed object wrapper

```
int main(int argc, char* const* argv) {
   try {
         // Initialize the ORB.
         CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
         // Initialize the BOA.
         CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

         // install two 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 object.
         Bank::AccountManager_var manager(
            (new AccountManager("BankManager"))->_this());
         // reference count is now two. One for the new, and one that is caused
         // by calling _this, which always duplicates the reference.
         // So we need to release the extra reference
         CORBA::release(manager);

         boa->obj_is_ready(manager);
         cout << manager << " is ready." << endl;

         // Wait for incoming requests
         boa->impl_is_ready();
      }
      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.

Notice that when the AccountManager object is activated, the _this method is used, which will return the first object wrapper in the chain. When activating an object implementation or invoking a method it offers, the server must use the _this method. Failing to do so will circumvent any object wrappers that may be registered.

Removing typed object wrappers

The `<interface_name>::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.

See Chapter 4, "Generated classes," in the *VisiBroker for C++ Reference* for more information.

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 been created. Only subsequently created and object implementation will be affected.

Combined use of un-typed and typed object wrappers

If you choose to use both typed and un-typed object wrappers in your application, all `pre_method` methods defined for the un-typed wrappers will be invoked prior to any typed object wrapper methods defined for an object. Upon return, all typed object wrapper methods defined for the object will be invoked prior to any `post_method` methods defined for the un-typed wrappers.

The `objwrap` sample applications **client.C** and **server.C** make use of a sophisticated design that allows you to use command-line properties to specify which, if any, typed and un-typed object wrappers are to be used.

Command-line arguments for typed wrappers

Table 22.3 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**.

Table 22.3 Command-line arguments for the controlling typed object wrappers

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.

Table 22.3 Command-line arguments for the controlling typed object wrappers (continued)

Bank wrappers properties	Description
-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 **BankWrappers/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.

Code sample 22.8 shows how the initializer uses a set of PropStruct objects to track the command-line options that have been specified and then add or remove AccountObjectWrapper objects for the appropriate locations.

Code sample 22.8 Initializer for a typed object wrapper

```
static const CORBA::ULong kNumTypedAccountProps = 2;
static PropStruct TypedAccountProps[kNumTypedAccountProps] =
{ { "BANKaccountCacheClnt", CachingAccountObjectWrapper::factory,
      VISObjectWrapper::Client },
  { "BANKaccountCacheSrvr", CachingAccountObjectWrapper::factory,
      VISObjectWrapper::Server }
};
static const CORBA::ULong kNumTypedAccountManagerProps = 4;
static PropStruct TypedAccountManagerProps[kNumTypedAccountManagerProps] =
{ { "BANKmanagerCacheClnt", CachingAccountManagerObjectWrapper::factory,
      VISObjectWrapper::Client },
  { "BANKmanagerSecurityClnt", SecureAccountManagerObjectWrapper::factory,
      VISObjectWrapper::Client },
  { "BANKmanagerCacheSrvr", CachingAccountManagerObjectWrapper::factory,
      VISObjectWrapper::Server },
  { "BANKmanagerSecuritySrvr", SecureAccountManagerObjectWrapper::factory,
      VISObjectWrapper::Server },
};

void BankInit::update(int& argc, char* const* argv)
{
   if (argc > 0) {
      init(argc, argv, "-BANK");
      CORBA::ULong i;
      for (i=0; i < kNumTypedAccountProps; i++) {
            CORBA::String_var arg(property_value(TypedAccountProps[i].propname));
            if (arg && strlen(arg) > 0) {
                  // clear the current setting since we are now using it
                  replace_value(TypedAccountProps[i].propname, "");
                  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(
               property_value(TypedAccountManagerProps[i].propname));
            if (arg && strlen(arg) > 0) {
                  // clear the current setting since we are now using it
                  replace_value(TypedAccountManagerProps[i].propname, "");
                  if (atoi((char*) arg)) {
                        Bank::AccountManagerObjectWrapper::add(_orb,
                              TypedAccountManagerProps[i].fact,
                              TypedAccountManagerProps[i].loc);
                  }  else {
                        Bank::AccountManagerObjectWrapper::remove(_orb,
                            TypedAccountManagerProps[i].fact,
                            TypedAccountManagerProps[i].loc);
                  }
            }
      }
   }
}
```

Command-line arguments for un-typed wrappers

Table 22.4 shows the command-line arguments you can use to enable the use of un-typed object wrappers for the sample bank applications implemented in **client.C** and **server.C**.

Table 22.4 Command-line arguments for controlling un-typed object wrappers

Bank wrappers properties	Description
-TRACEWRAPclient <numwraps>	Instantiates the specified number of un-typed object wrapper factories for tracing wrappers for a client application.
-TRACEWRAPserver <numwraps>	Instantiate the specified number of un-typed object wrapper factories for tracing on a server application.
-TRACEWRAPboth <numwraps>	Instantiate the specified number of un-typed object wrapper factories for tracing for both a client and server application.
-TIMINGWRAPclient <numwraps>	Instantiate the specified number of un-typed object wrapper factories for timing on a client application.

Table 22.4 Command-line arguments for controlling un-typed object wrappers (continued)

Bank wrappers properties	Description
-TIMINGWRAPserver <numwraps>	Instantiate the specified number of un-typed object wrapper factories for timing on a server application
-TIMINGWRAPboth <numwraps>	Instantiate the specified number of un-typed object wrapper factories for timing on both a client and a server application.

Initializers for un-typed wrappers

The un-typed wrappers are created and registered in the `TraceWrapInit::update` and `TimingWrapInit::update` methods, defined in **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 un-typed object wrappers.

Code sample 22.9 shows a portion of the **tracewrap.C** file, which will install the appropriate un-typed object wrapper factories, based on the command-line properties you specify.

Code sample 22.9 Initializer for an un-typed object wrapper

```
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 arg(property_value(propname));
```

```
            if (arg && strlen(arg) > 0) {
                  // clear the current setting since we are now using it
                  replace_value(propname, "");
                  int numNew = atoi((char*) arg);
                  char key_buf[256];
                  for (CORBA::ULong j=0; j < numNew; j++) {
                     sprintf(key_buf, "%s-%d", propname, list->size());
                     list->add(new TraceObjectWrapperFactory(loc,
                           (const char*) key_buf));
                  }
            }
         }
      }
   }
```

Executing the sample applications

Before executing the sample applications, make sure that an osagent is running on your network. You can then execute the server application without any tracing or timing object wrappers by using the command

Example

```
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 command

Example

```
prompt> client John
```

You can also execute this command if want a default name to be used.

