Introduction to Orbix C++ Edition

IONA Technologies PLC
September 2000

Orbix is a Registered Trademark of IONA Technologies PLC.

While the information in this publication is believed to be accurate, IONA Technologies PLC makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IONA Technologies PLC shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Java is a trademark of Sun Microsystems, Inc.

COPYRIGHT NOTICE

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of IONA Technologies PLC. No third party intellectual property right liability is assumed with respect to the use of the information contained herein. IONA Technologies PLC assumes no responsibility for errors or omissions contained in this book. This publication and features described herein are subject to change without notice.

Copyright © 2000 IONA Technologies PLC. All rights reserved.

All products or services mentioned in this manual are covered by the trademarks, service marks, or product names as designated by the companies who market those products.

M 2 4 8 9

Contents

Preface

Orbix is a suite of integrated IONA products offering an advanced execution environment for CORBA applications. This standards-based deployment infrastructure delivers secure, transactional, distributed applications in a managed environment.

This guide presents an overview of the components of Orbix and how to build multi-tier systems.

Orbix documentation is periodically updated. New versions between releases are available at this site:

`http://www.iona.com/docs/orbix/orbix33.html`

If you need assistance with Orbix or any other IONA products, contact IONA at `support@iona.com`. Comments on IONA documentation can be sent to `doc-feedback@iona.com`.

Objectives of Orbix

The objectives of Orbix are as follows:

- To provide a rich set of integrated components that can be used as a basis for delivering advanced applications. These components are accessible from standard APIs, and are supported with graphical user interfaces (GUI) and system management facilities.
- To enable integration with other system resources, such as databases.
- To enable access to Orbix server applications from within (intranet) and outside (Internet) corporate networks.

Audience

The *Introduction to Orbix C++ Edition* is intended for use by application designers who wish to familiarize themselves with Orbix C++, and develop distributed applications using Orbix C++ components. This guide assumes as a prerequisite that the reader is familiar with programming in C++ and/or Java.

This guide does not discuss every API in great detail, but gives a general overview of the capabilities of the Orbix component set, and an overview of their APIs. For complete information on each Orbix component, consult the individual programming guides.

Organization of this Guide

The Orbix guide is divided into the following parts:

Part I Introduction to CORBA and Orbix

This part gives an overview of the architecture of OrbixOTM, including a brief overview of each of the components and services of the OrbixOTM suite.

Part II The StockWatch Demonstration

This part guides the reader through a worked example called StockWatch. Initially, the example is based on code generated by the Orbix Code Generation Toolkit. In later chapters, the examples are based directly on `example1...example5` from the *iona*`/demos/` directory.

The StockWatch example is progressively extended as the various features of Orbix are incorporated.

Related Documentation

The *Introduction to Orbix C++ Edition* is part of a set of manuals that are delivered with your Orbix installation. The following table provides a comprehensive list of all related documents.

Related Documentation	
Component	**Reference**
Orbix C++	*Orbix C++ Programmer's Guide* *Orbix C++ Programmer's Reference* *Orbix C++ Administrator's Guide*
Orbix Java	*Orbix Programmer's Guide Java Edition* *Orbix Programmer's Reference Java Edition* *Orbix Administrator's Guide Java Edition*
Orbix Code Generation Toolkit	*Orbix Code Generation Toolkit Programmer's Guide*
OrbixCOMet	*OrbixCOMet Programmer's Guide and Reference*
OrbixNames	*OrbixNames Programmer's and Administrator's Guide*
Orbix Wonderwall	*Orbix Wonderwall Administrator's Guide*
OrbixEvents	*OrbixEvents Programmer's Guide*
OrbixSSL	*OrbixSSL C++ Programmer's and Administrator's Guide* *OrbixSSL Java Programmer's and Administrator's Guide*
OrbixOTS	*OrbixOTS Programmer's and Administrator's Guide*

Table 0.1: *Reference Material*

Document Conventions

This guide uses the following typographical conventions:

`Constant width`	Constant width in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `#include <stdio.h>`
Italic	Italic words in normal text represent *emphasis* and *new terms*. Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example: `cd /users/`*your_name*

This guide may use the following keying conventions:

< >	Some command examples may use angle brackets to represent variable values you must supply (this is an older convention).
... . . .	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

Part I

Introduction to CORBA and Orbix

I

Introduction to CORBA and Orbix

Orbix is a software environment that allows you to build and integrate distributed applications. Orbix is a full implementation of the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) specification. This chapter introduces CORBA and describes how Orbix implements this specification.

CORBA and Distributed Object Programming

The diversity of modern networks makes the task of network programming very difficult. Distributed applications often consist of several communicating programs written in different programming languages and running on different operating systems. Network programmers must consider all of these factors when developing applications.

The Common Object Request Broker Architecture (CORBA) defines a framework for developing object-oriented, distributed applications. This architecture makes network programming much easier by allowing you to create distributed applications that interact as though they were implemented in a single programming language on one computer.

CORBA also brings the advantages of object-oriented techniques to a distributed environment. It allows you to design a distributed application as a set of cooperating objects and to re-use existing objects in new applications.

The Role of an Object Request Broker

CORBA defines a standard architecture for Object Request Brokers (ORBs). An ORB is a software component that mediates the transfer of messages from a program to an object located on a remote network host. The role of the ORB is to hide the underlying complexity of network communications from the programmer.

An ORB allows you to create standard software objects whose member functions can be invoked by *client* programs located anywhere in your network. A program that contains instances of CORBA objects is often known as a *server*.

When a client invokes a member function on a CORBA object, the ORB intercepts the function call. As shown in Figure 1.1, the ORB redirects the function call across the network to the target object. The ORB then collects results from the function call and returns these to the client.

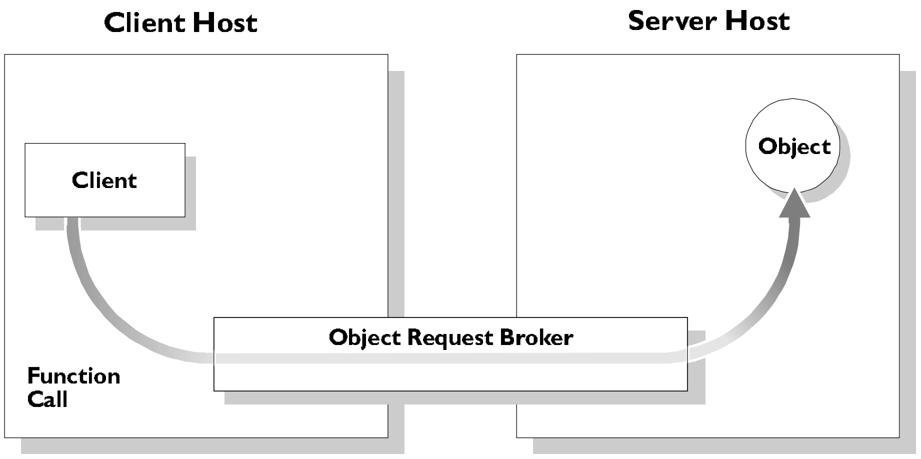


Figure 1.1: *The Object Request Broker*

The Nature of Objects in CORBA

CORBA objects are just standard software objects implemented in any supported programming language. CORBA supports several languages, including C++, Java, and Smalltalk.

With a few calls to an ORB's application programming interface (API), you can make CORBA objects available to client programs in your network. Clients can be written in any supported programming language and can call the member functions of a CORBA object using the normal programming language syntax.

Although CORBA objects are implemented using standard programming languages, each CORBA object has a clearly-defined interface, specified in the CORBA Interface Definition Language (IDL). The interface definition specifies which member functions are available to a client, without making any assumptions about the implementation of the object.

To call member functions on a CORBA object, a client needs only the object's IDL definition. The client does not need to know details such as the programming language used to implement the object, the location of the object in the network, or the operating system on which the object runs.

The separation between an object's interface and its implementation has several advantages. For example, it allows you to change the programming language in which an object is implemented without changing clients that access the object. It also allows you to make existing objects available across a network.

The Structure of a CORBA Application

The first step in developing a CORBA application is use CORBA IDL to define the interfaces to objects in your system. You then compile these interfaces using an IDL compiler.

An IDL compiler generates C++ from IDL definitions. This C++ includes *client stub code*, which allows you to develop client programs, and *object skeleton code*, which allows you to implement CORBA objects.

As shown in Figure 1.2 on page 6, when a client calls a member function on a CORBA object, the call is transferred through the client stub code to the ORB. If the client has not accessed the object before, the ORB refers to a database, known as the *Implementation Repository*, to determine exactly which object should receive the function call. The ORB then passes the function call through the object skeleton code to the target object.

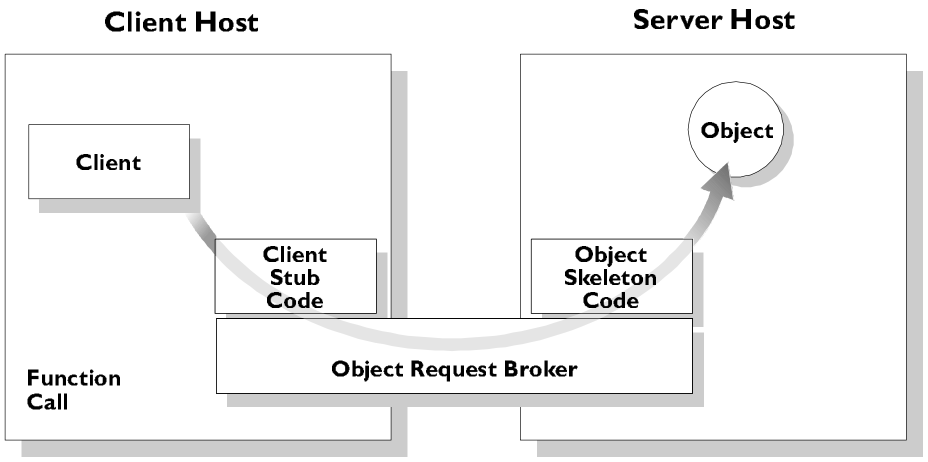


Figure 1.2: *Invoking on a CORBA Object*

The Structure of a Dynamic CORBA Application

One difficulty with normal CORBA programming is that you have to compile the IDL associated with your objects and use the generated C++ code in your applications. This means that your client programs can only call member functions on objects whose interfaces are known at compile-time. If a client wishes to obtain information about an object's IDL interface at runtime, it needs an alternative, *dynamic* approach to CORBA programming.

The CORBA *Interface Repository* is a database that stores information about the IDL interfaces implemented by objects in your network. A client program can query this database at runtime to get information about those interfaces. The client can then call member functions on objects using a component of the ORB called the *Dynamic Invocation Interface* (DII), as shown in Figure 1.3 on page 7.

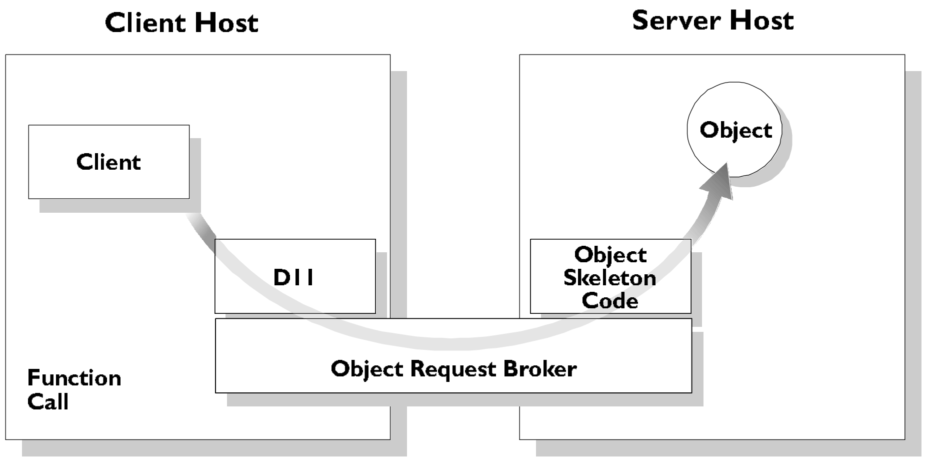


Figure 1.3: *Client Invoking a Function Using the DII*

CORBA also supports dynamic server programming. A CORBA program can receive function calls through IDL interfaces for which no CORBA object exists. Using an ORB component called the *Dynamic Skeleton Interface* (DSI), the server can then examine the structure of these function calls and implement them at runtime. Figure 1.4 on page 8 shows a dynamic client program communicating with a dynamic server implementation.

Interoperability between Object Request Brokers

The components of an ORB make the distribution of programs transparent to network programmers. To achieve this, the ORB components must communicate with each other across the network.

In many networks, several ORB implementations coexist and programs developed with one ORB implementation must communicate with those developed with another. To ensure that this happens, CORBA specifies that ORB components must communicate using a standard network protocol, called the *Internet Inter-ORB Protocol* (IIOP).

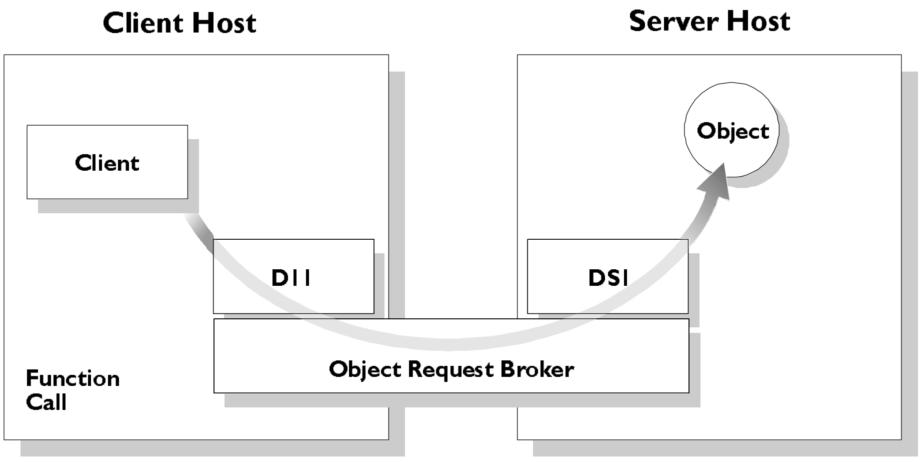


Figure 1.4: *Function Call Using the DII and DSI*

The Object Management Architecture

An ORB is one component of the OMG's Object Management Architecture (OMA). This architecture defines a framework for communications between distributed objects.

As shown in Figure 1.5 on page 9, the OMA includes four elements:

- Application objects.
- The ORB.
- The *CORBAservices*.
- The *CORBAfacilities*.

Application objects are objects that implement programmer-defined IDL interfaces. These objects communicate with each other, and with the CORBAservices and CORBAfacilities, through the ORB. The CORBAservices and CORBAfacilities are sets of objects that implement IDL interfaces defined by CORBA and provide useful services for some distributed applications.

When writing Orbix applications, you may require one or more CORBAservices or CORBAfacilities. This section provides a brief overview of these components of the OMA.

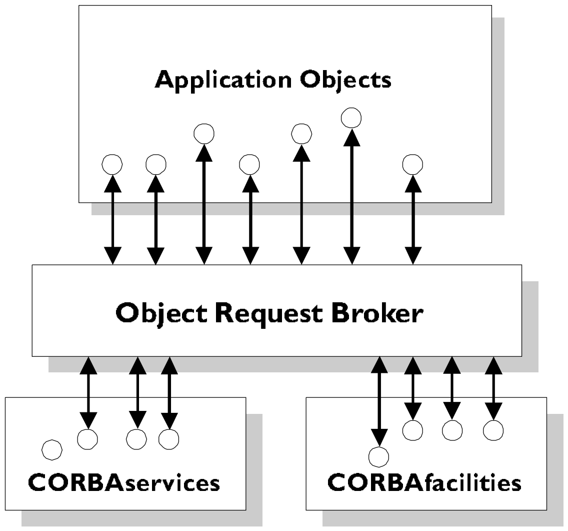


Figure 1.5: *The Object Management Architecture*

The CORBAservices

The CORBAservices define a set of low-level services that allow application objects to communicate in a standard way. These services include the following:

- The *Naming Service*. Before using a CORBA object, a client program must get an identifier for the object, known as an *object reference*. This service allows a client to locate object references based on abstract, programmer-defined object names.
- The *Trader Service*. This service allows a client to locate object references based on the desired properties of an object.
- The *Object Transaction Service*. This service allows CORBA programs to interact using transactional processing models.

- The *Security Service*. This service allows CORBA programs to interact using secure communications.
- The *Event Service*. This service allows objects to communicate using decoupled, event-based semantics, instead of the basic CORBA function-call semantics.

IONA Technologies implements several CORBAservices including all the services listed above.

The CORBAfacilities

The CORBAfacilities define a set of high-level services that applications frequently require when manipulating distributed objects. The CORBAfacilities are divided into two categories:

- The *horizontal* CORBAfacilities.
- The *vertical* CORBAfacilities.

The horizontal CORBAfacilities consist of user interface, information management, systems management, and task management facilities. The vertical CORBAfacilities standardize IDL specifications for market sectors such as healthcare and telecommunications.

How Orbix Implements CORBA

Orbix is an ORB that fully implements the CORBA 2 specification. By default, all Orbix components and applications communicate using the CORBA standard IIOP protocol.

The components of Orbix are as follows:

- The *IDL compiler* parses IDL definitions and produces C++ code that allows you to develop client and server programs.
- The *Orbix library* is linked against every Orbix program and implements several components of the ORB, including the DII, the DSI, and the core ORB functionality.

- The *Orbix daemon* is a process that runs on each server host and implements several ORB components, including the Implementation Repository.
- The *Orbix Interface Repository server* is a process that implements the Interface Repository.

Orbix also includes several programming features that extend the capabilities of the ORB. These features are described in Part IV, “Advanced Orbix Programming”.

In addition, Orbix is an enterprise ORB that combines the functionality of the core CORBA standard with an integrated suite of services: OrbixNames, OrbixEvents, OrbixOTS, and OrbixSSL. This chapter introduces the architecture of Orbix and briefly describes each of these services.

Note: Only an overview of these components is given here. For more detailed descriptions of functionality, refer to the individual programming guides and reference guides that accompany each component.

Orbix Components

Table 1.1 gives a brief synopsis of the Orbix suite.

Orbix	The multithreaded Orbix Object Request Broker (ORB) is at the heart of Orbix. This is IONA Technologies' implementation of the OMG (Object Management Group) CORBA specification.
Orbix Code Generation Toolkit	The Orbix Code Generation Toolkit is a powerful development tool that can automatically generate code from IDL files. The Orbix Code Generation Toolkit contains an IDL parser called *IDLgen* and ready-made applications called *genies* that allow you to generate Java or C++ code from CORBA IDL files automatically.
Orbix Wonderwall	Orbix Wonderwall is a firewall product that enables CORBA applications to be deployed on an intranet or the Internet with protection against hostile access to your internal network.
OrbixCOMet	OrbixCOMet is a bridge that links the world of COM with CORBA. It enables COM applications to invoke on CORBA objects as if they were COM objects, and vice versa.
OrbixOTS	OrbixOTS brings transactional object capability to enterprise-wide applications. This is IONA Technologies' implementation of the OMG CORBAservices Object Transaction Service (OTS).
OrbixSSL	OrbixSSL integrates Orbix with Secure Socket Layer (SSL) security. Using OrbixSSL, distributed applications can securely transfer confidential data across a network. OrbixSSL offers CORBA level zero security.
OrbixNames	OrbixNames maintains a repository of mappings that associate objects with recognisable names. This is IONA Technologies' implementation of the OMG CORBAservices Naming Service.
OrbixEvents	OrbixEvents enables asynchronous communication between groups of objects via event channels. This is IONA Technologies' IIOP-based OMG CORBAservices Event Service.

Table 1.1: *The Orbix Suite*

Orbix Architecture

The overall architecture of Orbix and it's components is shown in Figure 1.6. On the lower part of Figure 1.6, a number of CORBA servers and clients are shown attached to an intranet and, on the top left, a sample client is shown attached to the system via the Internet. It is necessary to pencil in a number of server hosts in this basic illustration because Orbix is an intrinsically distributed system. In contrast to the star-shaped architecture of many traditional systems, with clients attached to a central monolithic server, the architecture of Orbix is based on a collection of components cooperating across a number of hosts.