Example

```
prompt> client
```

Turning on timing and tracing object wrappers

To execute the client with un-typed timing and tracing object wrappers enabled, use this command:

Example

```
prompt> client -TRACEWRAPclient 1 -TIMINGWRAPclient 1
```

To execute the server with un-typed wrappers for timing and tracing enabled, use this command:

Example

```
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:

Example

```
prompt> client -BANKaccountCacheClnt 1 _BANKmanagerCacheClnt 1 \ -BANKmanagerSecurityClnt
```

To execute the server with typed wrappers for caching and security enabled, use this command:

Example

```
prompt> server -BANKaccountCacheSrvr 1 -BANKmanagerCacheSrvr 1 \ -BANKmanagerSecuritySrvr 1
```

Turning on typed and un-typed wrappers

To execute the client with all typed and un-typed wrappers enabled, use this command:

Example

```
prompt> client -BANKaccountCacheClnt 1 -BANKmanagerCacheClnt 1 \
   -BANKmanagerSecurityClnt 1 \ -TRACEWRAPclient 1 -TIMINGWRAPclient 1
```

To execute the server with all typed and un-typed wrappers enabled, use this command:

Example

```
prompt> server BANKaccountCacheSrvr 1 BANKmanagerCacheSrvr 1 \
   -BANKmanagerSecuritySrvr 1 \ -TRACEWRAPclient 1 -TIMINGWRAPclient 1
```

Executing a co-located client and server

The following command will execute a co-located server and client with all typed wrappers enabled, the un-typed wrapper enables for just the client, and the un-typed tracing wrapper for just the server, use this command:

Example

```
prompt> server -BANKaccountCacheClnt 1 -BANKaccountCacheSrvr 1 \
   -BANKmanagerCacheClnt 1 -BANKmanagerCacheSrvr 1 \ -BANKmanagerSecurityClnt 1
   -BANKmanagerSecuritySrvr 1 \ -TRACEWRAPserver 1 -TIMINGWRAPclient 1
```

Chapter

23

Using the ORB management interface

This chapter explains how to use the ORB management interfaces to query and set server properties. This chapter includes the following major sections:

Overview

The VisiBroker ORB management interface allows your client applications to monitor and manage the properties of object servers. Many of the attributes you can manage are similar to the ORB and BOA options described in Appendix A, "Using command-line options," of the *VisiBroker for C++ Reference*. These attributes include items such as

- Current number of client connections
- Maximum number of connections allowed
- Send buffer size
- Receive buffer size
- Connection time-out value

Sample client application

Several sample client applications are provided in the VisiBroker distribution in the directory **examples/orbmgr**. These example applications will be used to illustrate the features of the ORB Management interfaces.

Clients created with other CORBA environments

Client and server applications created with VisiBroker for C++ automatically have access to ORB Management interfaces when they link with the VisiBroker library. It is possible for a client application created with another vendor's CORBA product to manage a server application developed with Visibroker by following these steps.

1. Use the IDL compiler provided with the non-VisiBroker product to generate C++ source code from the **vorbmgr.idl** file, located in the **idl** directory of your VisiBroker distribution.
2. Compile the client stub source code generated in step 1.
3. Link the client stubs with your client application.

Understanding attributes

An attribute represents a property for a server application. Some attributes can only be retrieved while others can be retrieved or set. Code Sample 23.2 shows the IDL definition for the `Attribute` structure. The `id` member uniquely identifies the attribute. The `value` member contains the attribute's current setting or state, stored in an `Any` object. The `read_only` member indicates whether the attribute can be set or if it is read-only.

Code sample 23.1 IDL definition for the Attribute structure

```
struct Attribute {
   string id;
   any value;
   boolean is_readonly;
};
```

Getting and setting attributes

The abstract base class `AttributeSet` provides methods for obtaining and setting the value of attributes. It is used to derive the `Server` and `Adapter` classes that your client application will use to manage server applications.

Server and adapter classes

The `Server` class provides methods for obtaining all of the object adapters offered by a server application, getting the server's process id, obtaining the activation policy, and shutting down the server application.

Many of the attributes offered by the `Adapter` class are available as BOA command-line options, which are described in Appendix A, "Using command-line options," of the *VisiBroker for C++ Reference*.

The `Adapter` classes also offer attributes that represent run-time states or conditions, such as the number of currently active connections. These types of attributes have no corresponding BOA command-line options.

Furthermore, there are some BOA command-line options for which there are no corresponding `Adapter` attributes.

Getting attributes

The `AttributeSet::get_attribute` method allows you to obtain the value of an attribute, given its name. The names of the attributes and their data types are described in Table 23.3, Table 23.4, and Table 23.5.

Setting attributes

The `AttributeSet::set_attribute` method allows you to set the value of an attribute. When setting an attribute, you specify the attribute's identifier and the new value. Some attributes are read-only and cannot be set. Attempting to set a read-only attribute will cause an `AttributeReadOnly` exception to be thrown. See Table 23.3, Table 23.4, and Table 23.5 for a description of the attributes and their data types.

Note If a server has not been started with the `-ORBsecureSetAttr 0` option, any attempt by a client application to set an attribute for that server will result in a `CORBA::NO_PERMISSION` exception being thrown.

Using the Server class

The `Server` class gives your client application access to a set of read-only and read-write ORB-style attributes for a server application.

Server methods

The Server class provides the following methods:

Table 23.1 Methods offered by the Server class

Method	Description
get_adapter	Returns an Adapter object that can be used to query and set BOA-style attributes.
activation_policy	Returns the Server's activation policy.
get_all_adapters	Returns all the adapters used by the server application
process_id	Returns the process identifier associated with the server application.
shutdown	Causes the server application to shutdown.

Obtaining a server reference

Code Sample 23.2 shows a portion of the **oskill.C** client application that illustrates how to obtain a Server object reference. In order to obtain a Server reference, the example client application follows these steps:

1. Bind to an object offered by the server to be managed.
2. Using the object reference obtained in step 1, invoke the Object::_resolve_reference method, specifying the identifier ORBManager.
3. Use the narrow method to narrow the object reference to an ORBManager::Server type.

Code sample 23.2 Obtaining a Server object reference

```
. . .
int main(int argc, char** argv)
{
   ORBManager::Server_var srv;
   . . .

   try {
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
      CORBA::Object_var obj;
      . . .
            obj = orb->bind(repid, name, host);
      . . .
      obj = obj->_resolve_reference("ORBServer");

      ORBManager::Server_var srv;
      srv = ORBManager::Server::_narrow(obj);
      . . .
```

Listing All Attributes

Code Sample 23.3 shows a portion of the **inspect.C** client application that illustrates how to get a list of all the attributes offered by a Server. To list all of the attributes offered by an Adapter, follow the same procedure but use an Adapter object reference instead.

Code sample 23.3 Listing all attributes for a Server object

```
int main(int argc, char** argv)
{
   ORBManager::Server_var srv;
   ORBManager::ObjectSeq_var objs;
   . . .
   try {
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
      CORBA::Object_var obj;
      . . .
            obj = orb->bind(repid, name, host);
      . . .
      //
      // obj = obj->_resolve_reference("ORBServer");
      //
      obj = dii_resolve_reference(obj, "ORBServer");