Some standard services, such as the CORBA Naming Service (OrbixNames) and the CORBA Event Service (OrbixEvents), are implemented as clearly identifiable processes with an associated executable. There can be many instances of these processes running on one or more machines.

Other services, for example the Orbix Object Transaction Service (OrbixOTS), rely on cooperation between components. They are, either wholly or partly, based on libraries linked with each component. You cannot, for example, point to a single process and say that it embodies OrbixOTS. Services such as this are intrinsically distributed.

Since Orbix has an open, standards-based architecture it can readily be extended to integrate with other CORBA-based products. In particular, as Figure 1.6 shows, integration with a mainframe is possible when Orbix is combined with an ORB running on OS/390.

For more information on Orbix, see the *Orbix Programmer's Guide C++ Edition*, *Orbix Programmer's Reference C++ Edition* and *Orbix Administrator's Guide C++ Edition*.

In the remainder of this section on Orbix architecture, each of the components of Orbix will be presented with a brief description of the main features.

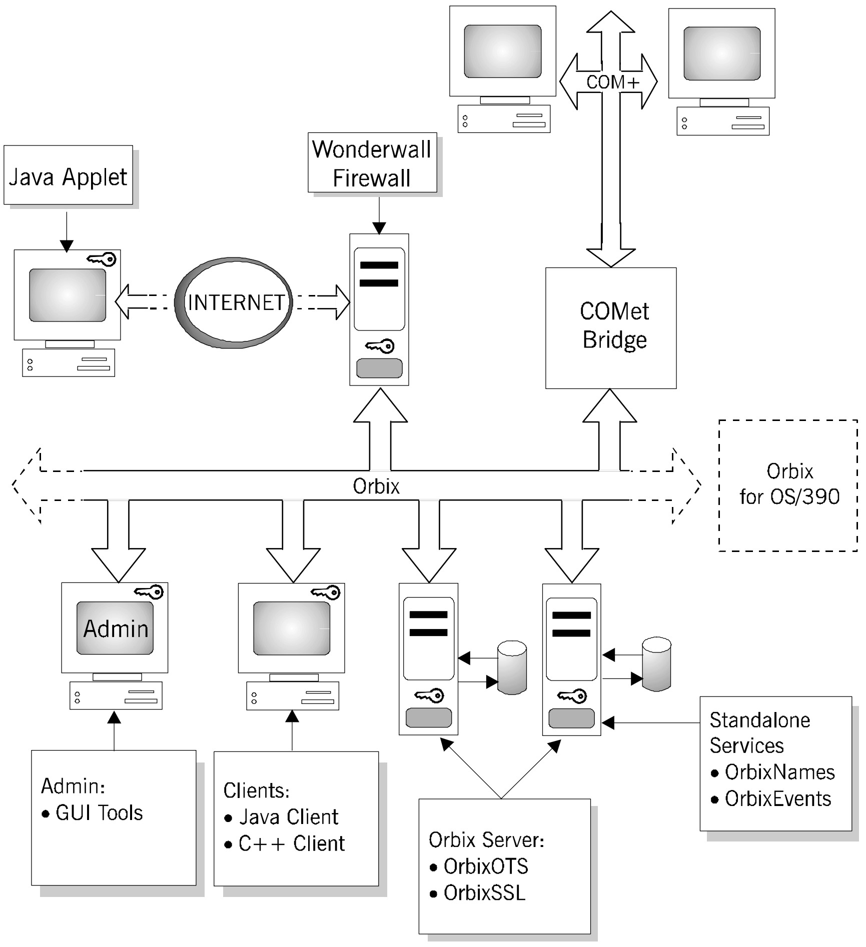


Figure 1.6: *The Orbix Architecture*

OrbixNames—The Naming Service

OrbixNames is IONA's implementation of the CORBA Naming Service. The role of OrbixNames is to allow a name to be associated with an object and to allow that object to be found using that name. A server that holds an object can register it with OrbixNames, giving it a name that can be used by other components of the system to subsequently find the object. OrbixNames maintains a repository of mappings (*bindings*) between names and object references. OrbixNames provides operations to do the following:

- Resolve a name.
- Create new bindings.
- Delete existing bindings.
- List the bound names.

Using a Naming Service such as OrbixNames to locate objects allows developers to hide a server application's location details from the client. This facilitates the invisible relocation of a service to another host. The entire process is hidden from the client.

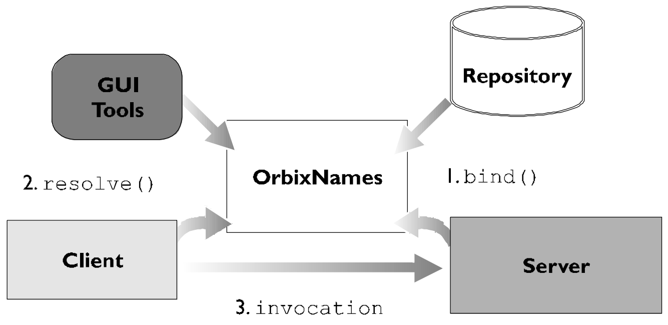


Figure 1.7: *The OrbixNames Architecture*

Figure 1.7 summarizes the functionality of OrbixNames, which is as follows:

1. A server registers object references in OrbixNames. OrbixNames then maps these object references to names.
2. Clients resolve names in OrbixNames.
3. Clients remotely invoke on object references in the server.

OrbixNames, which runs as an Orbix server, has a number of interfaces defined in IDL that allow the components of the system to use its facilities. Other features of OrbixNames include an enhanced GUI browser interface. OrbixNames can support clients that use either IIOP or the Orbix protocol.

For more information on OrbixNames, see the *OrbixNames Programmer's and Administrator's Guide*.

OrbixEvents—The Event Service

OrbixEvents is IONA Technologies' implementation of the CORBA Event Service enabling decoupled communication between objects via a set of one or more event channels. OrbixEvents supports typed and untyped event models. It is implemented as a stand-alone Orbix server.

Event channels mediate the transfer of events between suppliers (where an event originates) and consumers (where an event is transferred) as follows:

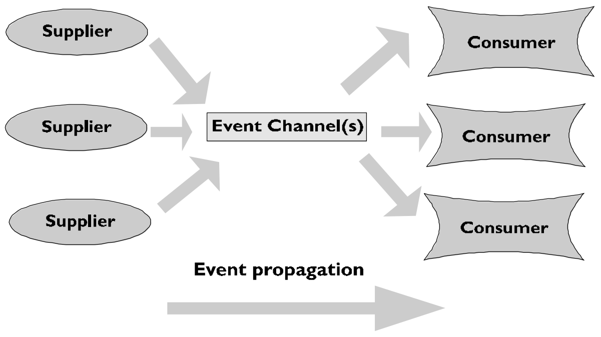


Figure 1.8: *Schematic View of OrbixEvents Architecture*

1. Event channels allow consumers to register interest in events, and they store this registration information.
2. Event channels accept incoming events from suppliers.
3. The channels forward supplier-generated events to registered consumers.

Suppliers and consumers connect to the event channel and not directly to each other. To the supplier, the event channel appears as a single consumer. To the consumer, the event channel appears as a single supplier. In this way, the event channel decouples suppliers and consumers.

Suppliers can issue events to any number of consumers using a single event channel, and any supplier or consumer can connect to more than one channel. There is no correlation between the number of suppliers and the number of consumers. New consumers and suppliers can be added easily to the system.

Suppliers, consumers, and event channels are implemented as CORBA applications. Events are defined using standard IDL. Suppliers, consumers, and event channels each implement clearly-defined IDL interfaces that support the steps required to transfer events in a distributed system.

For more information on OrbixEvents, see the *OrbixEvents Programmer's Guide*.

OrbixOTS—The Transaction Service

OrbixOTS brings distributed transactional capability to enterprise-wide applications. OrbixOTS is an implementation of the CORBAservices Object Transaction Service (OTS) specification. In practice OrbixOTS is implemented as a linked-in library to which Orbix clients and servers must link. This reduces the risk of OrbixOTS being a single point of failure.

The OMG OTS standard requires that OTS be able to import and export transactions to and from XA compliant resource managers. In accordance with this requirement, OrbixOTS is fully compatible with X/Open compliant software. The design of OrbixOTS is based on the X/Open reference model and includes the following improvements:

- The procedural XA and TX interfaces have been replaced with a set of CORBA interfaces defined in IDL.

- All inter-component communication is mandated to be via CORBA function calls on remote method invocations.

OrbixOTS allows resources that export XA interfaces to participate in a distributed transaction. Correspondingly, any common off-the-shelf database supporting the XA interface can participate in an OrbixOTS transaction.

Most commercial relational databases and some message queuing systems export XA interfaces which can be integrated into OrbixOTS applications.

OrbixOTS acts as a transaction and recovery co-ordinator for distributed transactions. The process can be summarized as follows:

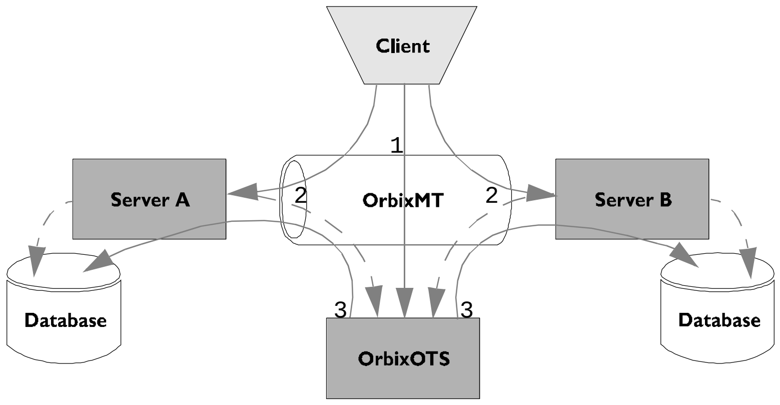


Figure 1.9: *OrbixOTS Functionality Overview*

1. Clients begin or commit a transaction.
2. Clients invoke requests as part of a transaction. Servers register resources with the transaction.
3. The OrbixOTS transaction manager then transparently co-ordinates the transaction and failure recovery, including resource management, voting for commit or rollback, and heuristic outcomes.

Distributed Transactions

Consider the case where two bank accounts reside in different applications, control threads, processes or machines: to guarantee Atomicity, Consistency, Isolation, and Durability (ACID) on the complete system, distributed transaction processing (DTP) must be employed.

An external entity usually called a transaction co-ordinator is required to allow a transaction to span more than one application, process, or machine. It does this by keeping track of resources involved in the transaction, and by co-ordinating transaction completion.

The transaction co-ordinator uses a two-phase commit (2PC) protocol to commit distributed transactions. Firstly, all resources are asked to *prepare* the transaction and to return a *vote* to indicate whether they are willing to make modifications durable. If all resources voted to commit, then they are asked to commit in turn: if one or more resources voted to rollback, then all resources are asked to rollback in turn. In this way, atomicity is largely assured. For further details on 2PC, see "Two-Phase Commit Protocol" on page 168 and the *OrbixOTS Programmer's and Administrator's Guide*.

X/Open DTP Standard

The X/Open company has defined a standard for DTP systems called the DTP reference model. This identifies three components in a DTP scenario—the application, the resource manager, and the transaction manager—and defines procedural interfaces between them; XA between transaction managers and resource managers, and TX between the application and transaction manager.

For more details on the X/Open DTP standard, see the *OrbixOTS Programmer's and Administrator's Guide*.

Security with OrbixSSL

OrbixSSL introduces Level 0 CORBA security, as specified by the OMG, to the Orbix product suite. Level 0 corresponds to the provision of authentication and session encryption, which maps onto the functionality provided by the Secure Socket Layer (SSL) library.

SSL is a protocol for providing data security for applications that communicate across networks via TCP/IP. By default, Orbix applications communicate using the standard CORBA Internet Inter-ORB Protocol (IIOP). These application-level protocols are layered above the transport-level protocol TCP/IP.

OrbixSSL provides authentication, privacy, and integrity for communications across TCP/IP connections as follows:

Authentication	Allows an application to verify the identity of another application with which it communicates.
Privacy	Ensures that data transmitted between applications can not be understood by a third party.
Integrity	Allows applications to detect whether data was modified during transmission.

To initiate a TCP/IP connection, OrbixSSL provides a security ‘handshake’. This handshake results in the client and server agreeing on an ‘on the wire’ encryption algorithm, and also fulfils any authentication requirements for the connection. Thereafter, OrbixSSL’s only role is to encrypt and decrypt the byte stream between client and server.

The steps involved in establishing an OrbixSSL connection are as follows:

1. The client initiates a connection by contacting the server.
2. The server sends an X.509 certificate to the client. This certificate includes the server’s public encryption key.
3. The client authenticates the server’s certificate (for example, an X.509 certificate, endorsed by an accredited certifying authority).
4. The client sends the certificate to the server for authentication.
5. The server generates a session encryption key and sends it to the client encrypted using the client’s public key: the session is now established.

Once the connection has been established, certain data is cached so that in the event of a dropping and resumption of the dialogue, the handshake is curtailed and connection re-establishment is accelerated.

For more information on OrbixSSL, see *OrbixSSL Programmer’s and Administrator’s Guide*.

Part II

The StockWatch Demonstration

2

The StockWatch Demonstration

Orbix is introduced here by presenting the steps involved in writing a simple application and concentrating on the features of the core ORB. The Orbix Code Generation Toolkit is used to generate a starting point for the demonstration application.

Overview of the StockWatch Demonstration

To illustrate Orbix and its components, a simple StockWatch example is developed. This example provides stock price information for a selected number of stocks. It consists of a server process, which accesses stock prices from persistent storage, and a number of client processes.

The StockWatch demonstration is a sample implementation of an application that gathers and distributes information about a range of stocks. A history of prices is archived for each of the stocks and this archive can be queried by subscribers to the StockWatch service. Likewise, whenever stock is traded the StockWatch service allows the new price to be recorded leading to an update in the price history archive.

The StockWatch application is not concerned with modeling all aspects of a stock exchange—for example StockWatch does not provide any facilities for trading stock, such as a broker service. The demonstration focuses on providing a service to record stock prices, archive the data and make the historical price data available to subscribers.

In the *IONARoot*/`demos/common/src` directory of your Orbix installation you will find the code for a demonstration called StockWatch. This demonstration forms the basis for the example code throughout this book:

1. Example 1 "The Naming Service" on page 95 introduces object location transparency using OrbixNames.
2. Example 2 "The Event Service" on page 117 introduces decoupled event-based communication using OrbixEvents.
3. Example 3 "The Object Transaction Service" on page 143 introduces distributed transaction processing using OrbixOTS.
4. Example 4 "Security" on page 177 introduces security in the form of OrbixSSL.
5. Example 5 "Load Balancing" on page 187 introduces load balancing for Orbix applications.

Starting Point for StockWatch

The example code for the current chapter is not supplied with the StockWatch demonstration. Instead, the starting code for this chapter is automatically generated by the Orbix Code Generation Toolkit. A small amount of additional code has to be supplied by the developer to complete the sample application.

In later chapters, as the examples become more complex, the sample code for StockWatch is based directly on the code found in the *IONARoot*/`demos/common/src` directory of Orbix.

Configuration Hints

Some basic hints on Orbix configuration are given here to help you get started and check that your system is correctly configured. For complete details on configuring Orbix consult the *Orbix C++ Administrator's Guide* and the installation guide for your particular platform.

The following aspects of configuring Orbix are discussed below:

- Environment variables.
- Orbix configuration files.
- OCGT configuration.

For convenience, this section assumes that Orbix has been installed in its default location:

`/opt/iona`	UNIX default Orbix root directory.
`C:\Iona`	Windows default Orbix root directory.

This Orbix root directory is referred to as *IONARoot* in the following subsections.

Environment Variables

The following environment variables are a basic part of Orbix configuration on most platforms:

`PATH`	Add the following directories: *IONARoot*/`bin` *IONARoot*/`contrib`
Library Path	On UNIX, this variable specifies the directory where Orbix shared libraries are located. The name of this variable depends on the particular platform. For example, on Solaris it is `LD_LIBRARY_PATH` and on HP-UX it is `SHLIB_PATH`. This variable should include the following in its list of directories: *IONARoot*/`lib`
`IT_CONFIG_PATH`	This must point at the directory containing the Orbix configuration file `iona.cfg`. To use the default Orbix configuration files, set this equal to the following directory: *IONARoot*/`config`
`IT_IDLGEN_CONFIG_FILE`	Specifies the location of the configuration file `idlgen.cfg` used to configure the Orbix Code Generation Toolkit. To use the default version of `idlgen.cfg`, set it equal to the following: *IONARoot*/`config/idlgen.cfg`

As an alternative to using the environment variable `IT_CONFIG_PATH` you can use the variable `IT_IONA_CONFIG_FILE` to specify the full pathname of the Orbix configuration file. This allows you to set the name of the Orbix configuration file to something other than `iona.cfg`.

Orbix Configuration Files

After installing Orbix, a complete set of configuration files can be found underneath the directory *IONARoot*/`config`. The main configuration file is `iona.cfg`, which in turn includes other configuration files such as `common.cfg`, `orbixnames3.cfg`, `orbix3.cfg` and others.

These files can be edited either directly using a text editor, or using the Configuration Explorer Tool supplied with Orbix.

OCGT Configuration

The configuration file for the Orbix Code Generation Toolkit (OCGT) is called `idlgen.cfg`. Before running the idlgen tool, you must set the environment variable `IDLGEN_CONFIG_FILE` to the absolute pathname of this configuration file.

To check the basic configuration of OCGT, run the following command:

```
idlgen -list
```

which should generate output similar to the following:

```
available genies are...

cpp_equal.tcl   cpp_op.tcl      cpp_random.tcl  stats.tcl
cpp_genie.tcl   cpp_print.tcl   idl2html.tcl
```

Defining the Stock Interface

The starting point for the implementation of StockWatch is the definition of an interface between the client and the server. This interface is defined using the OMG Interface Definition Language (IDL), which allows you to specify interfaces in a language-neutral manner. For example, it does not matter whether the server or client is implemented in Java or C++; in each case the same IDL specification can be used

The IDL for StockWatch is defined as follows:

```
     // IDL
1    // File: StockWatch.idl

2    typedef float Money;
     typedef string Symbol;

     // Interfaces

3    interface Stock {
4          Symbol getSymbol();
           string getDescription();
           Money getCurrentPrice();
     };
```

The most important part of the IDL file is the definition of the `Stock` interface. It represents a class of CORBA objects, which make up the basic building blocks of a CORBA application.

The preceding IDL file can be explained as follows:

1. Two consecutive forward slashes // introduce a comment. All characters from the // up to the end-of-line are ignored by the IDL compiler.
2. The `typedef` directive has a similar syntax to the `typedef` of C and C++. In the above example, `Money` becomes a synonym for the type `float`, and `Symbol` becomes a synonym for the type `string`.
3. Interface definitions are introduced by the `interface` reserved keyword. There is a strong similarity between the syntax of an IDL interface and a C++ or Java class. The construction is called `interface` instead of `class` to underline the fact that IDL does not provide the implementation. Implementation details are left to the target language (such as Java or C++) instead.
4. Within the scope of the `Stock` interface, between the curly braces, three *operations* are defined: `getSymbol()`, `getDescription()` and `getCurrentPrice()`. The syntax of these operation declarations is similar to the syntax of method declarations in Java or C++. These particular operations take no parameters, but they do declare return values. If an operation has no return value it must specify the return type as `void`.

Note: All of the IDL operations are *public*. There is no equivalent in IDL to the C++ and Java concept of private scope.

The `Stock` IDL interface provides you with the starting point for both the client and server ends of the CORBA application.

A client programmer, supplied with a copy of the above IDL file `StockWatch.idl`, has all the information needed to write the client component. It does not matter whether the server is written in C++, Java, COBOL or PL/I, nor does it matter which platform the server runs on. The client programmer is insulated from these implementation details and can focus on writing a client that is compatible with the given IDL.

Likewise, the server programmer, supplied with a copy of the IDL file `StockWatch.idl`, can go ahead and implement the `Stock` interface without worrying about details of the client component.

The IDL Compiler

Using IDL makes it possible to specify the architecture of a CORBA application in a way that is platform independent and language independent. At some point, however, the interface has to be translated from its language-neutral format into the target language used to implement the client or server component, for example C++. The tool responsible for performing this translation is known as the *IDL compiler*.