      ORBManager::Server_var srv;
      srv = ORBManager::Server::_narrow(obj);
      . . .
      ORBManager::AttributeSeq_var    attrs;
      . . .
      attrs = srv->get_all_attributes();
      . . .
```

Shutting down a server application

Code Sample 23.4 shows a portion of the **oskill.C** client application that illustrates how a client application can shutdown a server application using a `Server` object reference. After successfully invoking this method, any object references the client has to that server will no longer be valid.

Note If a server has not been started with the `-ORBsecureShutdown 0` option, any attempt by a client application to invoke the `shutdown` method for that server will result in a `CORBA::NO_PERMISSION` exception being thrown.

Code sample 23.4 Using the Server object's shutdown method

```
   int main(int argc, char** argv)
{
   ORBManager::Server_var srv;
   . . .
   try {
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
      CORBA::Object_var obj;

      . . .
            obj = orb->bind(repid, name, host);
   . . .
      obj = obj->_resolve_reference("ORBServer");

      ORBManager::Server_var srv;
      srv = ORBManager::Server::_narrow(obj);
```

```
        . . .

        try {
           cout << "Please wait ..." << endl;
           cout.flush();
           srv->shutdown();
        }
        . . .
        catch(const CORBA::NO_PERMISSION&) {
           cerr << "No permission" << endl;
           return 1;
        }

    . . .
    cout << "shutdown successed" << endl;
    return 0;
}   . . .
```

Using the Adapter class

The `Adapter` class gives your client application access to a set of read-only and read-write BOA-style attributes for a server application. The read-only attributes can be retrieved, but not set. The read-write attributes can be retrieved and set.

Adapter methods

Since the `Adapter` class is derived from the `AttributeSet` class, it inherits the `get_attribute` and `set_attribute` methods. It adds the following methods:

Table 23.2 Methods offered by the Server class

Method	Description
`adapter_id`	Returns the Adapter object's; either, `TSingle`, `TPool`, or `TSession`.
`get_all_attributes`	Returns all of the Adapter object's attributes.
`get_attribute`	Returns the value of an `Adapter` attribute.
`persistent_objects`	Returns all the currently available persistent objects for the adapter.
`set_attribute`	Sets the value of an `Adapter` attribute.
`shutdown`	Causes the server application to shutdown.

Obtaining an Adapter reference

Code sample 23.5 shows a portion of the **setattr.C** application that obtains an Adapter reference. The sample client application follows these steps:

1 Obtain a `Server` object reference, as described in "Obtaining a server reference" on page 23-4.

2 Use the Server object reference obtained in step 1 to invoke the Server::get_all_adapters method. You may also use the Server::get_adapter and specify the particular adapter in which you are interested (TSingle, TPool, or TSession).

Code sample 23.5 Getting all Adapter objects offered by a Server

```
. . .
obj = dii_resolve_reference(obj, "ORBServer");
ORBManager::Server_var srv;
srv = ORBManager::Server::_narrow(obj);

if( srv->activation_policy() != extension::SHARED_SERVER ) {
   cerr << "Target is not a shared server" << endl;
   return 0;
}

// get all adapters
ORBManager::AdapterSeq_var adps;
adps = srv->get_all_adapters();
. . .
```

Getting and setting attributes

Code sample 23.6 shows a portion of the **setattr.C** client application that illustrates how to obtain and set an attribute. See page 23-8 for a list of Adapter attributes. This client application follows these steps:

1 Obtain an object reference, using the dii_resolve_reference method.
2 Narrow the object reference to an ORBManager::Server type.
3 Use the Server::get_all_adapters to obtain one or more Adapter object references.
4 Use the get_attribute method to obtain an attribute.
5 Use the set_attribute method to set the attributes value.

Note Attempting to set a read-only attribute will cause an AttributeReadOnly exception to be thrown.

Code sample 23.6 Getting and setting an Adapter attribute

```
. . .
obj = dii_resolve_reference(obj, "ORBServer");
ORBManager::Server_var srv;
srv = ORBManager::Server::_narrow(obj);

if( srv->activation_policy() != extension::SHARED_SERVER ) {
   cerr << "Target is not a shared server" << endl;
   return 0;
}

// get all adapters
ORBManager::AdapterSeq_var adps;
adps = srv->get_all_adapters();

ORBManager::Adapter_ptr adp;
unsigned short adp_count = adps->length();
```

```
ORBManager::Attribute attr;

cout << endl;

for(unsigned short i=0;i<adp_count;i++) {
   adp = adps->operator[](i);

   CORBA::Any_var val;

   val = adp->get_attribute(attrid);

   CORBA::Any ival = *(CORBA::Any_ptr)val;
   str2attr(value, ival);

   adp->set_attribute(attrid, ival);
   . . .
```

Listing all Adapter attributes

The procedure for listing all of the `Adapter` attributes is the same as the procedure used to list all `Server` attributes, described in "Listing All Attributes" on page 23-4. The list of attribute identifiers for `Adapter` objects returned will be different from those returned for `Server` objects.

Adapter Attributes

Table 23.3 Common Attributes for TSingle, TPool, and TSession adapters

Attribute	Type	Description
`OAactivatedConnections`	read-only unsigned long	Current number of active client connections
`OAactivatedRequests`	read-only unsigned long	Current number of outstanding operations requests.
`OArcvbufsize`	unsigned long	Specifies the size of the buffer (in bytes) used to receive messages. If not specified, a default value (dependent upon your operating system) will be used.
`OAsendbufsize`	unsigned long	Specifies the size of the buffer (in bytes) used to receive messages. If not specified, a default value (dependent upon your operating system) will be used.
`OAtcpnodelay`	boolean	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.

Table 23.4 Attributes for TPool and TSession adapters

Attribute	Type	Description
OAthreadStackSize	read-only unsigned long	Specifies the maximum thread stack size (in bytes) allowed.

Table 23.5 Attributes for TPool adapters

Attribute	Type	Description
OAallocatedThreads	read-only unsigned long	Current number of allocated threads.
OAconnectionMax	unsigned long	Maximum number of incoming connections allowed.
OAconnectionMaxIdle	Read-Write	Number of seconds that connections are allowed to be idle before being shutdown. A value of 0 means that connections will never time-out.
W OAshmsize	read-only unsigned long	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.
OAthreadMax	unsigned long	Maximum number of threads allowed.
OAthreadMaxIdle	unsigned long	Number of seconds a thread can exist without servicing any requests before it is returned to the system.
OAthreadMin	unsigned long	Minimum number of threads allowed.

Part

III

Appendixes

This part of the *VisiBroker for C++ Programmer's Guide* includes these appendixes.

Appendix A Using VisiBroker for C++ 3.3 with other VisiBroker ORBs

Appendix B CORBA exceptions

Appendix C Handling ORB communication events

Appendix

A

Using VisiBroker for C++ 3.3 with other VisiBroker ORBs

This appendix discusses compatibility issues with using VisiBroker for C++ version 3.3 with VisiBroker for Java, with VisiBroker for C++ version 2.0, and with other ORBs. Other backward-compatibility issues are also covered. This appendix includes these major sections:

Transitioning from VisiBroker for C++ 2.0

This release of VisiBroker for C++ introduces many new features, while still maintaining backward compatibility with version 2.0 of VisiBroker for C++ and all releases of VisiBroker for Java. Please read the following sections carefully.

Source level compatibility

Source level compatibility with VisiBroker for C++ version 2.0 has been maintained, but you must regenerate, recompile, and relink your code.

Name changes

The names of several commands, environment variable, directories, source files, and module names have been changed to ensure consistency with other VisiBroker products. Table A.1 summarizes the name changes.

Table A.1 Name changes from VisiBroker for C++ 2.0

Type	Old name	New name
Environment	ORBELINE	VBROKER_ADM
Variable	ORBELINE_IMPL_PATH	VBROKER_IMPL_PATH
Default install directory	orb20 or Orbeline2.0	vbroker
Tools	orbeline	idl2cpp
	regobj	oadutil reg
	unregobj	oadutil unreg
	listimpl	oadutil list
Modules	PMC_EXT	VIS_EXT
Source Files	pmcext.h	vext.h

Ensuring backward compatibility for VisiBroker for C++ 2.0 applications

The `-ORBbackcompat` flag configures the runtime to be compatible with VisiBroker for C++ 2.0 clients and servers. Pass '`-ORBbackcompat 1`' when deploying servers or clients in a VisiBroker for C++ 2.0 environment.

This flag causes the runtime to do the following:

- Double-register with the Smart Agent using the interface name (as in version 2.0), as well as using the repository ID (as in version 3.3). Clients will use interface name to locate providers.
- Typecode encoding complies with the CDR encoding from version 2.0.
- If an object is unnamed, backward compatibility will register with a pseudo object name. In version 3.3, all unnamed objects are locally-scoped (as in VisiBroker for Java).
- Sets '`-ORBbackdii`', which complies with the 1.0 IDL to C++ mapping for memory management.
- Sets the default multithreaded BOA to `TSession` (thread-per-session). Otherwise, it is the `TPool` (thread pool) BOA.
- Sets '`-ORBnullstring 1`', which allows `NULL` strings to be passed on the wire.

Note For backward compatibility with version 2.0 of the `idl2cpp` compiler, an `orbeline` executable has also been included which understands the version 2.0 compiler flags.

Passing command line arguments

In version 3.3, passing arguments to the ORB or BOA has been modified to be standards compliant and allow more consistent parsing. In the past, arguments were passed with `<parameter>=<value>`. In version 3.3, this has changed to `<parameter> <value>`. Passing version 2.0 parameters with the equals sign has been maintained for compatibility, but version 3.3 parameters (like `-OAid`) and version 2.0 parameters are parsed in the new style. Thus the command line in version 2.0 that reads