Here we are interested in the IDL compiler that translates IDL into C++. Generally speaking, IDL compilers are specialized to one particular language and platform. For example, Orbix C++ Edition for Windows NT supplies an IDL compiler that produces C++ code suitable for use with Microsoft Visual C++. If you want to translate IDL to Java instead, you can use the IDL compiler supplied with Orbix Java Edition.

The syntax for running the IDL compiler supplied with Orbix is as follows:

```
idl options IDLFile
```

A full list of *options* for the IDL compiler can be found by consulting the *Orbix C++ Programmer's Guide* and *Orbix C++ Programmer's Reference*. Two options in particular are used fairly frequently:

`-B`	Generate the extra code needed to support the *inheritance* approach to implementing an interface.
`-S`	Generate starting point code for the C++ classes used to implement the interfaces appearing in the IDL file.

For example, imagine you are a server programmer about to implement the IDL specified in `StockWatch.idl`. In that case it is likely that you will run the idl compiler as follows:

```
idl -B StockWatch.idl
```

resulting in the generation of the following three files:

```
StockWatch.hh
StockWatchC.cxx
StockWatchS.cxx
```

The `StockWatch.hh` file is a C++ header file that is included by the other two files. The `C` in `StockWatchC.cxx` stands for Client. This file contains the client stub code for StockWatch and is intended to be compiled and linked with the client application. The `S` in `StockWatchS.cxx` stands for Server. This file contains the server skeleton code for StockWatch and is meant to be compiled and linked with the server application.

Note: Both the stub and skeleton files, `StockWatchC.cxx` and `StockWatchS.cxx`, contain ORB-specific code. They are not intended to be edited by the application programmer and it is not recommended that you do so.

Another example of running the IDL compiler might include the `–S` flag, as follows:

```
idl -B –S StockWatch.idl
```

which generates the following files:

```
StockWatch.hh
StockWatchC.cxx
StockWatchS.cxx
StockWatch.ih
StockWatch.ic
```

In this case, two extra files are generated: `StockWatch.ih` and `StockWatch.ic`. These files contain starting point code for the classes that implement the StockWatch interfaces (in this example, there is just one class to implement the `Stock` interface). These two files are intended to be edited by the application developer. They form the basis for the implementation of CORBA objects.

However, this particular approach to implementing CORBA objects is not pursued in this chapter. Orbix provides a more powerful approach to generating starting point code, which is used instead.

Code Generation

The Orbix Code Generation Toolkit (OCGT) is a tool for generating code from IDL files. It is based on Ousterhout's Tool Command Language (Tcl)—a customizable scripting language. However it is *not* necessary to understand Tcl in order to use the OCGT.

In this chapter the OCGT is used to generate starting point code for the StockWatch application. The starting point generated by the OCGT is a complete client and server, requiring only a few modifications to be made by the application developer.

Before using the toolkit, you need to ensure that Orbix has been correctly configured and installed including the code generation option—check the installation guide for your platform.

The OCGT is run via the `idlgen` executable. A simple test of the `idlgen` executable can be made by running the following command:

```
   idlgen –list
```

A typical output from this command looks like this:

```
available applications are...

cpp_equal.tcl  cpp_op.tcl  cpp_random.tcl  stats.tcl
cpp_genie.tcl  cpp_print.tcl  idl2html.tcl
```

These toolkit applications are known as *genies*. A genie is a Tcl script that can generate output based on the contents of an IDL file. Genies can automate the generation of various coding patterns. For example, the genie `idl2html.tcl` takes an IDL file and convert it to HTML format with embedded links.

The genie that is of particular relevance to this chapter is `cpp_genie.tcl`. This genie can be used to generate comprehensive starting point code and has a number of options for supporting advanced features of Orbix.

Generating Starting Code

Perform the following steps to generate the starting point for this example:

1. Create a new directory, *OCGTExample*. Change directory to *OCGTExample* and create a new file, `StockWatch.idl`,containing the IDL shown in “Defining the Stock Interface” on page 27.
2. At a command-line prompt, enter the following command:

   ```
   idlgen cpp_genie.tcl -makefile -client -server \
        -interface StockWatch.idl
   ```

   (the backslash \ is used here to indicate continuation of the line). This command outputs the following lines to the screen:

   ```
   StockWatch.idl:
   cpp_genie.tcl: creating Stock_i.h
   cpp_genie.tcl: creating Stock_i.cxx
   cpp_genie.tcl: creating server.cxx
   cpp_genie.tcl: creating client.cxx
   cpp_genie.tcl: creating call_funcs.h
   cpp_genie.tcl: creating call_funcs.cxx
   cpp_genie.tcl: creating it_print_funcs.h
   cpp_genie.tcl: creating it_print_funcs.cxx
   cpp_genie.tcl: creating it_random_funcs.h
   cpp_genie.tcl: creating it_random_funcs.cxx
   cpp_genie.tcl: creating makefile
   cpp_genie.tcl: creating makefile.inc
   ```

The genie generates all the files you need, including the `makefile`. By default, a working dummy application is generated that can be compiled and run right away. You can modify this dummy application to provide the appropriate functionality for the new CORBA application, as detailed in the following sections.

Dummy Application Implementation

By default (or when the `-complete` option is specified) the OCGT generates client and server code, which is sufficiently complete to compile and run immediately. For example, the following command:

```
idlgen cpp_genie.tcl -makefile -client -server \
     -interface -complete StockWatch.idl
```

when applied to the file `StockWatch.idl` generates a complete demonstration application. Simply follow the steps of "Compile and Run the Demonstration" on page 46 to compile and run the demonstration.

The OCGT provides a simple default implementation of the client and server functionality.

Dummy Implementation of Server

The *dummy server* implements all of the interfaces appearing in the supplied IDL. Each operation and attribute of every interface is programmed to print out the parameters it was invoked with. The operation or attribute then generates random data for the return value and out parameters. Sometimes, on a random basis, the operation throws a CORBA exception instead.

The dummy server's `main()` function creates one instance of a CORBA object for every interface it has implemented, before entering a CORBA event loop.

Dummy Implementation of Client

The *dummy client* obtains an object reference for each of the CORBA objects in the server (one per interface). It then invokes every operation on every interface, passing random data as parameters.

Client Code

The starting code for the client is contained in the `client.cxx` generated file. A few lines of code have to be modified to implement the StockWatch client. In this example, it is assumed that there is only one `Stock` object in the server. The client connects to the single `Stock` object and invokes its operations.

Edit the `client.cxx` file.

Search for the `#include "call_funcs.h"` line and replace it with the `#include "StockWatch.hh"` line.

Search for the lines where the `call_Stock_`*OperationName*`()` function is called. Delete these lines and replace them with the lines of code highlighted in bold font below:

```
//---------------------------------------------------------------
// Edit the idlgen.cfg to have
// your own copyright notice placed here.
//---------------------------------------------------------------

//---------------------------------------------------------------
// File:client.cxx
// Description:Client main function.
//---------------------------------------------------------------

//--------
// #include's
//--------
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
1      #include "StockWatch.hh"

//--------
// Forward declaration of static functions.
//--------

//---------------------------------------------------------------
// Function:main()
// Description:Client main function.
//---------------------------------------------------------------
```

```
      int
      main(int argc, char **argv)
      {
2        Stock_var obj1;
         //--------
         // Set Orbix diagnostics level
         //--------
         CORBA::Orbix.setDiagnostics(1);

         ifstream iorfile;
         char myIor [ 2048 ];
         CORBA::Object_var tObj;

         try {
3           iorfile.open ("./Stock.ior");
            iorfile >> myIor;
            iorfile.close();
4           tObj = CORBA::Orbix.string_to_object(myIor);
            if ( CORBA::is_nil(tObj) ) {
               cerr << "Object reference is nil" << endl;
               cerr << "Have you run the server to create the IOR files?"
                    << endl;
               exit(1);
            }
            obj1 = Stock::_narrow(tObj);
         } catch (CORBA::SystemException sysEx ) {
            cerr << "Unexpected Exception: " << sysEx << endl;
            exit(1);
         } catch (...) {
            cerr << "Unknown Exception caught " << endl;
            exit (1);
         }

         //--------
         // Invoke the operations and attributes
         //--------
         try {
            CORBA::String_var theSymbolV, theDescriptionV;
5           theSymbolV = obj1->getSymbol();
            theDescriptionV = obj1->getDescription();
            Money theMoney = obj1->getCurrentPrice();

            cout << "Symbol: " << theSymbolV << endl;
```

```
        cout << "Description: " << theDescriptionV << endl;
        cout << "Current Price: " << theMoney << endl;
    }
    catch (CORBA::Exception& ex) {
        cerr << "error: " << ex << endl;
    }

    //--------
    // Terminate gracefully.
    //--------
    return 0;
}
```

The important steps are numbered and can be explained as follows:

1. The client includes the `StockWatch.hh` file. This header gives access to the types and interfaces defined in the `StockWatch.idl` stub code. It also recursively includes the `corba.h` header that declares the Orbix runtime programming interface.
2. The `obj1` object reference is declared to be of `Stock_var` type. The `Stock_var` type is a smart pointer class that holds references to `Stock` objects. Syntactically, its behavior imitates the `Stock*` pointer type so that you can, for example, dereference members of the `Stock` class as follows:

   `(*obj1).getSymbol()`

   or use the dereferencing operator `->` as follows:

   `obj1->getSymbol()`
3. A *stringified object reference*, `myIor`, is read from a local file, `Stock.ior`. It is assumed that the stringified object reference has already been written to `Stock.ior` by the server program.

   This method of distributing an object reference (writing to a file) is suitable only for simple demonstrations. Realistic applications use the CORBA Naming Service to distribute object references to clients.
4. The `myIor` stringified object reference is converted to an object reference, `obj1`, using the standard `CORBA::ORB:string_to_object()` function.

Locating a remote CORBA object is a critical step for the client programmer. There are three main approaches supported by Orbix:

- Writing stringified object references to a file (or files).

  This is the approach used throughout this chapter because it is relatively easy to use.
- The CORBA Naming Service.

  This approach is more useful for realistic applications—see "The Naming Service" on page 95.
- The `_bind()` mechanism. (Deprecated.)

5. The `obj1` object reference allows you to make remote invocations on a `Stock` CORBA object—the C++ member functions have the same name as the corresponding `Stock` IDL operations.

   The return values of `getSymbol()` and `getDescription()` are assigned to variables of `CORBA::String_var` type. The `CORBA::String_var` type is a smart pointer equivalent for the `char*` type. It relieves you of the responsibility of deleting the string, because the `CORBA::String_var` destructor does it for you.

IDL-to-C++ Mapping

The return type of `getSymbol()` and `getDescription()` is declared to be the `string` IDL type, and the return type of `getCurrentPrice()` is the `float` IDL type. These IDL data types have simple counterparts in C++. The standard *IDL to C++ mapping* defines the correspondence as follows:

IDL	C++
`string`	`char *`
`float`	`CORBA::Float`
`interface Stock`	`class Stock`
`getSymbol()` operation	`getSymbol()` member function

There is a close parallel between the definition of interfaces in IDL and the definition of classes in C++.

Exception Handling

All invocations on a remote CORBA object should be enclosed within a `try` block similar to the following:

```
try {
      // Remote invocations go in here.
}
catch (CORBA::SystemException &ex) {
      cerr << ex << endl;
}
```

The `CORBA::SystemException` class is the base class for exceptions raised within the ORB runtime. An instance, `ex`, of a system exception is caught by reference, which is the recommended way to catch exceptions in the C++ language. The exception `ex` can then be printed directly to an output stream using the << operator, because the ORB runtime defines the appropriate overloaded form of this operator.

You must get used to the idea that enclosing CORBA invocations in such a `try/catch` block is necessary. Because of the uncertainty of what can happen to the network separating client from server, exception conditions are more frequent in a distributed system than in a stand-alone application.

Server Code

The starting point for the server code is contained in the three files `Stock_i.h`, `Stock_i.cxx` and `server.cxx` generated by `cpp_genie.tcl`.

The main purpose of the server application is to provide an implementation of the interfaces defined in the `StockWatch.idl` file. In this example there is just one interface to implement, the `Stock` interface. The class that implements `Stock` is called `Stock_i` and is defined in the files `Stock_i.h` and `Stock_i.cxx`.

Note: The convention followed in this book (and, by default, the code generation toolkit) is that an interface called *InterfaceName* is implemented by the class *InterfaceName*`_i`. This convention is adopted for convenience only; you are free to call the implementation class whatever you like.

The code in `server.cxx` contains the `main()` function and carries out the steps to initialize the ORB.

Implementing the Stock Interface

In this sample implementation of the `Stock` interface, the object's business logic is not provided. Instead, for the purpose of illustration, the `Stock_i` class is hardcoded to return fixed information about a single stock: the IONA stock. Later on, in "Integration with a Database" on page 79, a more realistic implementation is provided that retrieves the stock information from a database.

To complete the implementation of the `Stock_i` class, you have to supply the function bodies for `getSymbol()`, `getDescription()` and `getCurrentPrice()`. This requires you to edit just the `Stock_i.cxx` file (the `Stock_i.h` file is already complete).

Replace the existing code in the function bodies with the lines highlighted in bold font below:

```
// C++
// File: Stock_i.cxx
. . .
// . . . Some lines of code omitted . . .
//-----------------------------------------------------------------
// Function:getSymbol()
//
// Description: Implementation of the IDL operation
//    Stock::getSymbol().
//-----------------------------------------------------------------

char *
Stock_i::getSymbol(
      CORBA::Environment &)
         throw(CORBA::SystemException)
{
   return CORBA::string_dup("IONA");
}

//-----------------------------------------------------------------
// Function:getDescription()
// Description: Implementation of the IDL operation
//    Stock::getDescription().
```

```
//---------------------------------------------------------------

char *
Stock_i::getDescription(
      CORBA::Environment &)
         throw(CORBA::SystemException)
{
   return CORBA::string_dup("IONA Technologies Inc.");
}

//---------------------------------------------------------------
// Function:getCurrentPrice()
//
// Description: Implementation of the IDL operation
//    Stock::getCurrentPrice().
//---------------------------------------------------------------

Money
Stock_i::getCurrentPrice(
      CORBA::Environment &)
         throw(CORBA::SystemException)
{
   return (Money) 18.0[1];
}
```

Although the operations are declared without parameters in IDL, they have one parameter, the `CORBA::Environment` parameter, when mapped to C++. This parameter can safely be ignored because it is defaulted in the `Stock_i.h` header file and is not used in any of the example code here. The `CORBA::Environment` parameter is needed only for legacy platforms that do not support native C++ exception handling—see the *Orbix C++ Programmer's Guide* and the *Orbix C++ Programmer's Reference*.

The preceding functions that return a string, `char*`, do not return the string directly but copy the string and return a pointer to the copy. For example, the `getSymbol()` function returns `CORBA::string_dup("IONA")` instead of returning the string literal, `IONA`, directly.

This is a basic memory management rule in CORBA: when returning data from an operation, always returning a *copy* of the data, not the original data itself.

1. The price (in dollars) of IONA stock at the time of its flotation in April 1997.

This is consistent with good coding practice in C++. You can never predict when calling code is going to delete the return value. By passing back a copy of the data, you prevent the CORBA object's data from being prematurely deleted.

Simple data types like `Money` (a floating point number) are passed by value, so no copying is required.

Implementing the Server Main Function

The server `main()` function is implemented in the `server.cxx` file and performs miscellaneous initialization, including initialization of the ORB. The `server.cxx` file generated by the OCGT is complete and needs no additions or modifications for this example. However, it is instructive to have a look at this file to gain an understanding of the main steps involved in initializing the server.

```
//-----------------------------------------------------------------
// Edit the idlgen.cfg to have
// your own copyright notice placed here.
//-----------------------------------------------------------------

//-----------------------------------------------------------------
// File:server.cxx
// Description:Server main function.
//-----------------------------------------------------------------

//--------
// #include's
//--------
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>
#include "Stock_i.h"

//-----------------------------------------------------------------
// Function:main()
// Description:Main Function of the server
//-----------------------------------------------------------------

int
main(int argc, char **argv)
{
        // Local Variables
        CORBA::ORB_var orbVar;
        CORBA::BOA_var boaVar;
        try {
1          orbVar = CORBA::ORB_init ( argc , argv, "Orbix");
           boaVar = orbVar->BOA_init ( argc, argv, "Orbix_BOA");
        } catch (CORBA::SystemException e) {
           cerr << "Unexpected System Exception :" << e << endl;
           exit (1);
        } catch (...) {
           cerr << "Unexpected Exception." << endl;
           exit (1);
        }

        Stock_var obj1;

        //--------
        // Initialise Orbix.
        //--------
2       orbVar->setDiagnostics(1);
        try {
3          boaVar->impl_is_ready("StockWatchSrv", 0);
        } catch(CORBA::SystemException &ex) {
           cerr << "impl_is_ready() failed" << endl
              << ex << endl;
           exit(1);
        }
4       obj1 = Stock_i::_create("Stock-1");
        //--------
        // Application-specific initialisation.
        //--------
        ofstream ofile;
        ofile.open ("./Stock.ior");
5       ofile << obj1->_object_to_string();
        ofile.close ();

        //--------
        // Main event loop.
        //--------
        try {
6          CORBA::Orbix.processEvents(-1);
        } catch(CORBA::SystemException &ex) {
           cerr << "processEvents() failed" << endl
```

```
            << ex << endl;
         exit(1);
      }

      //--------
      // Terminate.
      //--------
      return 0;
   }
```

The main steps in the server `main()` function are as follows:

1. References to an initial ORB object and an initial BOA object are obtained using the `CORBA::ORB_init()` and `CORBA::ORB::BOA_init()` functions, respectively. The BOA object returned from `BOA_init()` offers functionality equivalent to the now deprecated `CORBA::Orbix` static BOA instance.
2. The `CORBA::ORB::setDiagnostics()` function is called, with diagnostic level equal to `1`. This is an optional step that sets the level of diagnostics generated by Orbix and written to the standard output. There are three levels of diagnostics:

`0`	No diagnostics.
`1`	Generate diagnostic output each time a connection is opened or closed.
`2`	Generate diagnostic output for every *request* or *reply* that enters or leaves the application.

   The default level of diagnostics is `1`. The higher level of diagnostics, `2`, can be a valuable aid to debugging a distributed application.
3. The `CORBA::BOA::impl_is_ready()` function is called. This performs initialization of the basic object adapter (BOA) and tells the Orbix daemon that the server is ready to receive requests. From this point on, the server has internally set the values of the *server name*, *host* and *port* that it is using (all of the information required to identify and locate a server).

The function signature for `impl_is_ready()` is:

```
// C++
void CORBA::BOA::impl_is_ready(
    CORBA::ImplementationDef_ptr serverName = "",
    CORBA::ULong timeOut=CORBA::ORB::DEFAULT_TIMEOUT,
    CORBA::Environment& = IT_chooseDefaultEnv ()
);
```

The `serverName` identifies the server process. The `timeOut` specifies how many milliseconds the server remains blocked in an event loop. The value specified here is zero, implying that the event loop is not started at this point.

4. A single CORBA object of `Stock_i` type is instantiated and assigned to the `obj1` object reference. The `Stock_i::_create(`*marker*`)` function delegates the creation of a new CORBA object to the `Stock_i` constructor. A marker is a unique identifier for the CORBA object.

   This use of `_create()` is an idiom of the code generator, not standard CORBA. If you prefer to call the constructor directly, you can edit the generated code (in `Stock_i.h` and `Stock_i.cxx`) to make the constructor public and change the signature of the constructor. The advantage of the OCGT idiom is that it can hide different approaches to object instantiation. See the *Orbix C++ Programmer's Guide* and the *Orbix Code Generation Toolkit Programmer's Guide* for details.

   Once a CORBA object has been instantiated, no further initialization is necessary. In Orbix, instantiation implicitly makes a CORBA object known to the ORB (enabling the object to receive invocations).

5. The `obj1` object reference is converted to a string by invoking `_object_to_string()` and the server writes the string to a local file, `Stock.ior`.

   This is a simple approach to distributing object references from servers to clients. Clients read the object reference directly from the `Stock.ior` file, assuming that they have access to the file via a networked file system or similar. This approach is only suitable for simple demonstrations—realistic applications would use the CORBA Naming Service.

6. The `CORBA::BOA::processEvents()` function is called. This call blocks as the server enters an internal event loop and makes itself ready to do useful work. From this point on, the server is able to accept connection attempts from clients. It can also receive and process remote invocations.

The `processEvents()` function unblocks and returns if there is no activity in the server (no new connection attempts, no incoming invocations) for the length of time specified in the inactivity timeout. The default inactivity timeout is 60 seconds, but the value can be customized via the optional argument to `processEvents()`.

Some sample values for the inactivity timeout are as follows:

`processEvents(0)`	No events processed.
`processEvents()`	Default timeout is used (60 seconds)
`processEvents(30*1000)`	Specifies timeout in milliseconds (for example, 30 seconds).
`processEvents( CORBA::Orbix.INFINITE_TIMEOUT)`	Never times out.

If a server does time out and exit, this does not ordinarily cause problems. Servers are normally configured so that they can be restarted on demand when needed by clients. In applications where the latency of a server restart is unacceptable, an infinite timeout can be used.

Compile and Run the Demonstration

The `makefile` generated by the code generation toolkit has a complete set of rules for compiling both the client and server applications. To compile the client and server:

Windows

At a command-line prompt, from the *OCGTExample* directory enter:

```
> nmake
```

UNIX

At a command-line prompt, from the *OCGTExample* directory enter:

```
% make
```

Run the application as follows:

1. Run the Orbix daemon.

   The Orbix daemon is responsible for bootstrapping connections between CORBA clients and servers and can, if necessary, activate dormant servers on demand. Information about CORBA servers is stored in the the *Implementation Repository*, a database of CORBA servers maintained by the Orbix daemon. Exactly one Orbix daemon runs on each server host.

   Open a new MS-DOS prompt, or `xterm` window (UNIX).

   Windows

   ```
   > orbixd
   ```

   UNIX

   ```
   % orbixd
   ```

   The Orbix daemon runs in the foreground and logs its activities to this window.

2. Register the server with the daemon.

   Every Orbix server must be registered with the Orbix daemon before it runs for the first time. Registration only needs to be performed once per server.

Open a new MS-DOS prompt, or `xterm` window (UNIX).

Windows

At a command-line prompt, from the *OCGTExample* directory enter:

```
> nmake putit
```

UNIX

At a command-line prompt, from the *OCGTExample* directory enter:

```
% make putit
```

This script outputs the following lines to the screen:

```
putit StockWatchSrv OCGTExample/server.exe
[264:New Connection
(foobar.iona.ie,IT_daemon,*,userid,pid=275,optimised) ]
```

The makefile uses the Orbix `putit` utility to register the server—see the *Orbix Administrator's Guide* for details.

3. Run the server program.

   Open a new MS-DOS prompt, or `xterm` window (UNIX). From the *OCGTExample* directory enter the name of the executable file—`server.exe` (Windows) or `server` (UNIX).

   The server outputs the following lines to the screen:

```
[StockWatchSrv:New Connection
(foobar.iona.ie,IT_daemon,*,userid,pid=275,optimised) ]
[StockWatchSrv:Server "StockWatchSrv" is now available
to the network ]
[ Configuration tcp/1591/cdr ]
```

   The server performs the following steps when it is launched:

   - It instantiates and activates a single `Stock` CORBA object.
   - The stringified object reference for the `Stock` object is written to the local `Stock.ior` file.
   - The server opens an IP port and begins listening on the port for connection attempts by CORBA clients.

4. Run the client program.

   Open a new MS-DOS prompt, or `xterm` window (UNIX). From the *OCGTExample* directory enter the name of the executable file—`client.exe` (Windows) or `client` (UNIX).

The client outputs the following lines to the screen:

```
[335:New Connection
(foobar.iona.ie,IT_daemon,*,userid,pid=275,optimised) ]
[335:New IIOP Connection (foobar.iona.ie:1591) ]
Symbol: IONA
Description: IONA Technologies Inc.
Current Price: 18
```

The client performs the following steps when it runs:

- ♦ It reads the stringified object reference for the `Stock` object from the local `Stock.ior` file.
- ♦ It converts the stringified object reference into an object reference.
- ♦ It calls the remote `Stock` operations by invoking on the object reference. This causes a connection to be established with the server and the remote invocation to be performed.

5. When you are finished, terminate all processes.

   Shut down the server by typing Ctrl-C in the window where it is running.

The passing of the object reference from the server to the client in this way is suitable only for simple demonstrations. Realistic server applications use the CORBA Naming Service to export their object references instead.

Smart Pointers and Dumb Pointers

In the course of CORBA development, it is frequently necessary to create and use heap-allocated data. The IDL-to-C++ mapping defines a set of smart pointer types that make it easier for the developer to manage heap-allocated memory and avoid memory leaks.

This section describes the following types of smart pointer:

- Smart pointer for strings.
- Smart pointer for object references.

Smart Pointer for Strings

For the basic string IDL type there are two representations in C++:

IDL	C++	Kind of pointer
string	char *	Dumb pointer
	CORBA::String_var	Smart pointer

The string IDL type maps to a conventional C++ string, that is a null terminated array of char. Therefore, just like a normal string, it is referenced using a char* pointer. This is the dumb pointer referred to in the table above.

The CORBA::String_var smart pointer aids memory management. In order to appreciate its role, it is necessary to begin with a few remarks about the memory management of strings.

When you are writing client code you should keep in mind that return values from CORBA invocations are dynamically allocated and must therefore be deleted when you are finished with them. In particular, you must avoid writing code such as the following:

```
// C++
// Wrong! – memory will be leaked.
cout << "Symbol: " << obj1->getSymbol() << endl;
```

This code is incorrect and leaks memory. The string returned from getSymbol() is dynamically allocated so it must be deleted.

Allocation and deletion of CORBA strings should be done with the aid of the following three helper functions:

```
static char * CORBA::string_alloc(CORBA::ULong len);
static void CORBA::string_free(char *str);
static char * CORBA::string_dup(const char *p1)
```

The CORBA::string_alloc() and CORBA::string_free() functions should be used instead of new and delete to allocate and delete CORBA strings. This ensures portability of your code across all platforms and is required for CORBA compliancy. The function CORBA::string_dup() is a convenient function that combines memory allocation and string copying into a single step.

Armed with these functions, it is now possible to rewrite the above code fragment correctly:

```
// C++
// The correct way:
char * tmpP = obj1->getSymbol();

cout << "Symbol: " << tmpP << endl;
CORBA::string_free(tmpP);
```

This version of the code avoids leaking the memory associated with the returned string.

There are, however, some drawbacks to this memory management approach. For example, the requirement to release the string using `CORBA::string_free()` can easily be forgotten in the midst of a large volume of code. More seriously, if an exception is raised before the `CORBA::string_free()` line is reached the string's memory is leaked.

The `CORBA::String_var` smart pointer type is introduced to make it easier to avoid memory leaks. The use of smart pointer classes is a common C++ trick. At a minimum, it involves overloading the operator for dereferencing `operator*`. The smart pointer is designed to imitate the syntax and semantics of an ordinary pointer. However, in certain respects it displays smart behaviour not normally associated with an ordinary (dumb) pointer.

The smart behaviour of `CORBA::String_var` relates to memory management. It has the intelligence to automatically delete the string it is pointing at, once it goes out of scope. This is illustrated by the following short example:

```
// C++
// Automatic memory management:

{ // Begin local scope
    CORBA::String_var tmpV = obj1->getSymbol();

    cout << "Symbol: " << tmpV << endl;

} // String is auto-deleted at end of this scope
```

There is no need to call `CORBA::string_free()` when using the `tmpV` smart pointer. As soon as `tmpV` goes out of scope, its destructor calls `CORBA::string_free()`—safely disposing of the string.

The nett effect of using the `CORBA::String_var` type is that dynamically allocated strings behave like automatic variables.

Smart Pointer for Object References

An object reference in C++ can also be referenced either by a smart or a dumb pointer. For example, when the `Stock` interface is mapped to C++ two pointer types are made available:

IDL	**C++**	**Kind of Pointer**
`Stock`	`Stock_ptr`	Dumb pointer
	`Stock_var`	Smart pointer

A dumb pointer of `Stock_ptr` type behaves essentially like an ordinary pointer. You can think of it as being synonymous[2] with the `Stock*` pointer. The dumb pointer type is typically used on the server side when implementing the methods of a CORBA object.

In order to appreciate the significance of the smart pointer `Stock_var`, it is necessary to present a short, and incomplete, discussion of the memory management of object references. For a comprehensive discussion of memory management for object references consult the *Orbix C++ Programmer's Guide*.

Consider the following code extract, which shows an incorrect example that ignores memory management issues:

```
// C++
// Client code
...
main() {
   Stock_ptr obj1;

   try {
      iorfile.open ("./Stock.ior");
      iorfile >> myIor;
      iorfile.close();
      obj1 = CORBA::Orbix.string_to_object(myIor);
      ...
```

2. In fact, in the current version of Orbix it is `typedef`'ed to be `Stock*`. Nonetheless, you should always use `Stock_ptr` for portability and CORBA compliancy.

```
   }
   catch(CORBA::SystemException &ex) { exit(1); }
   ...
   // Wrong! – we forgot to release the object reference
   // <----- 'obj1' gets leaked when we go out of scope
}
```

The problem with this example is that an object reference gets created when the call is made to `string_to_object()` but is subsequently never deleted.

To help you manage the memory associated with `Stock` object references, CORBA supplies two functions:

```
static Stock_ptr Stock::_duplicate(Stock_ptr obj)
static void CORBA::release(CORBA::Object_ptr obj);
```

The `Stock::_duplicate()` static method is called to make a copy of a given object reference. The `CORBA::release()` method is called to delete an object reference. For convenience, you can think of these functions as copying and deleting instances of object references. In reality, there is only one instance of a particular object reference—the ORB uses reference counting to create the illusion that references are being copied and deleted.

On the server side, an object reference is created by calling the implementation class constructor. For example, a `Stock` object can be obtained by creating an instance of the `Stock_i` class.

With the help of the preceding memory management functions, the code extract can be written correctly as follows:

```
// C++
// Client code
...
main() {
   Stock_ptr obj1;

   try {
      iorfile.open ("./Stock.ior");
      iorfile >> myIor;
      iorfile.close();
      obj1 = CORBA::Orbix.string_to_object(myIor);
   }
   catch(CORBA::SystemException &ex) { exit(1); }
   ...
   // The object reference is released here, avoiding leaks.
   CORBA::release(obj1);
}
```

The Stock_var smart pointer is a class with overloaded operators that is designed to imitate the syntax of an ordinary Stock* dumb pointer.A Stock_var variable automatically calls CORBA::release() on the object reference it is pointing at, as soon as the Stock_var variable goes out of scope.

For example, the code extract can be rewritten with the help of the Stock_var type, as follows:

```
// C++
// Client code
...
main() { // Begin scope of 'main()'
   Stock_var obj1;

   try {
      obj1 = Stock::_bind("Stock-1:StockWatchSrv", host);
   }
   catch(CORBA::SystemException &ex) { exit(1); }
   ...
   // Object reference is auto-released at the end of scope
} // End scope of 'main()'
```

In this example there is no need to call CORBA::release() explicitly because it is called automatically in the destructor of obj1.

Smart Pointers for Other Data Types

Smart pointer types are provided for all CORBA data types that can be allocated on the heap. These types are explained as they arise. For a comprehensive discussion of CORBA data types and smart pointers (_var types) consult the *Orbix C++ Programmer's Guide*.

3

Compound Types and Exceptions

The StockWatch example is extended to make use of the IDL compound types struct and sequence. User exceptions are also introduced and discussed.

Extending the Example

The version of the StockWatch demonstration developed in Chapter 2 allowed you to find out the price of a stock at a particular point in time. In this chapter the example is extended to implement a history feature, allowing you to access a list of recent prices for a particular stock.

The StockWatch IDL must be updated to implement the history feature. Two new IDL operations to the Stock interface are added:

`recordSale()`	Record the price at which the stock has just traded and add the information to the history list.
`getRecentPrices()`	Retrieve the history of recent prices for this particular stock.

The `recordSale()` operation is added to the interface to make it easy for CORBA clients to append the latest price to the history list.

The introduction of the history feature also requires the use of more flexible data types, so that a list of prices, which may be of variable length, can be specified as a return value. This need is satisfied by the IDL *compound types* and this chapter introduces the `struct` and `sequence` data types needed to pass around the history of prices.

The other topic introduced in this chapter is exception handling. How IDL allows you to specify *user-defined exceptions* via the `exception` data type is explained. The advantage of the IDL exception handling mechanism is that it integrates cleanly with the exception handling mechanisms of recent computer languages such as Java and C++.

The extended IDL for StockWatch that features support for price history and demonstrates exception handling is as follows:

```
// IDL
// File: StockWatch.idl

typedef float Money;
typedef string Symbol;
typedef string Date;

// Structs
struct PriceInfo {
      Money m_price;
      Date  m_when;
};
typedef sequence<PriceInfo> PriceInfoSeq;

// Exceptions
exception rejected {
      string m_reason;
};

// Interfaces
interface Stock {
      Symbol getSymbol();
      string getDescription() raises (rejected);
      Money getCurrentPrice() raises (rejected);
      PriceInfoSeq getRecentPrices() raises (rejected);
      void recordSale(in Money price) raises (rejected);
};
```

The new features of this IDL, such as parameter passing modes, `struct` and `sequence` data types, and exception handling are discussed in the sections below.

Parameter Passing Modes

The operation `recordSale()`, introduced in the above IDL, is the first example of an operation that passes a parameter. The signature of the operation in IDL is as follows:

```
// IDL
void recordSale(in Money price) raises (rejected);
```

The clause `raises(rejected)` is associated with exception handling and is ignored for now. The operation `recordSale()` takes a single parameter called `price` of type `Money` (where `Money` is `typedef`'ed to be of type `float`).

The signature of this operation is nearly identical to the syntax of a function declaration in C++ or Java, except for the appearance of the `in` keyword, an example of a *parameter passing mode*. The `in` keyword is used to indicate that the parameter `price` is passed from the client to the server.

There are three different parameter passing modes that can be specified in IDL, as follows:

Mode	**Meaning**
`in`	Parameter passes from client to server
`out`	Parameter passes from server back to client
`inout`	Parameter passes from client to server and back again

It is compulsory to specify the parameter passing mode for each operation parameter. If the mode is not specified for any parameter it is treated as a syntax error in the IDL.

To illustrate what happens for each of the different modes, consider the following fragment of IDL:

```
// IDL
interface Foo {
      void op(
         in string inParam,
         inout string inoutParam,
         out string outParam);
};
```

Figure 3.1 shows schematically how the various parameters are passed as `op()` is invoked. When the invocation is made, a request message is sent from client to server containing, amongst other things, the parameters `inParam` and `inoutParam`. When the server has finished processing the invocation it sends back a reply message from server to client including the parameters `inoutParam` and `outParam`.

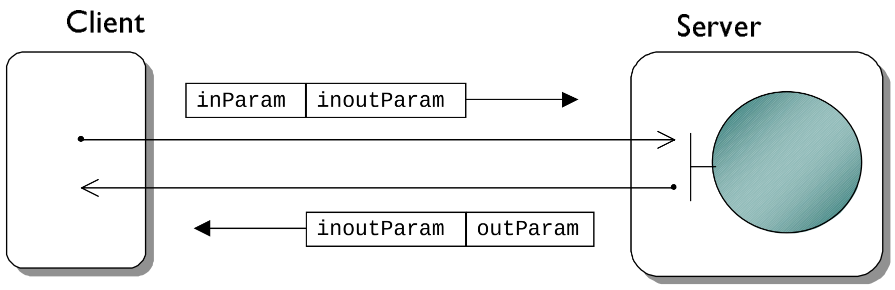


Figure 3.1: *Parameters Passed as Foo::op() is Invoked.*

The struct Data Type

IDL Syntax

Consider the `struct` defined in the file `StockWatch.idl`:

```
// IDL
struct PriceInfo {
     Money m_price;
     Date  m_when;
};
```

The syntax of `struct` declaration in IDL is virtually identical to the syntax used in the C programming language. The declaration above results in the definition of a new IDL type called `PriceInfo`. An arbitrary number of struct members can be declared within the curly braces. The semicolon at the end of the closing brace is mandatory.

Remember that IDL is a purely declarative language and there is no support in the language for manipulating data types such as this `PriceInfo` struct. The sole purpose of defining this struct in IDL is to allow it to be used as the parameter or return value of an IDL operation.

Fixed and Variable Length Structs

The CORBA standard makes an important distinction between two different kinds of struct. These different kinds of struct are known as *fixed* and *variable length* structs. Consider the following example of a struct:

```
struct FixedStruct {
     long l;
};
```

This defines a fixed length struct `FixedStruct`, so called because the length of the struct is known at compile time. It contains one member, the long integer `l`, therefore its total length is 32 bits[1].

1. The length of the IDL `long` type is defined to be exactly 32 bits.

Consider another example of a struct:

```
struct VariableStruct {
     string s;
};
```

This defines the variable length struct `VariableStruct`. In this case the length of the struct is not known at compile time. It has just one member, the variable length string `s`, therefore the length of the struct is also variable.

Although there is no obvious syntactical difference between the two examples, they differ in one important respect: the CORBA rules of memory management for fixed and variable length types are different. These rules of memory management are so important that it is helpful to think of *fixed length structs* and *variable length structs* as distinct IDL data types.

A discussion of memory management issues is postponed to the memory management section below.

C++ Mapping

Consider the `PriceInfo` struct once more. Under the action of the C++ compiler, it is mapped to a corresponding C++ struct called `PriceInfo`. Here is an example of a C++ code extract that uses the struct:

```
{
// 1. Allocate space for the PriceInfo struct (i.e. declare it)
PriceInfo myPriceInfo;

// 2. Initialize the PriceInfo struct
myPriceInfo.m_price = (CORBA::Float) 18.0;
myPriceInfo.m_when = CORBA::string_dup("April 1997");

// Use the PriceInfo struct
cout << "The price was: " << myPriceInfo.m_price << endl;
cout << "On the date: " << myPriceInfo.m_when << endl;

// 3. Deallocate the PriceInfo struct
// Happens automatically when leaving the current scope
}
```

In addition to the C++ struct called PriceInfo, the IDL compiler also generates definitions for two pointer types: the dumb pointer PriceInfo_ptr and the smart pointer PriceInfo_var. To see how a smart pointer is used in practice, the C++ code extract is rewritten to use the type PriceInfo_var instead an automatic variable, as follows:

```
{
// 1. Allocate space for the PriceInfo struct (i.e. declare it)
PriceInfo_var myPriceInfoV = new PriceInfo();

// 2. Initialize the PriceInfo struct
myPriceInfoV->m_price = (CORBA::Float) 18.0;
myPriceInfoV->m_when = CORBA::string_dup("April 1997");

// Use the PriceInfo struct
cout << "The price was: " << myPriceInfoV->m_price << endl;
cout << "On the date: " << myPriceInfoV->m_when << endl;

// 3. Deallocate the PriceInfo struct
// Happens automatically when leaving the current scope
}
```

Note that when using the PriceInfo_var type you use pointer syntax throughout. The only surprise comes at the last step, when it comes to deallocating the PriceInfo struct. Since PriceInfo_var is a smart pointer type it automatically deletes the memory pointed at as soon as it goes out of scope.

Deep Copy and Recursive Delete

There is more than one way to make a copy of the struct PriceInfo. For example, struct PriceInfo contains m_when, a pointer to a string. When the struct is copied there is the question of what happens to the string: whether just the pointer is copied, or whether a the whole string is copied. In other words, the copy operation might be a *shallow* copy or a *deep* copy.

CORBA requires that the copy is a deep copy. How this is achieved is an implementation detail.

This means that code such as the following performs a deep copy:

```
// C++
PriceInfo_var myPriceInfoV1 = new PriceInfo();

myPriceInfoV1->m_price = (CORBA::Float) 18.0;
myPriceInfoV1->m_when = CORBA::string_dup("April 1997");

PriceInfo_var myPriceInfoV2 = new PriceInfo();

// Perform a deep copy of struct 1 to struct 2
*myPriceInfoV2 = *myPriceInfov1;
```

That last line can also be written:

```
// Perform a deep copy of struct 1 to struct 2
myPriceInfoV2 = myPriceInfov1;
```

having the same effect. The smart pointer type `PriceInfo_var` is smart enough to realise that what you want is not a copy of the pointer to the struct but a deep copy instead.