```
server -ORBagentaddr=201.34.53.83 -OAport=12000
```

can now read

```
server -ORBagentaddr 201.34.53.83 -OAport 12000
```

NT services

NI The `osagent` and `oad` are now Windows NT services, making VisiBroker easier to set up and administer under Windows NT. These tools may also be used in console mode, as they were provided in version 2.0. See the *VisiBroker for C++ Reference* for complete details.

Windows registry integration

W Environment variables used by the VisiBroker runtime are added to the Windows registry when you install VisiBroker, eliminating the need for setting environment variables.

Enhanced thread and connection management

In version 2.0, threads were created based on client connections. In version 3.3, two different thread policies are available—thread-per-session and thread pooling. The thread-per-session policy provides the same thread management as provided in version 2.0. Thread pooling, however, allows threads to be assigned from a thread pool to handle individual client operation requests. The management of connections has also been improved in this release to increase performance and reduce overhead. Thread policies and connection management are described in Chapter 8, “Managing threads and connections.”

Interceptors

Communication event handlers have been replaced by interceptors, which provide more information during ORB communication. See Chapter 16, “Instrumenting and modifying the ORB with interceptors,” for information about interceptors. See Appendix C, “Handling ORB communication events,” for more information about communication event handlers.

Extended data type support

VisiBroker 3.3 supports the following extended data types.

IDL	Solaris	Windows 95	Windows NT
longlong	X	X	X
unsigned longlong	X	X	X
long double	X	—[1]	—[1]
wchar	—[2]	—	X
wstring	—[2]	—	X

1. Long double is a 16 byte type. Since the native long double on Windows is an 8 byte type, the IDL long double is not supported on the Windows platform.
2. Solaris 2.5 and 2.5.1 does not support Unicode, which VisiBroker uses on the wire when sending wchars and wstrings. Attempting to send wchars and wstrings on Solaris will result in a DATA_CONVERSION exception.

Additional new features with VisiBroker 3.3

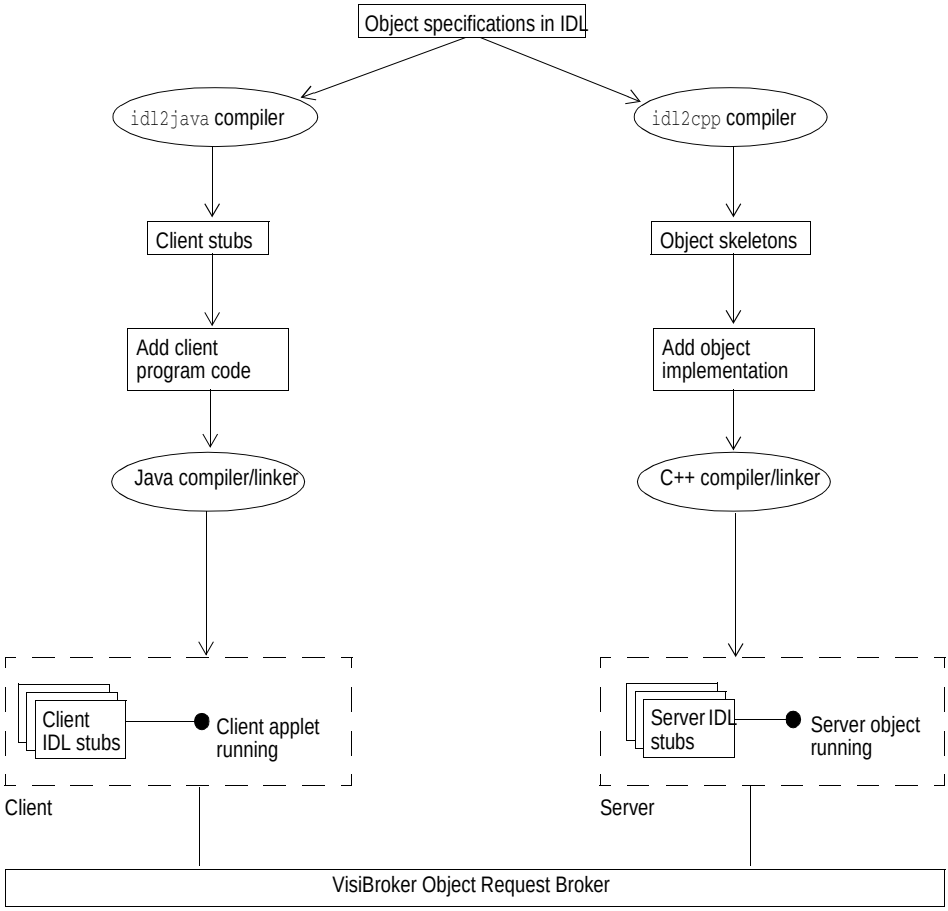
VisiBroker version 3.3 has several additional new features:

- Improved Location Service with triggers as described in Chapter 19, "Discovering object instances using the Location Service."
- Improved `idl2cpp` compiler as described in the *VisiBroker for C++ Reference*.
- Packaging of the Interface Repository.
- Customization features for the ORB with smart stubs as described in Chapter 17, "Customizing remote object invocations using smart stubs."
- Support for Dynamic Invocation (DII) when using in-process calls.
- Utilities for printing version information as described in the *VisiBroker for C++ Installation and Administration Guide*.

Interoperability with VisiBroker for Java

Clients and servers developed with VisiBroker for C++ are completely interoperable with clients and servers created with VisiBroker for Java, which is packaged separately. You can use the same IDL file as input to both the `idl2cpp` compiler and the `idl2java` compiler supplied with VisiBroker for Java. Figure A.1 shows a Java applet used with a C++ server. You can also create C++ clients that work with Java servers.

Figure A.1 Creating an applet with VisiBroker for Java and a server with VisiBroker for C++



Since C++ objects are registered with the Smart Agent, and the Gatekeeper communicates with the Smart Agent, Java clients can communicate directly with C++ objects without additional effort. See the *VisiBroker for Java Programmer's Guide* or the *VisiBroker for Java Gatekeeper Guide* for further details on the Gatekeeper.

Use of Java in the C++ utilities

A few of the utilities of VisiBroker for C++ have been written in Java to provide better consistency across platforms. They are

- `idl2cpp` compiler
- Interface Repository (`idl2ir`, `irep`)
- OAD registration utilities (`oadutil`)

The Java classes for these utilities are bundled in the **vbcpp.jar** file, located in the **<install_dir>/lib** directory. The use of Java is transparent to the user since VisiBroker comes bundled with JavaSoft's Java Runtime Environment (JRE).

Note For VisiBroker for Java users, the use of the JRE will not affect the choice of the VisiBroker for Java virtual machine.

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.

Appendix

B

CORBA exceptions

This appendix provides information about CORBA exceptions that can be thrown by the VisiBroker ORB, and explains possible causes for VisiBroker throwing them.

The following table lists CORBA exceptions, and explains reasons why the VisiBroker ORB might throw them.

Table B.1 CORBA exceptions and possible causes

Exception	Explanation	Possible causes
`CORBA::BAD_CONTEXT`	Not thrown by VisiBroker.	
`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::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.
`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.
`CORBA::BAD_PARAM`	A parameter passed to the 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.

Table B.1 CORBA exceptions and possible causes (continued)

Exception	Explanation	Possible causes
		ORB and BOA calls—such as `BOA::obj_is_ready`—throw this exception if an invalid `Object_ptr` is passed as an `in` argument (for example, if a `nil` reference is passed). An attempt may have been made to send a 0-length (`NULL`) string. To enable the sending and receiving of `NULL` strings, pass the command-line argument "`-ORBnullstring 1`" to the program that calls `ORB_init()`. 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`. An attempt was made to send a value that is out of range for an enumerated data type. The `BOA::create()` or `BOA::change_implementation()` methods may have been called with an `ImplementationDef_ptr` that is either set to `NULL`, or is not a `CreationImplDef`. An attempt may have been made to construct a `TypeCode` with an invalid `kind` value. An attempt may have been made to insert a `nil` object reference into an `Any`. An attempt may have been made to register a smart stub for a type that is not known. Ensure that the repository ID passed to the base smart stub constructor is valid. You may not pass the IDL name to this constructor—the generated repository ID must be passed instead. An `IOR` may have been specified via the `-ORBir_ior` command-line argument which failed to narrow to a `CORBA::Repository`. Using the DII and one way method invocations, an `OUT` argument may have been specified.
`CORBA::BAD_TYPECODE`	Not thrown by VisiBroker.	
`CORBA::COMM_FAILURE`	A communication failure has occurred.	An existing connection may have closed due to failure at the other end of the connection. A new connection request may have failed due to resource limits on the client or server machine. When `COMM_FAILURES` 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::DATA_CONVERSION`	Not thrown by VisiBroker.	

Table B.1 CORBA exceptions and possible causes (continued)

Exception	Explanation	Possible causes
CORBA::FREE_MEM	Not thrown by VisiBroker.	
CORBA::IMP_LIMIT	Not thrown by VisiBroker.	
CORBA::INITIALIZE	A necessary initialization has not been performed.	The ORB_init() or BOA_init() methods may not have been called. All clients must call the ORB_init() method prior to performing any ORB-related operations. Any server that wishes to export objects or generate object references must call the BOA_init() method. These two calls are typically made immediately upon program startup at the top of the main routine.
CORBA::INTERNAL	An internal ORB error has occurred.	An internal ORB error may have occurred.
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	Not thrown by VisiBroker.	
CORBA::INV_IDENT	Not thrown by VisiBroker.	
CORBA::INV_OBJREF	An invalid object reference has been encountered.	The ORB will throw this exception if an object reference is obtained that contains no usable profiles. For example, if an object reference that only supports SSL is passed to a client that only understands IIOP, the IIOP ORB will throw CORBA::INV_OBJREF. The ORB::string_to_object() method will throw this exception if the stringified object reference does not begin with the characters "IOR:".
CORBA::MARSHAL	An invalid message has been received by the BOA.	The data passed to VisiBroker was improperly initialized. For example, the length was set improperly in a sequence. The message received by the server may have been truncated. A string with length 0 may have been received. To enable the sending and receiving of 0-length strings, pass the command-line argument "-ORBnullstring 1" to the program that calls the ORB_init() method. If any exception occurs when attempting to read a TypeCode from a marshalled input stream, CORBA::MARSHAL will be thrown.

Table B.1 CORBA exceptions and possible causes (continued)

Exception	Explanation	Possible causes
CORBA::NO_IMPLEMENT	The requested object could not be located.	A _bind() call or some other remote operation fails because the target could not be found. To dynamically locate implementations through the VisiBroker bind() call, a Smart Agent must be running in your ORB domain. In addition, an implementation of the requested interface must be available on the same ORB domain. To verify the presence of a Smart Agent, run the osfind utility. This utility prints the locations of all Smart Agents on your current domain (that is, all Smart Agents listening on your environment's OSAGENT_PORT). The osfind utility will also print the interface name and instance name of all available implementations. In summary, prior to running the client program, 1. Verify that a Smart Agent is running and accessible on the network. 2. Verify that the desired implementation is available on the network. If the rebind() method is enabled and an object implementation becomes unavailable, NO_IMPLEMENT will be thrown if another provider cannot be located.
CORBA::NO_MEMORY	Not thrown by VisiBroker.	
CORBA::NO_PERMISSION	An operation has been requested for which the client has no permissions.	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.
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.
CORBA::NO_RESPONSE	No response was returned before a time-out occurred.	If BindOptions are used to set time-outs, this exception is raised when send and receive calls do not occur within the specified time.
CORBA::OBJ_ADAPTER	Not thrown by VisiBroker.	
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.

Table B.1 CORBA exceptions and possible causes (continued)

Exception	Explanation	Possible causes
		The `VISDictionary` classes throw this exception if an attempt is made to access an element that does not exist. To ensure that the requested entry exists in the dictionary, call the `includesKey()` method.
`CORBA::PERSIST_STORE`	Not thrown by VisiBroker.	
`CORBA::TRANSIENT`	An error has occurred, but the ORB believes it is possible to retry the operation. This exception is used internally by the ORB, and should not be seen by client code.	A communications failure may have occurred and the ORB is signalling that an attempt should be made to rebind to the server with which communications have failed. This exception will not occur if the `BindOptions` are set to false with the `enable_rebind()` method. If a server returns a `LOCATION_FORWARD` reply to a request, the ORB will reconnect to the new location and throw this exception to indicate that the request should be resent.
`CORBA::UNKNOWN`	The ORB could not determine the thrown exception.	The server throws something other than a correct exception such as a Windows 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`.
`CORBA::UnknownUser Exception`	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 ORB has no way to unmarshal the exception on its own.

Appendix

C

Handling ORB communication events

VisiBroker provides a set of communication event handling mechanisms for notifying clients and object implementations of ORB events. This section describes how you can create communication event handlers to implement accounting, tracing, debugging, logging, security, or encryption features. Communication event handlers are provided in this release for backward compatibility with VisiBroker version 2.0. The preferred feature for these types of purposes is the interceptor. See Chapter 16, "Instrumenting and modifying the ORB with interceptors," for details.

This appendix includes the following major topics:

Understanding and using communication event handlers

Communication event handlers objects allow client programs and object implementations to define methods that the ORB will invoke to handle ORB events such as the success or failure of a bind request or the failure of an object implementation. Two different communication event handler classes are provided because the types of ORB events that can be handled are different for clients and object implementations. The procedure for using a communication event handler, however, is similar for clients and object implementations.

1. Derive a communication event handler class for your object, defining the event methods you wish to handle.
2. Provide implementations for the event methods you wish to handle.

3 Add code to the client or object implementation to register the communication event handler.

Table C.1 ORB events that can be handled by client applications and object implementations

Client-side ORB events	Implementation-side ORB events
Bind succeeded	Bind request received
Bind failed	Unbind request received
Server aborted	Client aborted
Rebind succeeded	Pre-method
Rebind failed	Post-method

Client communication event handlers

Your client programs can register one or more communication event handler objects with the ORB to handle ORB events for a particular ORB object. Your client can also globally register one or more event handlers to handle ORB events for all ORB objects the client uses.

Code sample C.1 shows a portion of the **vext.h** include file that contains the class definition for the `ClientEventHandler` class.

Code sample C.1 ConnectionInfo structure and the ClientEventHandler class