A similar question can be posed with respect to *deletion* of a struct. If a struct contains a string, will the deletion of the struct also result in the deletion of the member string or must the string be deleted as a separate step? CORBA specifies that deleting the struct also results in deletion of the member string. In fact, it is a general rule for compound CORBA data types that deletes are recursive. With any highly recursive IDL data type, it is only necessary to delete the top level item for all associated memory to be deallocated[2].

2. An exception to this rule is the case where you have allocated the data yourself and used some special options in the constructor.

The sequence Data Type

IDL Syntax

An IDL sequence is used to define what is effectively a variable length array. A simple example of a sequence can be defined as follows:

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

This defines a new IDL type `LongSeq` which can be used to hold a variable number of IDL longs. A sequence is known within IDL as a *template type* because the declaration of a sequence is parameterized by another IDL type. This should not be confused with the concept of a C++ template; a sequence need not even map to a C++ template.

You are allowed to declare sequences of any IDL type, not just basic types such as long declared above. For example, the extension to `StockWatch.idl` declares the following sequence:

```
// IDL
struct PriceInfo {
      Money m_price;
      Date  m_when;
};
typedef sequence<PriceInfo> PriceInfoSeq;
```

In fact, it is even permissible to define sequences of sequences:

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

which allows you to get around the limitation that sequences are only one-dimensional.

In all of these examples, the sequence is `typedef`'ed to give it a name. In fact it is usually mandatory to give a sequence a name using a `typedef`. Thus the following fragment of IDL is syntactically incorrect:

```
// IDL
interface Foo {
      // Wrong! Anonymous sequence not allowed!
      void op(in sequence<long> theSeq);
};
```

and would give rise to a syntax error if you tried to pass it through the IDL compiler.

However, there is one notable exception to the rule that anonymous sequences are not allowed. The syntax of IDL allows a very special case where a sequence is being defined as a member of a struct:

```
// IDL
struct Node {
      string nodeName;
      sequence<Node> subNodes;
};
```

This fragment appears to violate IDL syntax but, because of its potential usefulness, this declaration syntax *is* allowed as a special case in IDL.

Variable Length Sequence

In the case of sequences, there is no need to make the distinction between fixed and variable length types as was done in the case of IDL structs. This is because sequences are inherently of variable length.

C++ Mapping

A sequence is mapped to a class in C++. This C++ class is designed to be similar to a one-dimensional array, except that it has two extra attributes: a `length()` and a `maximum()`. This is illustrated in Figure 3.2.

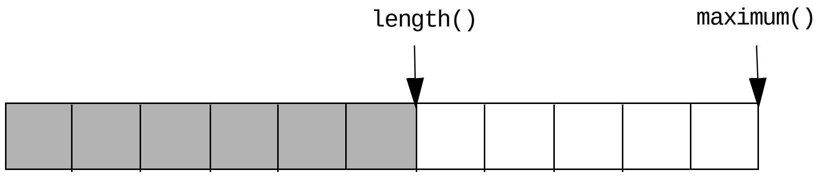


Figure 3.2: *The Length and Maximum of a Sequence Type*

The meaning of the `length()` and `maximum()` attributes is as follows:

- `length` The number of elements of the sequence that are currently in use. Valid indices for the sequence must lie in the range `[0, length-1]`. The length also indicates the number of elements that are marshalled when the sequence is passed a parameter of a remote invocation.
- `maximum` The number of elements allocated in the sequence memory buffer. This attribute is greater than or equal to the length attribute. Note that the number of allocated elements (the maximum) increases automatically as the length is increased.

As an example, consider how to use a simple sequence of `long`:

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

When this is mapped to C++, a corresponding C++ class called `LongSeq` is defined. A sample of C++ code using `LongSeq` might look like the following:

```
// C++
{
// 1. Allocate the sequence (i.e. declaration)
LongSeq x(10);   // x.maximum() is 10
                 // x.length() is 0
x.length(4)
// 2. Initialize the sequence, etc.
// Can only access 4 elements
CORBA::ULong i;
for (i=0; i<4; i++) x[i] = i;

// Increasing length reallocates memory if necessary
x.length(20);
for (i=4; i<20; i++) x[i] = i;

// 3. Delete the sequence
// Happens automatically when it goes out of scope
}
```

The constructor for `LongSeq` takes a single parameter that sets the maximum of the sequence, that is it makes an initial allocation of 10 elements. Notice that a sequence always starts out with its length set to zero, therefore the first thing you have to do to a sequence is explicitly set its length.

The sequence is accessed via an overloaded `[]` (indexing) operator. This gives the sequence the appearance of a one-dimensional array. Note the type of the index must be a `CORBA::ULong`.

In spite of its name, the attribute `maximum()` does not pose a real limit to the size of a sequence. It is quite permissible, as in this example, to increase the length beyond the maximum. The effect is that the maximum is automatically increased to accommodate the new length.

A smart pointer `LongSeq_var` is also declared for this sequence. The above example can be written using the `LongSeq_var` pointer, as follows:

```
// C++
{
// 1. Allocate the sequence
LongSeq_var xV = new LongSeq(10);
              // xV->maximum() is 10
              // xV->length() is 0
xV->length(4)
// 2. Initialize the sequence, etc.
// Can only access 4 elements
CORBA::ULong i;
for (i=0; i<4; i++) xV[i] = i;

// Increasing length reallocates memory if necessary
xV->length(20);
for (i=4; i<20; i++) xV[i] = i;

// 3. Delete the sequence
// Happens automatically when it goes out of scope
}
```

On the whole, the smart pointer `LongSeq_var` uses pointer syntax. An exception to the expected syntax is the indexing `xV[i]` where you might have expected `(*xV)[i]`. In fact, either of these is correct. The first version is supported as well for convenience.

Deep Copy and Recursive Delete

Copying of sequences obeys deep copy semantics. That is, if a sequence contains a compound type that contains other types, up to an arbitrary degree of nesting, then a simple copy of the sequence recursively copies all of the nested data contained in the sequence. Consider the simple example of a sequence of strings:

```
    // IDL
    typedef sequence<string> StringSeq;
```

The type `StringSeq` can be initialized and copied as follows:

```
// C++
// 1. Allocate and initialize the first sequence.
StringSeq_var originalSeq = new StringSeq(2);
                 // originalSeq->maximum() is 2
                 // originalSeq->length() is 0
originalSeq->length(2);
originalSeq[(CORBA::ULong)0] =
           CORBA::string_dup("First string");
originalSeq[(CORBA::ULong)1] =
           CORBA::string_dup("Second string");
```

```
// 2. Allocate the second sequence.
StringSeq_var copyOfSeq = new StringSeq(2);
                 // originalSeq->maximum() is 2
                 // originalSeq->length() is 0

// 3. Deep copy from originalSeq into copyOfSeq
*copyOfSeq = *originalSeq;
```

When the deep copy is carried out, by assigning `*originalSeq` to `*copyOfSeq`, new copies of the two strings are made and the corresponding `char*` pointers for the new strings are stored in `(*copyOfSeq)[0]` and `(*copyOfSeq)[1]` respectively.

In keeping with the general rule for compound CORBA types, sequences also have recursive delete semantics. Given a sequence with nested levels of data, it is only necessary to call `delete` on the top level sequence in order to effect a recursive delete of all data associated with the sequence. In the above code extract, deletion is carried out automatically by the smart pointer type `StringSeq_var`[3].

Memory Management

The rules for the memory management of CORBA data types are important. Mismanagement of memory leads either to dumping core (where memory is unexpectedly deleted) or leaking of memory (if data is not deleted when it should be). This is the price paid for explicit control over memory allocation in C++. In Java, by contrast, memory management is much easier; it is taken care of by the garbage collector.

There are three significant steps in the life-cycle of a CORBA data type:

- *Allocation*—either on the stack or on the heap (using `new`).
- *Initialization*
- *Deallocation*—either automatic, or using `delete`.

3. An exception to this rule is the case where you have allocated the data yourself and used some special options in the constructor.

The CORBA developer has to understand where each of these steps is carried out, whether in the *calling* code (client) or in the *called* code (server). The rules are different for each of the different parameter passing modes. Thus there are different memory management rules for each of the following five cases:

- `in`
- `inout`
- `out`—fixed length type
- `out`—variable length type
- return value

The memory management rules for each of these five cases can be summarized by the following table:

	`in`	`inout`	`out` (fixed-len)	`out` (variable-len)	return value
Client	`A, I, D`	`A, I, D`	`A, D`	`D`	`D`
Server	`C`	`C, DO`	`I, C`	`A, I, C`	`A, I, C`

`A` = Allocate

`I` = Initialize

`D` = Deallocate

`C` = Copy param explicitly, if needed on server

`DO` = Deallocate old param, if changed by server

Table 3.1: *Summary of Memory Management Rules for CORBA Data Types.*

For the StockWatch demonstration it is enough to understand the rules for return values and the `in` mode. The sample client and server code given at the end of this chapter can be used as an example of the correct memory management of return values. For a full exposition of memory management issues, consult the *Orbix C++ Programmer's Guide*.

Memory Management and the OCGT

If you do not have time to become expert in CORBA memory management, there is significant help available in the form of the Orbix Code Generation Toolkit (OCGT). Considerable care was taken during the development of the OCGT to ensure that code would be generated strictly in accordance with the rules of memory management. One of the intended uses of the OCGT, therefore, is to generate sample code that serves as a model of correct memory management.

For example, given an IDL file `Foo.idl`, the following command:

```
idlgen cpp_genie.tcl -makefile -client -server \
        -interface -complete Foo.idl
```

generates a sample application that shows how parameters passed to CORBA invocations are correctly allocated, initialized and deallocated. By default, the generated code uses the smart pointer (`_var`) types wherever possible. If you want to see how the same example would look using dumb pointers instead, you can use the `-novar` option to the `cpp_genie`. For example:

```
idlgen cpp_genie.tcl -makefile -client -server \
        -interface -complete -novar Foo.idl
```

When using dumb pointers, memory deallocation is an explicit operation. It is therefore easier to see where deallocation occurs.

Exception Handling

Exception handling is a very important capability of the ORB. Distributed applications can often be affected by network failures or configuration problems. It is therefore necessary to program these applications in a way that responds to errors, enabling them to deal with network errors or other transient failures in a flexible way.

Another reason for the importance of exception handling is to assist in the component oriented approach to application development. In a CORBA environment, server components and clients components are typically developed by different groups of people. In this sort of environment, errors due to incorrect usage or implementation limits of a particular component should be accompanied by an error message that is as informative as possible. These kinds of error message can be helpful to client component developers during the testing phase.

All CORBA exceptions are arranged in a hierarchy. There are two broad categories of exceptions: CORBA system exceptions, which inherit from the base class `CORBA::SystemException`; and CORBA user exceptions, which inherit from the base class `CORBA::UserException`. Both of these exception classes, in turn, are derived from the base class `CORBA::Exception`. This exception hierarchy is illustrated in Figure 3.3.

$$\begin{tikzcd}
& \texttt{CORBA::Exception} & \\
\texttt{CORBA::SystemException} \arrow[ur] & & \texttt{CORBA::UserException} \arrow[ul]
\end{tikzcd}$$

Figure 3.3: *Hierarchy of Exception Classes in CORBA.*

The first category of exception is the *CORBA system exception*. Exceptions of this type are raised exclusively by the ORB and they represent errors that originate in the ORB runtime or stub/skeleton code. Errors in this category include `COMM_FAILURE`, indicating network problems, and `INV_OBJ_REF`, indicating that a particular object was not found on the server. Exceptions of this type can be raised on an IDL operation without being explicitly declared. Most of the exceptions in this category are standardized by the OMG and can be used interoperably between ORBs.

The second category of exception is the *CORBA user exception*. Exceptions of this type are intended to be used by application developers to signal application-level exceptions. Unlike CORBA system exceptions, it is not possible to raise arbitrarily any user exception on any operation. It is necessary, first of all, to declare the user exception in IDL and to explicitly associate the user exception with an operation by appending a `raises()` clause.

Consider, for example, the following extract from the StockWatch IDL file:

```
        // IDL
        // File: StockWatch.idl

        ... // Some definitions omitted.

        // Exceptions
1       exception rejected {
              string m_reason;
        };

        // Interfaces

        interface Stock {
              Symbol getSymbol();
2             string getDescription() raises (rejected);
              Money getCurrentPrice() raises (rejected);
              PriceInfoSeq getRecentPrices() raises (rejected);
              void recordSale(in Money price) raises (rejected);
        };
```

This IDL can be explained as follows:

1. Before an exception can be used it must be declared. The syntax for declaring an exception is similar to that of a struct, with the keyword `exception` being substituted for the keyword `struct`. The purpose of declaring fields in a user exception is to make the exception as informative as possible. However, it is also permissible to declare an exception that has no fields, for example:

   ```
   // IDL
   exception rejected {};
   ```

   would also be a valid declaration for an exception.

2. It is not enough just to define the exception in IDL. Before you can use it, it is also necessary to add a `raises()` clause to each operation that might raise the exception, as shown here. The `raises()` clause supports a comma separated list of exceptions, if multiple exceptions are required. This declaration mechanism bears a similarity to the C++ `throw()` declaration. However, it differs in an important respect: the absence of a `raises()` clause in an operation declaration implies that *no* user exceptions may be raised by that operation (system exceptions are always permitted).

When programming with user exceptions there are two aspects to their use. One aspect is on the server side, where the user exception can be constructed and thrown. The other aspect is on the client side, where the user exception can be caught and handled.

On the server side, consider this example of the exception `rejected` being thrown:

```
// C++
...
// Some lines of code omitted...
void
Stock_i::recordSale(Money price, CORBA::Environment &)
        throw (CORBA::SystemException, rejected)
{
        // Throw a CORBA User Exception ...
        throw rejected("recordSale not implemented");
}
```

The user exception `rejected` maps to a class of the same name in C++. The constructor for `rejected` takes a single string argument that is used to initialize the field `m_reason`. If an exception has multiple fields, the C++ constructor features multiple arguments to initialize the corresponding fields.

On the client side, the exception `rejected` can be explicitly caught as follows:

```
// C++
... // Some lines of code omitted.

try {
        obj1->recordSale(latestPrice);
}
catch (rejected& ex) {
        cerr << "UserException: rejected: reason = "
           << ex.m_reason << endl;
}
```

The exception `rejected` is caught by reference, which is the recommended way of catching CORBA exceptions. The fields of the exception are accessible from the caught instance `ex`, as in `ex.m_reason`.

It is also possible to catch the whole category of user exceptions with a single catch clause. This works because user exceptions, such as `rejected`, inherit from the base class `CORBA::UserException`. For example:

```
// C++
... // Some lines of code omitted.

try {
        obj1->recordSale(latestPrice);
}
catch (CORBA::UserException& ex) {
        cerr << ex << endl;
}
```

When this generic user exception is printed to `cerr`, it gives the type of the user exception that was caught.

Some examples of how to throw and catch exceptions also appear in the sample code in the following sections.

Client Code

Sample code that makes use of the new features of the `Stock` interface is shown below. The code is intended to illustrate the concepts introduced in the preceding section and is not intended to be a realistic example of a client program.

A complete sample client program can be obtained by pasting the code below into the `// Add Code Here` section of an OCGT generated client:

```
// C++
// 1. Invoke 'recordSale()' with a recent price
Money latestPrice = 18.0;

try {
        obj1->recordSale(latestPrice);
}
catch (rejected& ex) {
        cerr << "UserException: rejected: reason = "
           << ex.m_reason << endl;
}
catch (CORBA::SystemException& ex) {
        cerr << ex << endl;
}
```

```
// 2. Invoke 'getRecentPrices()' and print results
PriceInfoSeq_var priceHistoryV;

try {
        priceHistoryV = obj1->getRecentPrices();
}
catch (rejected& ex) {
        cerr << "UserException: rejected: reason = "
           << ex.m_reason << endl;
}
catch (CORBA::SystemException& ex) {
        cerr << ex << endl;
}

cout << "PriceInfoSeq = {" << endl;
for ( CORBA::ULong i=0; i < priceHistoryV->length(); i++) {
        cout << "\t" << "[" << i << "] = {" << endl;
        cout << "\t\t" << "m_price = "
           << priceHistoryV[i].m_price << endl;
        cout << "\t\t" << "m_when = "
           << priceHistoryV[i].m_when << endl;
        cout << "\t}" << endl;
}
cout << "}" << endl;

// Deallocation of 'priceHistoryV' takes place
// automatically at the end of the current scope.
```

First of all, the client invokes `recordSale()` to add a new price to the history of sales. Then it invokes `getRecentPrices()` and prints out the complete price history it received.

Note that it is necessary to declare the index of the sequence to be of type `CORBA::ULong`. The C++ compiler relies on the type of the index to find the correct definition of the overloaded `[]` (indexing) operator.

Server Code

The OCGT can be used, as in the previous chapter, to generate the starting point for the server. The server code can then be completed by filling in the missing method bodies of the `Stock_i` class in file `Stock_i.cxx`.

The code below shows a sample implementation of the two new operations `recordSale()` and `getRecentPrices()`. It is not intended to provide a realistic implementation of these operations. The purpose of this code sample is to illustrate how to use the features introduced in this chapter, that is the manipulation of structs and sequences, and the use of exception handling.

```
// C++
//-----------------------------------------------------------------
// Function:getRecentPrices()
//
// Description: Implementation of the IDL operation
//    Stock::getRecentPrices().
//-----------------------------------------------------------------

PriceInfoSeq*
Stock_i::getRecentPrices(
      CORBA::Environment &)
         throw(CORBA::SystemException,
               rejected)
{
         // Dummy Implementation:
         // Always returns the same data...

         // Allocation
         PriceInfoSeq* theInfoSeqP = new PriceInfoSeq(2);
         theInfoSeqP->length(2);

         // Initialization
         (*theInfoSeqP)[(CORBA::ULong)0].m_when =
            CORBA::string_dup("27 April 1999");
         (*theInfoSeqP)[(CORBA::ULong)0].m_price =
            (Money) 18.0;

         (*theInfoSeqP)[(CORBA::ULong)1].m_when =
            CORBA::string_dup("28 April 1999");
         (*theInfoSeqP)[(CORBA::ULong)1].m_price =
            (Money) 18.5;

         return theInfoSeqP;
}
```

```
//-----------------------------------------------------------------
// Function:recordSale()
//
// Description: Implementation of the IDL operation
//    Stock::recordSale().
//-----------------------------------------------------------------

void
Stock_i::recordSale(
      Money price,
      CORBA::Environment &)
         throw(CORBA::SystemException,
               rejected)
{
         // Dummy implementation:
         // Echo the parameters to standard out...
         cout << "Attempted to record sale." << endl;
         cout << "Price = " << price << endl;

         // Throw a CORBA User Exception now...
         throw rejected("recordSale not implemented");
}
```

Notice how the memory management rules are applied to the return value of the operation `getRecentPrices()`. The return value `theInfoSeqP` is declared to be a dumb pointer `InfoSeq*`. This is deliberate, because if `theInfoSeqP` were declared to be a smart pointer `InfoSeq_var` the data it pointed at would be deleted as soon as the method returned. Declaring it to be a dumb pointer is the only approach that makes sense.

Deallocation of the return value `theInfoSeqP` is the responsibility of the calling code (usually the ORB runtime). It is deallocated after the method returns.

4

Integration with a Database

The concept of a three-tier application is discussed and applied to the StockWatch example.

Versions of the client application described in this chapter are located in the `/demos/common/cxx` directory of your Orbix installation. The server code is located in the `/demos/common/src/servers/example`*`x`* directory of your Orbix installation (where *x*=1-5).

Designing a Three-Tier Architecture

When building applications that use or generate persistent data, a key architectural decision is how many tiers to include in the architecture.

Historically many applications were structured in two tiers, where application and database servers were integrated, and the client application would typically have explicit knowledge about the detail and schema of the database. This would commonly extend to having embedded SQL calls in the client. Although this approach has certain advantages, there are a number of problems with this tight coupling of database clients and servers. For example:

- Any change to the database schema requires that all of the clients which access that database also potentially need to change. This can lead to technical and logistical problems, especially where large numbers of clients are concerned.
- Clients can become bloated if they need to incorporate large database libraries in order to operate.

- Distribution of clients over a wide area can be difficult to manage.
- Large numbers of clients can create a bottleneck at the database server. This can be addressed by replicating the database, or by introducing a transaction processing monitor to multiplex connections. Both of these approaches can lead to added complexity.

These and other problems have led to the widespread adoption of a three-tier approach to building distributed applications. Three-tiered architectures facilitate communication between application clients and application servers, which in turn communicate with databases. Orbix allows you to develop three-tier systems.

Benefits of a Three-Tier Architecture

CORBA, with its use of IDL to define application interfaces, is well suited for use in a three-tier environment. By additionally employing OTS technology, applications can readily co-ordinate two-phase commit transactions across distributed components. Well-designed three-tier CORBA applications can also make use of CORBA Naming and Trader Services in order to provide improved scalability and location transparency for object services on the network.

The main benefits of the three-tier approach include the following:

- There is a much higher level of abstraction between the client and the database. An intervening layer of application logic means that particular database design aspects (including the type and version of the database itself) can be hidden from the client, and changed.
- Once the application server, which is a client to the database, exposes a well-defined and unchanging interface to its clients, modifying or swapping out the database is easy.
- Additionally, when the load on a given application server becomes too high, it is straightforward to add more servers to balance the load.

Factory Objects

This section introduces factory objects, a useful concept in the design of three-tier applications with CORBA. There is nothing fundamentally new introduced here, from the point of view of CORBA, but factory objects are very commonly used in the design of CORBA interfaces. You can think of it as a *factory pattern*, a simple design pattern.

The motivation for factory objects can be got by taking a closer look at the existing design of the StockWatch system. In the current system there is only one type of CORBA object, the `Stock` object. As the number of stocks increases, a separate `Stock` object has to be instantiated in the server for each stock symbol. Ultimately, in a fully fledged system, thousands of `Stock` objects may be needed leading to the situation illustrated in Figure 4.1.

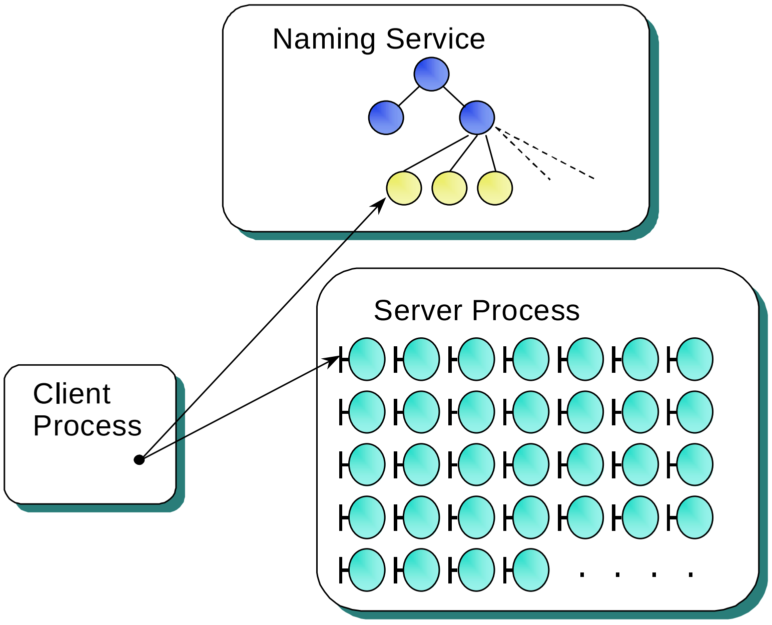


Figure 4.1: *No Factory Object—All Objects Published in Naming Service*

This approach does not scale particularly well. Some of the problems with this are as follows:

- With a very large number of stocks, there might not be enough memory to instantiate all of the `Stock` objects.
- There may be unacceptable latency while the server starts up because it takes a long time to instantiate all of the objects.
- The access that the client has to these objects is inflexible. Clients must know the identifier for each `Stock` object and access them directly. There is no facility, for example, to search for particular kinds of stock.

To avoid instantiating all of the `Stock` objects at once, it would be better to have some mechanism for creating `Stock` objects dynamically, as they are needed. In other words, what is needed is something like the distributed equivalent to a C++ constructor.

There is no such thing as a constructor in IDL. However you can get the effect that you need by defining a *factory interface*. A factory interface is any interface that can be used to obtain references to other objects.

In the case of `Stock` objects the factory interface `StockWatch` can be introduced both to manage the `Stock` objects and to dynamically instantiate them as they are needed. The `StockWatch` interface is defined as follows:

```
// IDL
// File – StockWatch.idl

. . .

typedef sequence<Symbol> SymbolSeq;

interface StockWatch {

  SymbolSeq getSymbols ()
    raises (rejected);

  Stock getStockBySymbol (in Symbol aSymbol)
    raises (rejected);

};
```

The first operation `getSymbols()` returns a list of symbols recorded in the database, each of which corresponds to a `Stock` object.

The second operation `getStockBySymbol()` is the operation analogous to a constructor. Given a valid symbol, it dynamically instantiates the corresponding `Stock` object and returns a reference to it.

It is easy to spot a factory interface: one of its operations has a return type that is the name of another interface. In this example, the operation `getStockBySymbol()` has the return type `Stock`.

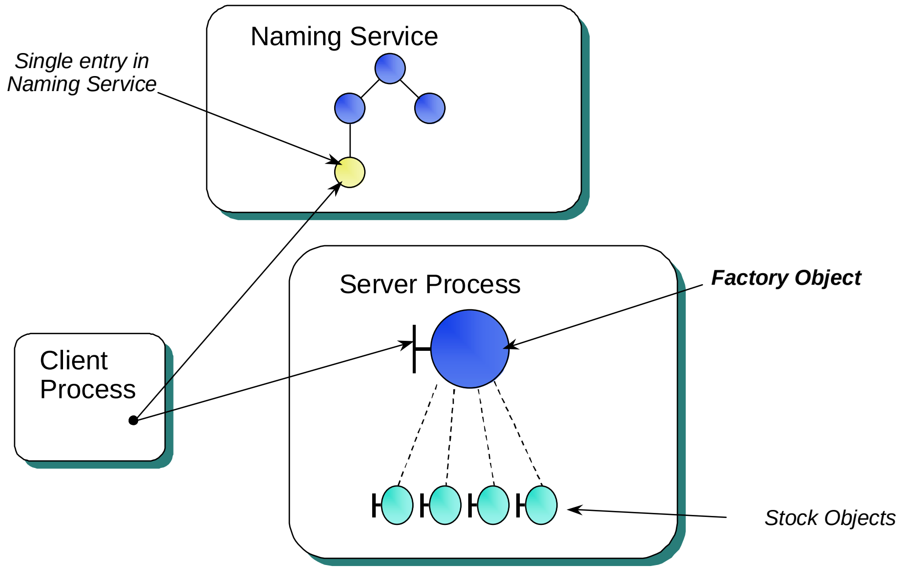


Figure 4.2: *Factory Object—One Object Published in the Naming Service*

With the introduction of the `StockWatch` factory interface, the system can by described by Figure 4.2. In this system there is one `StockWatch_i` object (implementing the `StockWatch` interface) and a small number of `Stock` objects.

A sample implementation of the `StockWatch` interface is provided below in the Server Implementation section.

StockWatch Three-Tier Application

To illustrate Orbix we work through a simple StockWatch example. This example provides stock price information for a selected number of stocks. It consists of a server process that accesses stock prices from persistent storage, in this case an Oracle database. Figure 4.3 illustrates the components in the application. These consist of:

- A C++ StockWatch client, and a Java StockWatch client.

  These clients invoke IDL operations in the server via the CORBA IIOP protocol.
- A C++ StockWatch server.

  The server processes client requests, and can access and communicate implicitly with persistent storage via SQL statements embedded in 'C'.
- Persistent storage.

  Storage, for StockWatch, could be in the form of one of the following databases: Oracle or SQLServer.

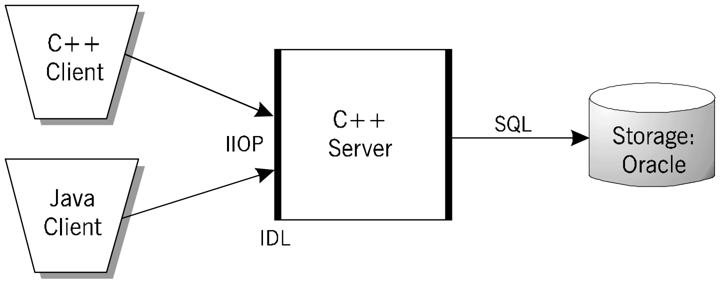


Figure 4.3: *Simple Three-Tiered Architecture*

The three tier architecture features two important interfaces, indicated in Figure 4.3 by a thick black line at each side of the C++ server.

These interfaces can be described as follows:

- *1st–2nd tier interface*—The IDL interface. This represents the interface between remote clients and the CORBA server.
- *2nd–3rd tier interface*—The backend interface. This represents the interface between the C++ server and the database. The StockWatch example defines this interface in terms of an abstract C++ class called `backend`.

The Database Schema

Two database tables are used in the StockWatch example. The `SYMBOLS` table contains the name, symbol description and current stock price(s) of a company.

Table `SYMBOLS`	
Field Type & Names	**Field Descriptions**
string `NAME`	Name of symbol.
string `SYM_DESC`	Description of symbol.
number `VALUE`	Current stock price(s) of company.

Table 4.1: *The SYMBOLS Table*

Example SQL command to create the `SYMBOLS` table:

```
CREATE TABLE SYMBOLS ( NAME VARCHAR(20) PRIMARY KEY,
              SYM_DESC VARCHAR(50) NOT NULL,
              VALUE REAL );
```

The PRICES table contains the history of stock prices for a given company and maintains this history on a date basis.

Table PRICES	
Field Type & Names	**Field Descriptions**
string NAME	Name of stock.
number VALUE	Current value of stock.
number PRICE_DATE	Stock price history.

Table 4.2: *The PRICES Table*

Example SQL command to create the PRICES table:

```
CREATE TABLE PRICES ( NAME VARCHAR(20) NOT NULL,
              VALUE REAL,
              PRICE_DATE VARCHAR(30) NOT NULL );
```

The IDL Interfaces

This IDL plays the role of interface between the first and second tier of the three-tier application. The IDL used for example 1 of the StockWatch example is as follows:

```
// IDL
// CORBA IDL for the StockWatch example

#include "EventService.idl"

// Basic type definitions
typedef string Date;
typedef float  Money;
typedef string Symbol;
typedef sequence<Symbol> SymbolSeq;

// Exceptions
exception rejected {
  string m_reason;
};
```

```
// Structs
struct PriceInfo {
  Money m_price;
  Date  m_when;
};
typedef sequence<PriceInfo> PriceInfoSeq;

// Interfaces
interface Stock {
  Symbol getSymbol();
  string getDescription ()
    raises (rejected);
  Money getCurrentPrice ()
    raises (rejected);
  PriceInfoSeq getRecentPrices ()
    raises (rejected);
  void recordSale(in Money price)
    raises (rejected);
  CosEventChannelAdmin::ConsumerAdmin getFeed();
};

interface StockWatch {
  SymbolSeq getSymbols ()
    raises (rejected);
  Stock getStockBySymbol (in Symbol aSymbol)
    raises (rejected);
};
```

This version of the StockWatch IDL features a new operation `Stock::getFeed()` which is used in conjunction with the event service. It is not needed for example 1 and is discussed later in connection with the event service.

The Backend Interface

The backend interface is defined as an abstract C++ class that hides database details and exposes methods useful to the StockWatch server. This abstract class plays the role of interface between the second and third tier of the three-tier application. It is defined as follows:

```
class backend {

public:
   ~backend() {}
   virtual char*  get_all_symbols(
                    char**& symbols,
                    int& numSymbols)=0;

   virtual char*  get_symbol_description(
                    const char* sym,
                    char*& desc)=0;

   virtual char*  get_symbol_prices(const char* sym,
                   float*& prices,
                   char**& times,
                   int& num_returned)=0;

   virtual char*  get_current_price(const char*, float&)=0;

   virtual char*  put_current_price(const char*, const float)=0;
};
```

The `get_all_symbols()` returns a list of all stock symbols in the database, `get_symbol_description()` returns a description of a specified stock symbol, and `get_symbol_prices()` returns the price of a selected stock symbol.

Class backend is defined as an abstract C++ class. You can easily plug in different underlying database systems by sub-classing backend.

Server Implementation

The structure of the server code is outlined in Figure 4.4.

The top layer of Figure 4.4 is the StockWatch IDL interface that stands between the first and second tiers of the application.

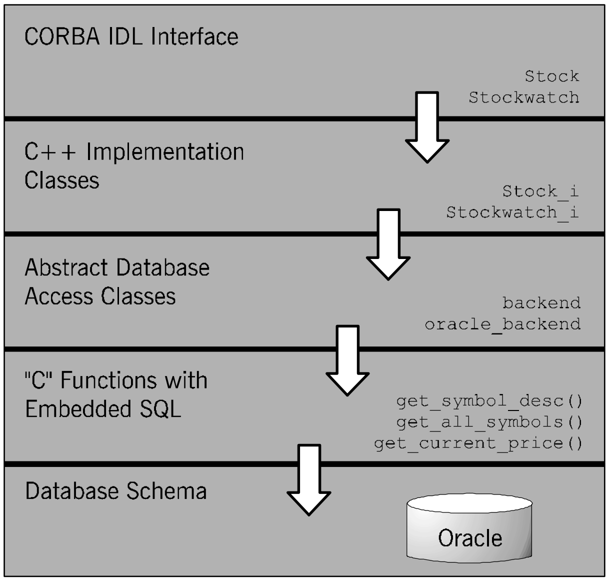


Figure 4.4: *Structure of StockWatch Server Program*

The middle layer of Figure 4.4 is the backend interface, defined as an abstract C++ class, that stands between the second and third tier of the application. Specific sub-classes of backend, such as oracle_backend, are implemented to allow different databases to plug into the StockWatch demonstration. The implementation details of these backend classes are not considered here since that is purely an exercise in database programming.

Implementation of Stock_i

Sample implementations for the Stock interface have already been given in the previous chapters. To integrate with the database, Stock_i must be modified to retrieve information from the database, via the backend class, and use this data for return values instead. The necessary modifications are straightforward and not presented here.

Implementation of StockWatch_i

The implementation of the StockWatch interface is instructive as it illustrates typical features of the factory design pattern, discussed earlier in this chapter. It also affords an opportunity to illustrate how the backend interface is used to interact with the database. The code for the constructor of the StockWatch_i class is as follows:

```
// C++
#define EXCEPTIONS
#include "StockWatch_i.h"
#include <iostream.h>
#include <stdio.h>

StockWatch_i::StockWatch_i(
              backend* b,
              unsigned char feed)
{
      m_backend = b;
      m_cache = new StockObjCache;

      if (feed){ ... // Ignore code for using event service.
      }
}
```

The variables m_backend and m_cache are two private members of the StockWatch_i class. The m_backend is used to hold a reference to the backend object and it provides the interface for all subsequent interaction with the database.

The m_cache holds a reference to a hash table. It is used to store references to Stock objects that are currently active in memory. The stock symbol is used as the key and the stock object reference as the value in the hash table.

The first of the IDL operations to be implemented is the `getStockBySymbol()` method:

```
      Stock_ptr StockWatch_i::getStockBySymbol(
                     const char* sym,
                     CORBA::Environment &IT_env)
      {
1           Stock_i* stk = m_cache->find(sym);
            if (stk){
              cout << "Found one in the cache" << endl;
              return Stock::_duplicate(stk);
            }

            //
            // A non-zero return value indicates a database error.
            // A null out value for desc indicates that the symbol
            // does not exit.
            //

            char* desc;
            char* status =
2              m_backend->get_symbol_description(sym,desc);
            if (status){
               throw rejected(status);
            }
            if (!strlen(desc)){
               throw rejected("Bad Symbol");
            }
            delete desc;

            stk = new Stock_i(sym,m_backend,m_eventChanMgr);
            m_cache->insert(sym,stk);
            return Stock::_duplicate(stk);
      }
```

This code can be explained as follows:

1. The implementation checks to see whether the stock object has already been loaded into memory from the database by searching the cache. If an object reference is found in the cache, the reference to the existing `Stock_i` object is duplicated and used as the return value.

The call to `Stock::_duplicate()` is needed because of the rules of memory management. To prevent the ORB from deallocating the returned object reference, a call is made to `Stock::_duplicate()`.

2. If the implementation fails to find the stock symbol in its cache, it proceeds to check whether a record exists for this particular symbol in the database. If the method `get_symbol_description()` fails it can be deduced that the symbol does not exist, in which case a CORBA user exception of type `rejected` is raised to alert the client.

   If a record of the symbol *does* exist in the database, the implementation proceeds to instantiate a `Stock_i` object, records this object reference in its cache and returns it, taking care to duplicate it first.

The other operation of the StockWatch interface is `getSymbols()` which returns a list of all the stock symbols recorded in the database. It is implemented as follows:

```
SymbolSeq* StockWatch_i::getSymbols(
              CORBA::Environment &IT_env)
{
      char** symbolArray;
      int numSymbols = 0;

      char* status = m_backend->get_all_symbols(
                    symbolArray,
                    numSymbols);
      if (status){
         throw rejected(status);
      }
      SymbolSeq* sseq = new SymbolSeq();
      sseq->length(numSymbols);

      for (int i = 0; i < numSymbols;i++){
         (*sseq)[(CORBA::ULong) i] =
              CORBA::string_dup(symbolArray[i]);
         delete symbolArray[i];
      }

      if (numSymbols) {
         delete symbolArray;
      }
      return sseq;
}
```

The implementation is a straightforward wrapper for the function backend::get_all_symbols(). First get_all_symbols() is invoked, returning symbolArray. The array of strings symbolArray is then converted to the CORBA type SymbolSeq.

Note how the rules for memory management of CORBA types are applied in this case. A dumb pointer type SymbolSeq* is used in the declaration of sseq rather than a _var type—it is not the responsibility of the implementation code to ensure that the return value is deallocated. The deallocation of the return value takes place *after* the getSymbols() method has returned and is carried out by the calling code.

5

The Naming Service

The standard CORBA Naming Service is introduced and programming examples are given. A non-standard wrapper class, with convenient features for accessing the naming service, is also presented.

Many large scale applications use a CORBA-compliant naming service such as OrbixNames as a centralized object repository. OrbixNames is IONA Technologies' implementation of the CORBA Naming Service specification. The role of OrbixNames is to allow a name to be associated with a CORBA object, and to allow that object to be found subsequently by resolving that name. Instead of requiring that a client bind to a specific server on some host, the server *binds* a name-to-object mapping in the names repository. The client *resolves* that object reference by looking up the name. The name still needs to be available to the client, but it need not be concerned with the object's server location.

OrbixNames Concepts

OrbixNames maintains a database of *bindings* between names and object references. A binding is an association between a name and an object. The naming service provides operations to resolve a name, to create new bindings, to delete existing bindings, and to list the bound names.

A name is always resolved within a given *naming context*. The naming context objects in the system are organized into a naming graph, which may form a naming hierarchy, much like that of a filing system.

Figure 5.1 illustrates the components of the application:

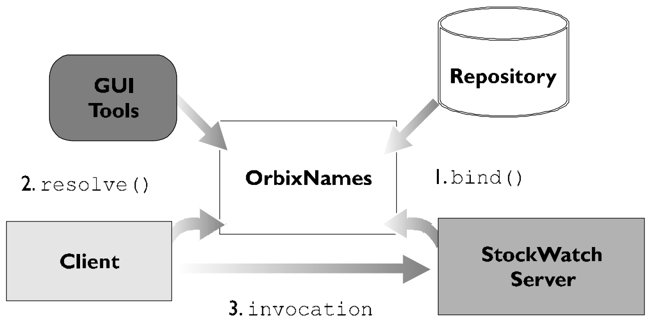


Figure 5.1: *Using OrbixNames in Orbix*

1. A StockWatch server publishes StockWatch object references in OrbixNames. OrbixNames maps names to these StockWatch object references, thereby maintaining a repository of name associations in the form of a database.
2. Clients resolve names in OrbixNames.
3. Clients remotely invoke operations on object references in the StockWatch server.