```
class VIS_EXT
{
      struct ConnectionInfo {
            CORBA::String_var hostname;
            CORBA::UShort port;
            CORBA::Long fd;
            ...
      };

      class ClientEventHandler
      {
            public:
               virtual void bind_succeeded(CORBA::Object_ptr,
                        const ConnectionInfo&);
               virtual void bind_failed(CORBA::Object_ptr);
               virtual void server_aborted(CORBA::Object_ptr);
               virtual void rebind_succeeded(CORBA::Object_ptr,
                        const ConnectionInfo&);
               virtual void rebind_failed(CORBA::Object_ptr);
               ...
      };
      ...
};
```

Providing connection information

This structure represents all the information needed for a connection. It includes the host name where the object implementation resides, the port number, and the file descriptor used for the connection. This structure is modified when the connection to the object implementation is lost and a rebind operation is attempted.

Interpreting event communication from the ORB

When a communication event handler object has been registered for a particular ORB object, the ORB will call the `ClientEventHandler` methods when a specific event occurs. If the event handler is registered as a global event handler, the `ClientEventHandler` methods will be invoked for any event related to any object the client uses.

The `bind_succeeded()` method is invoked by the ORB when the client's request to bind to the ORB object has completed successfully. A pointer to the bound object is provided as a parameter as well as the connection information.

The `bind_failed()` method is invoked if the client's bind request fails. A pointer to the object to which the event is related is provided as a parameter.

The `server_aborted()` method is invoked if the connection to the object implementation is lost. A pointer to the object to which the event is related is provided as a parameter.

The `rebind_succeeded()` method is invoked when an attempt to reconnect to an object implementation succeeds. A pointer to the object that has been rebound is provided as a parameter as is the new connection information.

The `rebind_failed()` method is invoked when an attempt to reconnect to an object implementation fails. A pointer to the object to which the event is related is provided as a parameter.

Creating a client communication event handler

To implement a communication event handler for your client program, you must derive your own event handler class for the class you wish to monitor. You will need to implement only those event handler methods you wish to override. If you do not override a communication event handler method, no special processing will occur and no performance overhead will be added to the program.

Code sample C.2 shows how you would define a communication event handler for the account client, introduced in Chapter 4, "Quick start for development with VisiBroker for C++." Only three of the possible five methods offered by `ClientEventHandler` have been overridden and Code sample C.3 shows simple implementations for these methods.

Code sample C.2 Example communication event handler for the account client

```
class AccountClientHandler : public VIS_EXT::ClientEventHandler {
      ...
      public:
            void bind_succeeded(CORBA::Object_ptr, const ConnectionInfo&);
            void bind_failed(CORBA::Object_ptr);
            void server_aborted(CORBA::Object_ptr);
};
```

Code sample C.3 Implementation for the AccountClientHandler method