Naming Service IDL

Figure 5.2 shows a tree-like hierarchy in a naming service. The black discs represent naming contexts and the open discs represent object bindings. At the root of the hierarchy is the *root naming context*. This is generally the point of entry to the naming service: all other naming contexts and name bindings are accessible from the root. In Figure 5.2 are shown two other naming contexts `example1` and `example2`. The naming context `example1` shows two bindings `oracle` and `sqlsrv`.

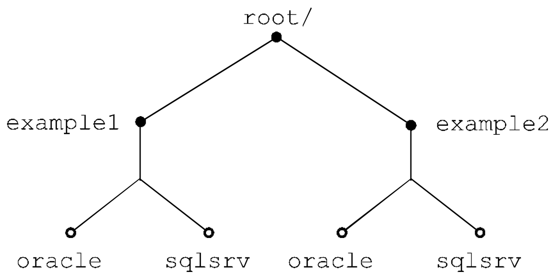


Figure 5.2: *StockWatch Naming Context Hierarchy*

A convenient string format is used to refer to entries in the naming service—a format used by the OrbixNames command-line utilities. Using this format[1], the bindings under the naming context `example1` can be denoted as `"example1.oracle"` and `"example1.sqlsrv"`. The `'.'` (dot) character is used to separate the components of a name.

Name Format

The format of a name is defined by the naming service IDL (see *OrbixNames Programmer's and Administrator's Guide* for a complete listing of this IDL) in the module `CosNaming`[2]. Names are defined in IDL as follows:

1. This is *not* a standard string format. The OMG has yet to ratify a standard string format for names. However, according to the current draft of the Interoperable Naming Service, the name component separator character is likely to be standardized as '/' (forward slash).

```
#pragma prefix "omg.org"

module CosNaming {

      typedef string Istring;

      struct NameComponent {
         Istring id;
         Istring kind;
      };
      typedef sequence<NameComponent> Name;
      ...
      // Further definitions not shown...
};
```

An individual name component is defined as a struct containing two fields, an `id` field and a `kind` field. The `id` field is intended to function as a simple identifier while the `kind` field is intended to describe the purpose of the named entity. Note, however, that the `kind` field currently does not enjoy any special status. In most cases it is left blank (that is, equal to an empty string).

The `id` field and `kind` field taken together are used to specify a name component uniquely. Therefore, if the `kind` field is not blank it must also be supplied as part of the name component in order to avoid ambiguity.

A complete `Name` is defined to be a sequence of name components. For example, the name `"example1.oracle"` translates into a sequence of two name components:

Sequence index	`id` field	`kind` field
`0`	`"example1"`	`""`
`1`	`"oracle"`	`""`

The `kind` field is implicitly assumed to be blank.

2. The `Cos` in `CosNaming` stands for *common object services*.

In the string format used by the OrbixNames utilities, a '-' (hyphen) is used to separate the `id` field from the `kind` field. For example, a name such as `"example1-TheKindField.oracle"` translates into:

Sequence index	`id` field	`kind` field
`0`	`"example1"`	`"TheKindField"`
`1`	`"oracle"`	`""`

There are, in general, no restrictions on the strings specified in the `id` or `kind` fields except that OrbixNames will not accept an empty string for the `id` field.

Binding

The most common operation carried out by servers is that of publishing an object binding in the naming service. The entry thus created can be resolved by a client, at a later stage, in order to locate the associated object.

The operation used to create an object binding is called `rebind()` and it is defined as part of the interface `NamingContext`. Most of the interesting functionality of the naming service is defined in the interface `NamingContext`. The relevant extract of IDL is as follows:

```
#pragma prefix "omg.org"

module CosNaming {
      interface NamingContext {
         ...
         void rebind(in Name n, in Object obj)
            raises(NotFound, CannotProceed, InvalidName);
         ...
      };
      ...
      // Further definitions not shown...
};
```

The operation `rebind()` takes two arguments: a `Name`, as defined in the previous section, and `Object`, representing an object reference.

The type Object has a special significance in IDL. It represents the base interface for all interfaces—in other words, all IDL interfaces implicitly inherit from type Object. Therefore, a parameter of type Object allows you to pass object references of arbitrary type.

When rebind() is invoked it creates an object binding—an association between the specified name and object reference, stored persistently in the *Names Repository*. Since an object reference gives the location of a CORBA object, another way of expressing this is to say that the object's location is stored under a particular name.

There are, in fact, two operations available for creating object bindings: these are rebind() and bind(). The operation rebind() has more convenient semantics because it allows you to overwrite existing entries in the naming service.

Resolving

The most common operation carried out by clients is that of *resolving* a name to obtain an object reference. In doing so, clients are effectively executing a find operation because an object reference contains location details for the remote object.

The operation used to resolve a name is called resolve() and is defined as part of the NamingContext interface. The relevant extract of IDL is as follows:

```
#pragma prefix "omg.org"

module CosNaming {
      interface NamingContext {
         ...
         Object resolve(in Name n)
            raises(NotFound,
               CannotProceed,
               InvalidName);
         ...
      };
      ...
      // Further definitions not shown...
};
```

The client supplies the Name of the object it wishes to locate and receives a return value Object, which is the corresponding object reference.

The special type `Object` is also used here because it offers the flexibility to return object references of any type.

How Servers Bind Objects in OrbixNames

The following code sample demonstrates how a server names an object. It shows how the `main()` function of a StockWatch server could be changed to publish a single StockWatch object to the naming service.

```
     // serverOracle.cc
           ...
           oracle_backend orac(argv[1], argv[2]);

           try {
                 // Set my Orbix server name
                 // It is recommended that this should be done BEFORE
                 // objects are instantiated, and put into OrbixNames
1                CORBA::Orbix.setServerName(serverName);
                 // Instantiate my instance of StockWatch
                 StockWatch_i swatch(&orac);
                 // Bind to OrbixNames
                 CosNaming::NamingContext_var rootCtx;
2                rootCtx =
                    CosNaming::NamingContext::_bind("root:NS",host);

                 // Construct a Name.
3                tmpName = new CosNaming::Name(2);
                 tmpName->length(2);
                 tmpName[0].id = CORBA::string_dup("example1");
                 tmpName[0].kind = CORBA::string_dup("");
                 tmpName[1].id = CORBA::string_dup("oracle");
                 tmpName[1].kind = CORBA::string_dup("");

                 // Now bind() my instance of StockWatch in OrbixNames
4                rootCtx->rebind(tmpName,&swatch);
                    // Now wait for incoming invocations.
5                CORBA::Orbix.impl_is_ready(serverName,TIMEOUT);
           } catch ... // Series of catch blocks hidden
```

The code is explained as follows:

1. Initialize the Orbix specific server name to `example1/oracle`. Orbix requires that applications which publish objects in OrbixNames call either `CORBA::BOA::setServerName()` or `CORBA::BOA::impl_is_ready()` before interaction with OrbixNames. Object references cannot be correctly created until the server name has been set because Orbix needs to embed the server name into the object reference.
2. Obtain a proxy to the `root` naming context using the Orbix `_bind()` feature. An alternative approach is to use IONA's implementation of the CORBA initialization service.
3. Construct an instance of `CosNaming::Name`. The name constructed can be written in the OrbixNames utilities string format as `"example1.oracle"`.
4. Create a name-to-object binding in OrbixNames. The use of `rebind()` handles the case where the name-to-object mapping already exists in OrbixNames.
5. A call to `impl_is_ready()` signals that the server is ready to receive invocations from clients and starts a CORBA event loop for the processing of these events.

How a Client Finds a Named Object

Clients find objects by resolving names and obtaining object references in the naming service. The following code sample illustrates how a C++ client finds a named object:

```
      // clientns.cc

      int main(int argc, char **argv) {
         StockWatch_var swatch;
         CosNaming::Name_var tmpName;

         char* server = argv[1];
         char* demo = argv[2];
         char* host = argv[3];

         try {
               // Bind to OrbixNames
               cout << " Binding to OrbixNames " << endl;
               CosNaming::NamingContext_var rootCtx;
1              rootCtx=CosNaming::NamingContext::_bind("root:NS",host);

               // Construct a Name
2              tmpName = new CosNaming::Name(2);
               tmpName->length(2);
               tmpName[0].id = CORBA::string_dup("example1");
               tmpName[0].kind = CORBA::string_dup("");
               tmpName[1].id = CORBA::string_dup(server);
               tmpName[1].kind = CORBA::string_dup("");

               // Now resolve() my instance of StockWatch in OrbixNames
3              CORBA::Object_var tmpObj = rootCtx->resolve(tmpName);

4              swatch = StockWatch::_narrow(tmpObj);
               if (CORBA::is_nil(swatch)) {
                  cerr << "Error: StockWatch narrow failed" << endl;
                  exit(1);
               }
            menuLoop(swatch);
            } catch ... // Series of catch blocks hidden
      }
```

The code is explained as follows:

1. Obtain a proxy to the `root` naming context.
2. Construct an instance of `CosNaming::Name`. The name constructed can be written in the OrbixNames utilities string format as `"example1.oracle"`.
3. Resolve the name in OrbixNames. If the binding exists, then an instance of `CORBA::Object` is returned, otherwise the exception `CosNaming::NamingContext::NotFound` is thrown.
4. Narrow the instance of `CORBA::Object` to a `StockWatch` proxy. The type `CORBA::Object` is the C++ representation of the IDL type `Object`. Since `CORBA::Object` is a generic base class it is not possible to invoke any useful methods on the returned object reference until it has been *narrowed* to the correct type.

   In this example, a `StockWatch` object is expected so the static method `StockWatch::_narrow()` is used to cast `tmpObj` to the correct type.

   If the narrowing should fail for any reason, the `_narrow()` method will return a nil object reference. It will not throw a CORBA exception. To check for an error condition arising from a failed narrowing you can use the function `CORBA::is_nil()` to test if the narrowed reference (in this example `swatch`) is nil or not.

Names Wrapper Demo

One drawback to the `NamingContext` interface is that it does not support the specification of names in a simple string format. Instead the user is forced to specify a name in the relatively ungainly form of a `CosNaming::Name`, a sequence of structs.

To simplify the interface to the naming service, Orbix supplies a sample implementation of a parser which can convert a string format name to a `CosNaming::Name`. The parser is implemented by the class `IT_Demo_NSWParser` and can be found in the directory *IONARoot*`/demos/demolib` of your Orbix installation.

The public interface to the parser class is as follows:

```
      // C++
      // File: IT_Demo_NSWParser.h

      class IT_Demo_NSWParser {
      public:
         static IT_Demo_NSWParser *
1        create(
            char componentSeparator,
            char idKindSeparator,
            char escape
         ) throw ();

         virtual CosNaming::Name *
2        stringToName(
            const char *prefix,
            const char *name
         ) throw ();

         void
3        defineNameSeparators(
            char componentSeparator,
            char idKindSeparator,
            char escape
         ) throw ();

      };
```

The three public methods of `IT_Demo_NSWParser` can be explained as follows:

1. The `create()` method is used instead of a constructor to create an instance of a parser object. For example, to create a parser that accepts strings in the same format as that used by the OrbixNames utilities you would initialize it as follows:

   ```
   // C++

   IT_Demo_NSWParser * theParserP;
   theParserP = create('.','-','\\');
   ```

2. The method `stringToName()` carries out the conversion of a string to a `CosNaming::Name`. The `prefix` is simply prefixed to the name argument before the conversion takes place. That is, given the parser initialized as above, calling `stringToName("example1","oracle")` would be equivalent to calling `stringToName("","example1.oracle")`.
3. The method `defineNameSeparators()` allows the string format accepted by the parser instance to be altered at any time.

There is another class `IT_Demo_NSW` provided in the directory *IONARoot*/`demos/demolib`, which provides a wrapper around the naming service. This wrapper class provides a slightly enhanced interface to the naming service and can be used instead of making invocations directly on the `NamingContext` interface. However, it must be borne in mind that this is *not* a standard interface and the code is supplied only as a demo.

The class `IT_Demo_NSW` has the following public methods:

```
   class IT_Demo_NSW {
   public:
1     IT_Demo_NSW () throw();
      virtual ~IT_Demo_NSW() throw();

2     virtual void setNamePrefix (const char *contextName) throw ();

3     virtual void clearNamePrefix () throw ();

4     virtual void registerObject (
         const char *objectName,
         CORBA::Object_ptr object
      ) throw (CORBA::Exception);

      virtual CORBA::Object_ptr
5     resolveName(const char *name) throw (CORBA::Exception);

      virtual void
6     removeObject (const char *objectName) throw (CORBA::Exception);

      enum BehaviourOption {
         ignoreNotFoundError = 0x01,
         createMissingContexts = 0x02,
         overwriteExistingObject = 0x04,
         deleteEmptyContexts = 0x08
      };
7          void setBehaviourOption(BehaviourOption option) throw ();
           ...
       };
```

The methods of class `IT_Demo_NSW` can be explained as follows:

1. The constructor for `IT_Demo_NSW` initializes the wrapper in a state where it accepts names given in the string format of the OrbixNames utilities. In fact, each wrapper instance is implicitly associated with an instance of the parser `IT_Demo_NSWParser` which it uses to convert strings to names.
2. The method `setNamePrefix()` specifies a name prefix which is prefixed to object names appearing in all subsequent invocations of `registerObject()`, `resolveName()` and `removeObject()`.
3. The method `clearNamePrefix()` clears the name prefix set by `setNamePrefix()`.
4. The method `registerObject()` replaces the functionality of `rebind()` or `bind()`, causing a name binding to be published to the naming service. It takes its name argument in a string format, prepending whatever name prefix is currently set. The semantics of this method are affected by the options set by `setBehaviourOption()`.
5. The method `resolveObject()` replaces the functionality of `resolve()`, looking up a name in the naming service and returning an object reference. It allows the name argument to be specified in a string format, prepending the current value of the name prefix.
6. The method `removeObject()` replaces the functionality of `unbind()` and is used to delete a name binding from the naming service. The current name prefix is prepended to the `objectName` argument to specify the name binding. The semantics of this method are affected by the options set by `setBehaviourOption()`.
7. There are four behaviour options, of type `BehaviourOption`, which can be set via the method `setBehaviourOptions()`. These options have the following effects:

   `ignoreNotFoundError`—causes the `removeObject()` method to ignore any errors of the type `CosNaming::NamingContext::NotFound`.

   `createMissingContexts`— `registerObject()` will create any naming contexts needed for the compound name.

overwriteExistingObject—cause registerObject() to overwrite any existing entry in the naming service (same semantics as rebind()).

deleteEmptyContexts—removeObject() will delete any contexts left empty as a result of removing an object from the naming service or as a result of the automatic removal of a context.

Server Code Using Names Wrapper

The following code extract shows the implementation of the main() function for the StockWatch server that connects to an Oracle database backend:

```
     // C++
     // File: serverOracle.cxx

     ...
     #include <it_demo_nsw.h>
     ...

     #define TIMEOUT 1*1000*60 // 1 mins.

     int main(int argc, char **argv) {
1       CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"Orbix");
        CORBA::BOA_var boa = orb->BOA_init(argc, argv, "Orbix_BOA");

        char serverName[64];
        char *username   = argv[1];
        char *passwd     = argv[2];

        sprintf(serverName,"%s/%s","example1","oracle");

        orb->setServerName(serverName);
        //Indicate server should not quit while clients are connected
        boa->setNoHangup(1);

        // For correct generation of IORs, one must call impl_is_ready
        // before any objects are created.
2       boa->impl_is_ready(serverName, 0 );

        // Create an implementation object
        oracle_backend orac(username, passwd);
```

```
          try{
3            IT_Demo_NSW ns_wrapper;
             ns_wrapper.setNamePrefix("example1");

4            StockWatch_var swatch =  new StockWatch_i(&orac);

             const char *object_name = "oracle";

5            ns_wrapper.setBehaviourOption(
                         IT_Demo_NSW::createMissingContexts);
             ns_wrapper.setBehaviourOption(
                         IT_Demo_NSW::overwriteExistingObject);

             // register the object in the naming service
6            ns_wrapper.registerObject(object_name,swatch);

7            boa->impl_is_ready(serverName, TIMEOUT);
          }
          catch (CORBA::Exception &ex) {
             cerr << ex << endl;
             CORBA::release(orb);
             exit(1);
          }
          cout << "server exiting" << endl;

          return 0;
       }
```

The code can be explained as follows:

1. The functions `CORBA::ORB_init()` and `CORBA::ORB::BOA_init()` are used to obtain, respectively, references to a `CORBA::ORB` object and a `CORBA::BOA` object. These two functions are part of the CORBA initialization service used for bootstrapping an ORB. The two references `orb` and `boa` obtained in this way are meant to be used in place of the global static object `CORBA::Orbix`. The methods accessible from the `CORBA::ORB` object are useful for both clients and servers. The methods exposed by the `CORBA::BOA` class (which inherits from `CORBA::ORB`) are intended to be used by server programs. The use of the object `CORBA::Orbix` is now deprecated.

2. The method `impl_is_ready()` is invoked with a zero timeout before any name bindings are published to the naming service. It must be invoked before any new object references are created. Note how it is invoked on the `boa` instance rather than using the `CORBA::Orbix` global object.
3. An instance of a names wrapper `IT_Demo_NSW` is created and the prefix is set to be `"example1"`.
4. An instance of interface `StockWatch` is created by instantiating a `StockWatch_i` object.
5. Two behaviour options are set for the names wrapper: `createMissingContexts` and `overwriteExistingObject`.
6. The method `registerObject()` publishes the object reference `swatch` to the naming service. Assuming that `object_name` is equal to `"oracle"`, the name used in the name binding is `"example1.oracle"` (since `"example1"` is the current prefix). On account of the behaviour options set in the preceding step, the naming context `"example1"` will be created if it does not already exist, and if the name binding `"example1.oracle"` already exists it will be overwritten.
7. When `impl_is_ready()` is called with a non-zero timeout the server enters a state where it begins processing client connection requests and invocations.

Client Code Using Names Wrapper

The following code extract shows the implementation of the `main()` function for the client `clientns.cxx` which connects to the StockWatch server with the help of the naming service:

```
// C++
// File: clientns.cxx

...
#include <it_demo_nsw.h>
...

int main(int argc, char **argv) {
   char* object_name = argv[1];
   char* demo = argv[2];
1          CORBA::ORB_var orb = CORBA::ORB_init(argc,argv,"Orbix");

           try {
2             IT_Demo_NSW ns_wrapper;
              ns_wrapper.setNamePrefix(demo);

3             CORBA::Object_var obj = ns_wrapper.resolveName(object_name);
4             StockWatch_var swatch = StockWatch::_narrow(obj);

5             if ( CORBA::is_nil(swatch) ) {
                cout << "StockWatch::_narrow() failed" << endl;
                CORBA::release(orb);
                exit(1);
              }

              menuLoop(swatch);
           }
           catch (CORBA::Exception &ex) {
              cerr << ex << endl;
              CORBA::release(orb);
              exit(1);
           }

           return 0;
        }
```

The code can be explained as follows:

1. The function `CORBA::ORB_init()` is used to obtain a reference, called `orb` above, to a `CORBA::ORB` object that is used on the client side. This function is part of the CORBA initialization service and replaces use of the now deprecated `CORBA::Orbix` object.
2. An instance of a names wrapper `IT_Demo_NSW` is created and the prefix is set to the value of `demo` equal to the string `"example1"` in this case.
3. The wrapper object is used to resolve a name given in string format. Assuming that `object_name` has been specified as `"oracle"` on the command line, and the prefix `"example1"` is in force, the name that is resolved is `"example1.oracle"`.

4. Narrow the instance of `CORBA::Object` to a `StockWatch` proxy. The type `CORBA::Object` is the C++ representation of the IDL type `Object`. Since `CORBA::Object` is a generic base class it is not possible to invoke any useful methods on the returned object reference until it has been *narrowed* to the correct type.

   In this example, a `StockWatch` object is expected so the static method `StockWatch::_narrow()` is used to cast `obj` to the correct type.
5. If the narrowing should fail for any reason, the `_narrow()` method will return a nil object reference. It will not throw a CORBA exception. To check for an error condition arising from a failed narrow you must use `CORBA::is_nil()` to test whether the narrowed reference is nil or not.