```
void AccountClientHandler::bind_succeeded(CORBA::Object_ptr obj,
            const ConnectionInfo&) {
         cout << "Event Handler bind_succeeded for: "
                  << obj->_interface_name() << endl;
}

void AccountClientHandler::bind_failed(CORBA::Object_ptr obj) {
         cout << "Event Handler bind_failed for: "
                  << obj->_interface_name() << endl;
}

void AccountClientHandler::server_aborted(CORBA::Object_ptr obj) {
         cout << "Event Handler server_aborted for: "
                  << obj->_interface_name() << endl;
}
```

Registering communication event handlers with the handler registry

You can use the static `HandlerRegistry::instance()` method to obtain a pointer to the registry and then invoke the methods for registering and un-registering various types of communication event handlers.

Code sample C.4 HandlerRegistry class

```
class HandlerRegistry{
      ...
      public:
            ...
            static HandlerRegistry_ptr instance();
            void reg_obj_client_handler(CORBA::Object_ptr obj,
                        ClientEventHandler_ptr handler);
            void reg_glob_client_handler(ClientEventHandler_ptr handler);
            void unreg_obj_client_handler(CORBA::Object_ptr obj);
            void unreg_glob_client_handler();
            void reg_obj_impl_handler(CORBA::Object_ptr obj,
                        ImplEventHandler_ptr handler);
            void reg_glob_impl_handler(ImplEventHandler_ptr handler);
            void unreg_obj_impl_handler(CORBA::Object_ptr obj);
            void unreg_glob_impl_handler();
            ...
};
```

Invoking HandlerRegistry methods from client programs

The `reg_obj_client_handler()` method is invoked by your client program to register a communication event handler for a specific object. The parameters passed to this method are a reference to the object and a pointer to the object's `ClientEventHandler`. If the object reference is not valid, an `InvalidObject` exception will be raised. If the communication event handler has already been registered for the specified object, a `HandlerExists` exception will be raised. You can use the `unreg_obj_client_handler()` method to un-register a previously registered communication event handler.

The `reg_glob_client_handler()` method can be invoked by your client program to register a communication event handler for all object the client uses. The parameter passed to this method is a pointer to the object's `ClientEventHandler`. If a global handler has already been registered, a `HandlerExists` exception will be raised. You can use the `unreg_glob_client_handler()` method to un-register a previously registered global communication event handler.

Note If both an object communication event handler and a global communication event handler are registered, the object communication event handler will take precedence for ORB events that occur which are related to its object. All other ORB events will be handled by the global communication event handler.

The `unreg_obj_client_handler()` method can be invoked by your client program to unregister a communication event handler for a specific object. A reference to the object whose communication event handler is to be removed is passed as a parameter. If the object reference is not valid, an `InvalidObject` exception will be raised. If no communication event handler has been registered for the specified object, a `NoHandler` exception will be raised.

The `unreg_glob_client_handler()` method can be invoked by your client program to un-register a global communication event handler. If no global communication event handler has been registered, a `NoHandler` exception will be raised.

Registering client communication event handlers

There are methods for registering both global and per-object communication event handlers. In either case, the client uses the `VIS_EXT::HandlerRegistry::instance()` method to obtain a pointer to the ORB's event handler registry. Code sample C.5 shows the registration process for a global communication event handler and shows how to register a per-object communication event handler. Both examples assume that the `AccountClientHandler` class has been defined and implemented, as shown in Code sample C.2 and Code sample C.3.

Code sample C.5 Registering a global client communication event handler

```
#include <iostream.h>
#include "account_c.hh"
int main(int argc, char *const *argv)
{
      CORBA::Boolean ret;
      // Initialize the ORB
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
```

```
        ...

        // Declare the account object
        Account_var *account_object;

        // Declare the account event handler
        AccountClientHandler client_handler;

        // Obtain a handle to the ORB's registry
        VIS_EXT::HandlerRegistery_ptr registry_handle =
              VIS_EXT::HandlerRegistry::instance();
        try {
              // Register the global event handler
              registry_handle->reg_glob_client_handler(&client_handler);
        }
        catch(const VIS_EXT::HandlerExists& excep) {
              cout << "A global handler was already registered" << endl;
        }

        // Bind to the account object and invoke methods

        ...

        try {
              // Un-register the global event handler
              registry_handle->unreg_glob_client_handler();
        }
        catch(const VIS_EXT::NoHandler& excep) {
              cout << "No global handler was registered" << endl;
        }

        return(1);
}
```

Code sample C.6 Registering a per-object client communication event handler

```
#include <iostream.h>
#include "account_c.hh"
int main(int argc, char *const *argv)
{
   CORBA::Boolean ret;
   // Initialize the ORB
   CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
   ...

   // Declare the account object
   Account_var *account_object;
   // Bind to the account object
   ...

   // Declare the account event handler
   AccountClientHandler client_handler;
```

```
    // Obtain a handle to the ORB's registry
    VIS_EXT::HandlerRegistery_ptr registry_handle =
            VIS_EXT::HandlerRegistry:instance();
    try {
            // Register the account event handler
            registry_handle->reg_obj_client_handler(account_object,
                    &client_handler);
    }
    catch(const VIS_EXT::HandlerExists& excep) {
            cout << "A handler was already registered" << endl;
    }

    // Invoke methods on account object
    ...

    try {
            // Un-register the account event handler
            registry_handle->unreg_obj_client_handler(account_object);
    }
    catch(const VIS_EXT::NoHandler& excep) {
            cout << "No handler was registered" << endl;
    }
    catch(const VIS_EXT::InvalidObject& excep) {
            cout << "Invalid object reference" << endl;
    }

    return(1);
}
```

Implementation communication event handlers

Your object implementations can register one or more communication event handlers with the ORB to handle ORB events for a particular ORB object. The implementation can also globally register one or more communication event handlers to handle ORB events for all ORB objects currently implemented. Implementation-side event handling can be used for a variety of purposes, such as refusing a client connection request based on the caller's identity.

Looking at the ImplEventHandler class

Code sample C.7 shows the `ImplEventHandler` class you will use to derive an implementation communication event handler. All of the methods shown make use of the `ConnectionInfo` structure, discussed on "Providing connection information" on page C-3.

Code sample C.7 ImplEventHandler class

```
class VIS_EXT
{
    ...
    class ImplEventHandler
```

```
    {
          public:
                virtual void bind(const ConnectionInfo&,
                      CORBA::Principal_ptr, CORBA::Object_ptr);
                virtual void unbind(const ConnectionInfo&,
                      CORBA::Object_ptr);
                virtual void client_aborted(const ConnectionInfo&,
                      CORBA::Object_ptr);
                virtual void pre_method(const ConnectionInfo&,
                      CORBA::Principal_ptr, CORBA::Object_ptr,
                      const char *, CORBA::Object_ptr);
                virtual void post_method(const ConnectionInfo&,
                      CORBA::Principal_ptr, CORBA::Object_ptr,
                      const char *, CORBA::Object_ptr);
    };
    ...
};
```

In the preceding figure, `CORBA::Principal_ptr` enables your client to send information to the server that identifies the client. The server can retrieve this data and determine if the requested action, such as a bind request, is to be performed.

Understanding ImplEventHandler methods

The `bind()` method is invoked every time a client wishes to connect to this object. This method allows your object implementation to do any special processing before the bind request is processed. Once this method returns, the BOA will proceed with the normal binding process. The parameters passed to this method are the connection information, the `Principal` value associated with the client and a pointer to the ORB object requested. This method may choose to reject the bind, based on the requestor's identity, by raising a `CORBA::NO_PERMISSION` exception.

The `unbind()` method will be invoked every time a client program calls the `CORBA::release()` method for a previously bound object. The BOA will pass control to this method before the un-bind occurs. The connection information and the object reference are both passed to this method.

The `client_aborted()` method will be invoked if the connection to a client program is lost. The connection information and the object reference are passed to this method.

The `pre_method()` method will be invoked every time a client program invokes a method on the object for which the handler is registered. After this method returns, the BOA will proceed with the method invocation. The connection information, `Principal` of the client, method name, and a pointer to the object are all passed to this method.

The `post_method()` method will be invoked after every invocation of a method by a client on the object being traced. After this method returns, the results of the method invocation will be returned to the client. The connection information, `Principal` of the client, method name and a pointer to the object are all passed to this method.

Note If the method invoked by the client raises an exception, the `post_method()` method will not be invoked.

Creating implementation communication event handlers

Code sample C.8 shows how you can create an implementation communication event handler for the `Accountimpl` object by deriving your own class from the `ImplEventHandler` class. Code sample C.9 shows the implementation for the `AccountImplHandler` methods defined. You only need to define and provide method implementations for those ORB events you wish to handle.

Code sample C.8 Example communication event handler class for the Accountimpl object implementation

```
class AccountImplHandler : public VIS_EXT::ImplEventHandler {
      public:
            void bind(const ConnectionInfo&, CORBA::Principal_ptr,
                        CORBA::Object_ptr);
            void unbind(const ConnectionInfo&, CORBA::Principal_ptr,
                        CORBA::Object_ptr);
};
```

Code sample C.9 Method implementations for the AccountImplHandler class methods

```
void AccountImplHandler::bind(const ConnectionInfo&,
            CORBA::Principal_ptr, CORBA::Object_ptr obj)
{
      cout << "Bind request arrived for " << obj->_interface_name << endl;
      ...
};

void AccountImplHandler::unbind(const ConnectionInfo&,
            CORBA::Principal_ptr, CORBA::Object_ptr obj)
{
      cout << "Un-Bind request arrived for " << obj->_interface_name << endl;
      ...
};
```

Registering implementation communication event handlers with the handler registry

As with client programs, the `HandlerRegistry` is used to register implementation event handlers. The `HandlerRegistry` class is show in Code sample C.4 on page C-4.

Invoking HandlerRegistry methods from object implementations

The `reg_obj_impl_handler()` method can be invoked by your object implementation to register a communication event handler with the BOA for a specific object. The parameters passed to this method are a reference to the object and a pointer to the object's `ImplEventHandler`. If the object reference is not valid, an `InvalidObject` exception will be raised. If a communication event handler has already been registered for the specified object, a `HandlerExists` exception will be raised. You can use the `unreg_obj_impl_handler()` method to un-register a previously registered communication event handler.

The reg_glob_impl_handler() method can be invoked by your object implementation to register a communication event handler with the BOA for all objects contained in the implementation. The parameter passed to this method is a pointer to the object's ImplEventHandler. If a global communication event handler has already been registered for this implementation, a HandlerExists exception will be raised. You can use the unreg_glob_impl_handler() method to un-register a previously registered global communication event handler.

Note If both an object communication event handler and a global communication event handler are registered, the object communication event handler will take precedence for ORB events that occur which are related to its object. All other ORB events will be handled by the global communication event handler.

The unreg_obj_impl_handler() method can be invoked by your object implementation to un-register a communication event handler for a specific object. A reference to the object whose communication event handler is to be removed is passed as a parameter. If the object reference is not valid, an InvalidObject exception will be raised. If a communication event handler has not been registered for the specified object, a NoHandler exception will be raised.

The unreg_glob_impl_handler() method can be invoked by your object implementation to un-register a global communication event handler. If no global communication event handler has been registered, a NoHandler exception will be raised.

Registering implementation communication event handlers

Like client programs, your object implementations will use similar procedures for registering both global and per-object communication event handlers. In both cases, the client uses the VIS_EXT::HandlerRegistry::instance() method to obtain a pointer to the BOA's event handler registry. Code sample C.10 shows the registration process for a global communication event handler and Code sample C.11 shows how to register a per-object communication event handler. Both examples assume that the AccountImplHandler class has been defined and implemented, as shown in Code sample C.9 on page C-9.

Code sample C.10 Registering a global communication event handler for an object implementation

```
#include <account_s.hh>
int main(int argc, char **argv)
{
   // Initialize ORB and Basic Object Adaptor (BOA)
   CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
   CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

   // Instantiate the AccountImpl object
   AccountImpl AccountImpl_obj = new AccountImpl("Jack B. Quick");

   // Define the impl_handler and registry instance
   AccountImplHandler impl_handler;
   VIS_EXT::HandlerRegistry_ptr registry_handle;
   registry_handle = VIS_EXT::HandlerRegistry::instance();
```

```
    try {
            // Register a global event handler
            registry_handle->reg_glob_impl_handler(&impl_handler);
    }
    catch(const VIS_EXT::HandlerExists& excep) {
            cout << "Global handler already defined" << endl;
    }

    // Instantiate Account Class, activate object and implementation
    ...
    try {
            registry_handle->unreg_glob_impl_handler();
    }
    catch(const VIS_EXT::NoHandler& excep) {
            cout << "Removal of global event handler failed" << endl;
    }
    ...
}
```

Code sample C.11 Registering a per-object communication event handler for an object implementation

```
#include <account_s.hh>
int main(int argc, char **argv)
{
    // Initialize ORB and Basic Object Adaptor (BOA)
    CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
    CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

    // Instantiate the AccountImpl object
    AccountImpl AccountImpl_obj = new AccountImpl("Jack B. Quick");

    // Define the impl_handler and registry instance
    AccountImplHandler impl_handler;
    VIS_EXT::HandlerRegistry_ptr registry_handle;
    registry_handle = VIS_EXT::HandlerRegistry::instance();
    try {
          // Register a global event handler
          registry_handle->reg_obj_impl_handler(AccountImpl_obj,
                  &impl_handler);
    }
    catch(const VIS_EXT::HandlerExists& excep) {
            cout << "Handler already defined for " <<
                    AccountImpl_obj->_instance_name << endl;
    }

    // Instantiate Account Class, activate object and implementation
    ...
    try {
          registry_handle->unreg_obj_impl_handler(AccountImpl_obj);
    }
    catch(const VIS_EXT::NoHandler& excep) {
            cout << "Removal of event handler failed for " <<
                    AccountImpl_obj->_instance_name << endl;
    }
    ...
}
```

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X