Configuring OrbixNames

IONA's implementation of the naming service, OrbixNames, is written in Java making it possible to run this service on a wide variety of platforms. However, users do not have to worry much about Java configuration issues because the OrbixNames server is packaged in two different forms:

1. OrbixNames is provided as a stand-alone executable `ns`. In this form, the Java runtime is combined with the executable and `ns` can be run in the same way as any binary executable.
2. OrbixNames is also provided as a set of classes `OrbixNames.jar` and a Java runtime is provided in the *IONARoot*`/tools/jre` directory. This is the form of naming service that is normally used in conjunction with OrbixWeb.

In the following sections it is assumed that the various IONA products have been installed in:

`/opt/iona`	UNIX default IONA root directory.
`C:\Iona`	Windows default IONA root directory.

This IONA root directory will be referred to as *IONARoot* in the following subsections.

Configuration Variables for OrbixNames

The configuration variables for OrbixNames are usually found in the file *IONARoot*/`config/orbixnames3.cfg`. The following environment variables are the ones that most commonly need to be edited for OrbixNames, and they are found in the `OrbixNames` configuration scope:

`IT_NAMES_SERVER`	The default server name for the naming service is `NS`. The server name specified here must be the same as that given in the `putit` command when registering the naming service.
`IT_NAMES_SERVER_HOST`	The name of the host where the naming service will be run.
`IT_NAMES_REPOSITORY_PATH`	The *names repository* is a file based repository for storing the persistent state of the naming service. This variable specifies the directory where the names repository is stored.

Two other configuration variables that are important for the naming service are `Common.IT_DEFAULT_CLASSPATH` and `Common.IT_JAVA_INTERPRETER`, from the file *IONARoot*/`config/common.cfg`. It should not be necessary to change these from the default install values. However, if the value of either of these variables is accidentally corrupted it can prevent the naming service from running.

Registering the Naming Service

Before the naming service can be used it must be registered with the Orbix daemon. The following steps assume that the naming service will be run on a host *NSHost*, which is just the value of `OrbixNames.IT_NAMES_SERVER_HOST`. The naming service can be registered as follows:

1. Ensure that an Orbix daemon is running on *NSHost*. If necessary, start an Orbix daemon by typing the following at a command prompt:

   ```
   orbixd
   ```

2. Register the naming service with the following command:

   Windows

   ```
   putit -h NSHost NS IONARoot\bin\ns
   ```

   UNIX

   ```
   putit -h NSHost NS IONARoot/bin/ns
   ```

   The option `-h` *NSHost* can be omitted if this command is executed on *NSHost*. Note that it is essential that the pathname given as the last argument is an absolute pathname.

   If you want to use the load balancing features of OrbixNames, you should register it using the following command instead:

   Windows

   ```
   putit -h NSHost NS "IONARoot\bin\ns -l"
   ```

   UNIX

   ```
   putit -h NSHost NS "IONARoot/bin/ns -l"
   ```

   The `-l` switch tells the naming service to enable load balancing. For further details, consult “Load Balancing” on page 187

3. To test that the naming service has been correctly registered, enter:

   ```
   catit NS
   ```

   to view the registration details.

4. Try running one of the naming service utilities:

   ```
   lsns -k
   ```

   This command lists the contents of the root context of the naming service. If this command is run against a newly configured naming service, you should see output like the following:

   ```
   [Contents of root]
      lost+found- (Context)
      ObjectGroups- (Context)
   [0 Objects, 3 Contexts]
   ```

   If this runs successfully it means the naming service is correctly configured.

Naming Service Utilities

A number of command-line utilities are supplied with the naming service.

To list the contents of a particular naming context, use the following command:

```
lsns -k NamingContext
```

To create a new naming context in the naming hierarchy use the following command:

```
putnewncns NewNamingContext
```

To print the object reference (IOR) associated with a particular name binding, use the following command:

```
catns NameOfBinding
```

All of these commands take their name arguments in the string format discussed in the section "Name Format" on page 97.

6

The Event Service

The standard CORBA Event Service is introduced. Programming examples are presented that illustrate the untyped push model of event propagation.

Introduction

OrbixEvents implements the CORBA Event Service specification. This specification defines a model for communications between ORB applications that supplements the direct operation call system that client/server applications normally use.

The CORBA Event Service introduces the concept of *events* to CORBA communications. An event originates at an event *supplier* and is transferred to any number of event *consumers*. Suppliers and consumers are completely decoupled: a supplier has no knowledge of the number of consumers or their identities, and consumers have no knowledge of which supplier generated a given event.

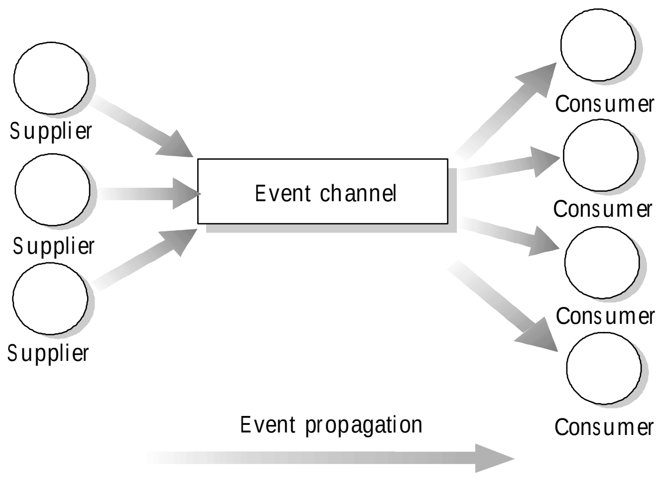


Figure 6.1: *An Event Channel Mediates the Flow of Events from Suppliers to Consumers.*

In order to support this model, the CORBA Event Service introduces to CORBA a new architectural element, called an *event channel*. An event channel mediates the transfer of events between suppliers and consumers as follows:

1. The event channel allows consumers to register interest in events, and stores this registration information.
2. The channel accepts incoming events from suppliers.
3. The channel forwards supplier-generated events to registered consumers.

Suppliers and consumers connect to the event channel and not directly to each other (preceding Figure 6.1). From a supplier's perspective, the event channel appears as a single consumer; from a consumer's perspective, the event channel appears as a single supplier. In this way, the event channel decouples suppliers and consumers

Types of Event Communication

There are two approaches to initiating event propagation—the *push* model and the *pull* model.

Two different kinds of event channel are provided, one of which allows you to send events via a generic interface (*untyped event channel*) and another which allows you to send events via a user defined interface (*typed event channel*). Only the untyped event channel is covered in this chapter. For details on how to program with typed event channels you should consult the *OrbixEvents Programmer's Guide*.

The Push Model

In this model the supplier initiates event propagation by sending an event to the event channel. Immediately following this, the event channel attempts to propagate this event to all of the registered consumers. In other words, the supplier decides when to send an event and propagation of the event proceeds without delay.

This model is often used as part of a *callback* design pattern. A server program, acting as supplier, can notify a large number of clients, acting as consumers, whenever an event of particular interest occurs. The StockWatch example makes use of this pattern to distribute updates on changing prices to `eventconsumer` clients. The StockWatch server (supplier) signals changes in stock prices by sending events to the `eventconsumer` clients (consumers).

The advantage of the push model is that it reverses the usual direction of event flow between client and server. Normally it is the client that sends events to the server, while the server waits passively. By contrast, the push model supports a pattern where the server takes the initiative in transmitting events to the client.

The Pull Model

In this model the consumer initiates event propagation by requesting an event from the event channel. In response, the event channel will go back to all of the registered suppliers and request an event from them. If one of the suppliers yields an event, this event can be propagated to the consumer. In other words, the consumer decides when it wants an event but propagation may be delayed if none of the suppliers are ready to supply an event.

The pull model is not discussed any further in this chapter. Consult the *OrbixEvents Programmer's Guide* for details.

Untyped Event Communication

Untyped event communication is so named because it does not allow the user to choose the *type* of the interface used to propagate events. Instead, the user is forced to use the generic interface for either the push or the pull model of events. For example, if electing to use the push model of untyped communication, you must use the following interface:

```
//IDL

// In scope of module 'CosEventComm'
interface PushConsumer {
      void push(in any data) raises (Disconnected);
      void disconnect_push_consumer();
};
```

to propagate events. Event propagation proceeds by invoking the operation `push()` on a `PushConsumer` object. An event is represented by the argument `data` which is of type any. Use of the IDL type any to represent an event gives a reasonable degree of flexibility to the interface because it allows any valid IDL type, including user defined IDL types, to be sent as arguments to the operation `push()`.

Typed Event Communication

For some applications, the restrictiveness of untyped event communication may be unsuitable. In these cases, the interfaces used for event propagation might already have been defined or it might be a requirement of the design to have well defined user interfaces for event propagation.

With typed event communication, the user defines an interface, subject to some restrictions, and event propagation in the typed push model proceeds by invoking the operations of this user defined interface (the typed pull model, however, is slightly more complicated). Data types passed are no longer restricted to be of type any, but can be declared to be any valid IDL type.

For details on how to use the typed model of event propagation, consult the *OrbixEvents Programmer's Guide*.

Callback Objects

In a simple CORBA application, all of the CORBA objects might reside in the server. A straightforward distinction then exists between server processes and client processes. Servers provide a home for CORBA objects and service incoming invocations from clients. Clients access these objects via remote invocation, but do not themselves contain CORBA objects. However, in some cases it can be useful to implement a CORBA object in a client process: these objects are known as *callback objects*.

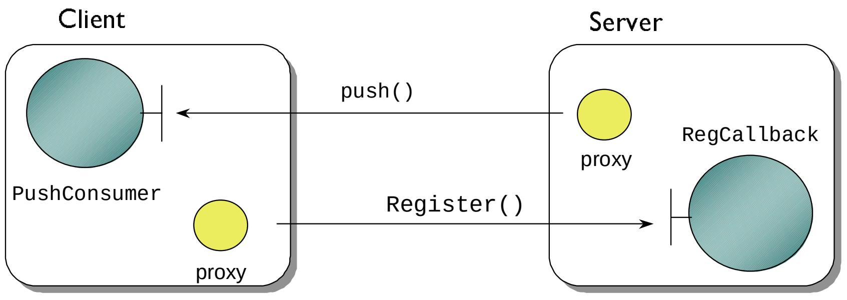


Figure 6.2: *Impure Client (left) Containing Callback Object.*

Client programs that do not implement any CORBA objects are referred to here as *pure clients*. Clients that implement callback objects are referred to here as *impure clients* as illustrated in Figure 6.2.

To see why it might be useful to implement a callback object, consider the example of a financial trader using a GUI client to interact with a broker server. For much of the time the broker server is in a passive mode, responding to requests from the trader's GUI client. However, for certain types of interaction the server needs to take an active role. For example, the rise and fall of stock prices could be monitored by sending updates to a screen showing price information in the GUI client. In order for this to work, there has to be a CORBA object implemented in the GUI client. The broker server can then call back on this CORBA object whenever it has new price information for the client.

The main features in the design of a callback can be illustrated with the help of the following sample IDL:

```
// IDL

exception Disconnected { };

// 1. Interface for the 'Callback object'
// Implemented on the client side.

interface PushConsumer {
   void push (in any data) raises (Disconnected);
   void disconnect_push_consumer ();
};

// 2. Interface for the 'Registration object'
// Implemented on the server side.

interface RegCallback {
   void Register(in PushConsumer PCObj);
   void Deregister(in PushConsumer PCObj);
};
```

Two interfaces are needed as part of the callback design: the interface for the callback object, `PushConsumer`, and the interface for registering the callback, `RegCallback`.

The interface `PushConsumer` exposes two operations. The operation `push()` is the operation that a server invokes when it wants to call back on the client. The argument to `push()` is declared to be any, a special type that allows any CORBA data type to be passed at runtime. Another operation, `disconnect_push_consumer()`, is declared which can be used by the server to tell the client that it will not be sending any more messages.

The interface `RegCallback` is implemented by the server. This interface is needed so that the client can pass the server a reference to the callback object. The operation `Register()` passes a reference to the client's `PushConsumer` object to the server. The operation `Deregister()` indicates that the client does not wish to receive any more messages via the callback object. The role of this interface is important because it is technically difficult to obtain references to callback objects. It is not possible to locate a callback object using `_bind()` because a client is not registered with the daemon.

Sample code for the implementation of `PushConsumer` and `RegCallback` interfaces is not given here. However, the concepts underlying callback objects recur in the discussion of the event service, for which sample code is given later in this chapter.

The Mainline of an Impure Client

The use of callback objects in a client has some programming implications for client developers. It changes the way in which a client `main()` function must be written.

In order to appreciate the changes that are made to the client `main()` function, it is helpful to look at the distinction that exists in CORBA between clients and servers.

A list of characteristics can be given for a CORBA server:

- Instances of CORBA objects are created.
- Object references are, optionally, published in the naming service.
- The server *must* call `CORBA::BOA::impl_is_ready()`.
- The server is registered with the Orbix daemon via a call to `putit`.

A client program that contains no CORBA objects is a pure client.

A pure client program has the following features:

- References to remote CORBA objects are obtained.
- Invocations are made on remote CORBA objects.
- The client *never* calls `CORBA::BOA::impl_is_ready()`.
- The client executable is *not* registered with the Orbix daemon via a call to `putit`.

It can be seen from this that a pure client is chiefly distinguished by the things it does not do.

A client program that implements one or more callback objects is an *impure client*. An impure client program has the following additional features:

- A callback object is instantiated and registered with the server.
- The impure client *must* call `CORBA::BOA::processEvents()` (or an equivalent Orbix event handling function).

Because the impure client implements a CORBA object it must start an Orbix event loop to process invocations on the callback object. This is a step that servers usually accomplish by calling `impl_is_ready()`. Clients, on the other hand, are prohibited from calling `impl_is_ready()` so they must call `processEvents()` instead.

Calling the `processEvents()` function can potentially cause a difficulty in the client code. When `processEvents()` is called it blocks and usually it remains blocked for the entire lifetime of the client. If the client is single threaded, it cannot perform other tasks.

In most cases, the best solution is to make the client multithreaded and to call `processEvents()` from a subthread, allowing the main thread to carry on with other client activities. For a detailed discussion of event handling options in Orbix, and for approaches that work with single threaded clients, consult the *Orbix C++ Programmer's Guide*.

Locating an Event Channel

The standard event service is under-specified in one respect; there is no single starting point that represents the whole event service and can be used to gain access to all event channels. Instead, the starting point for the event service is given by an event channel object. Since there are many event channel objects (as illustrated in Figure 6.3) the first task facing the programmer is how to locate the appropriate one.

The mechanism for locating event channels is implementation dependent and Orbix supports two different approaches: bootstrap via the Orbix specific `_bind()` call, or bootstrap via the Orbix specific `ChannelManager` interface. Each of these mechanisms is presented in turn.

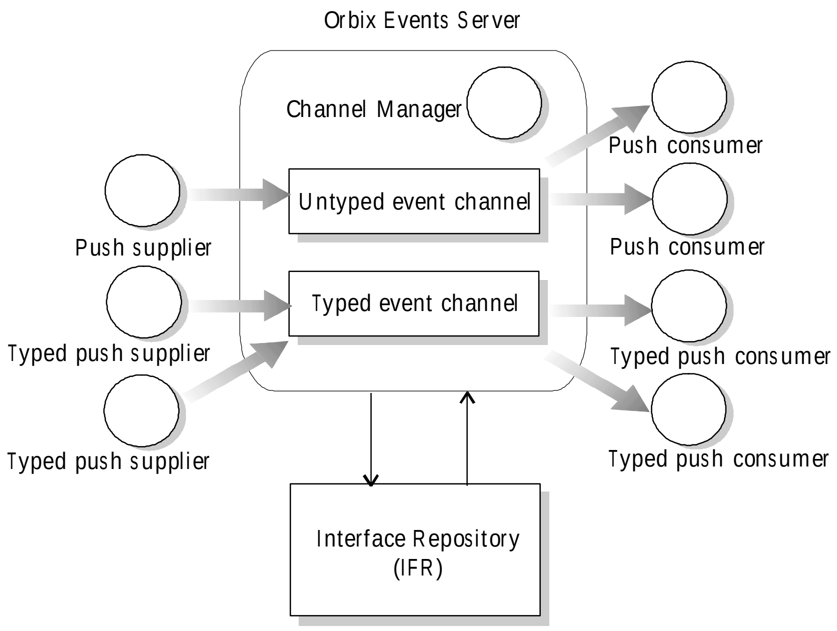


Figure 6.3: *Architecture of the OrbixEvents Server.*

Locating Channels via _bind()

When using the Orbix specific `_bind()` mechanism, the location of the event channel object is specified by the triplet of *marker*, *serverName* and *hostName*. Recall that an invocation of `_bind()` takes the following form:

```
interfaceName::_bind(
      "marker:serverName",
      "hostName"
      )
```

In this example, the *interfaceName* is `CosEventChannelAdmin::EventChannel`.

- The *marker* is equal to a channel identifier. A channel identifier is associated with every event channel when created by OrbixEvents. This identifier can be specified as one of the command-line options to the `es` event service executable.
- The *serverName* is the name of the event server as specified via the `putit` utility when it is being registered, by default `"ES"`.
- The *hostName* is the name of the host where the event server is running.

The following code extract illustrates this approach:

```
// C++

#include <EventService.hh>
...
CosEventChannelAdmin::EventChannel_var channelV;

try {
   channelV =
      CosEventChannelAdmin::EventChannel::_bind(
            "Channel_1:ES",
            "myHost.dublin.iona.ie"
            );
}
catch (CORBA::Exception& ex) {
   // Handle the exception...
}
// The reference 'channelV' is now initialized
// and ready for use.
```

Note that the channel identifier used in this example is "Channel_1". The server name ES is the default name for the Orbix event service.

Locating Channels via the ChannelManager Interface

Although the CORBA standard does not specify any interface for managing EventChannel objects, a ChannelManager interface is supplied with OrbixEvents for convenience. This ChannelManager is, necessarily, non-standard and takes care of such things as creating new event channels and finding existing ones (in other words, a kind of factory object).

The ChannelManager interface provides another option for locating event channels. An extract from the IDL file for the ChannelManager interface is given by:

```
// IDL
module OrbixEventsAdmin {

  exception duplicateChannel{ };
  exception noSuchChannel{ };

  interface ChannelManager
  {
    typedef sequence<string> stringSeq;

    CosEventChannelAdmin::EventChannel create
    (
      in string  channel_id
    ) raises (duplicateChannel);

    CosEventChannelAdmin::EventChannel find
    (
      in string channel_id
    ) raises (noSuchChannel);

    string findByRef
    (
      in CosEventChannelAdmin::EventChannel channel_ref
    )
    raises (noSuchChannel);

    stringSeq list();
    CosTypedEventChannelAdmin::TypedEventChannel createTyped
    (
      in string  channel_id
    ) raises (duplicateChannel);

    CosTypedEventChannelAdmin::TypedEventChannel findTyped
    (
      in string channel_id
    ) raises (noSuchChannel);

    string findByTypedRef
    (
      in CosTypedEventChannelAdmin::TypedEventChannel channel_ref
    )
    raises (noSuchChannel);

    stringSeq listTyped();
};
};
```

You can use `_bind()` to connect to the channel manager and then invoke `ChannelManager::find()` to locate an event channel. The programming steps needed to connect to a particular event channel via this method are given by the following code extract:

```
// C++

OrbixEventsAdmin::ChannelManager_var m_eventChanMgr;
CosEventChannelAdmin::EventChannel_var m_chan;

try{
   m_eventChanMgr =
      OrbixEventsAdmin::ChannelManager::_bind(":ES", eventsHost);
}
catch(CORBA::Exception& ex) {
   cerr << "Exception binding to OrbixEvents" << endl;
   exit(1);
}

try {
   m_chan = m_eventChanMgr->create("IONA");
}
```

```
catch(OrbixEventsAdmin::duplicateChannel) {
   cout << "Use already existing Channel " << sym << endl;
   try {
      m_chan = m_eventChanMgr->find("IONA");
   }
   catch(CORBA::Exception& ex) {
      cerr << "Error finding channel" << endl << ex << endl;
      exit(1);
   }
}
catch(CORBA::Exception& ex) {
   cerr << "Exception creating channel" << endl << ex << endl;
   exit(1);
}
```

As an alternative to using `_bind()` to connect to the `ChannelManager` object, you have the option of looking up the `ChannelManager` in the naming service.

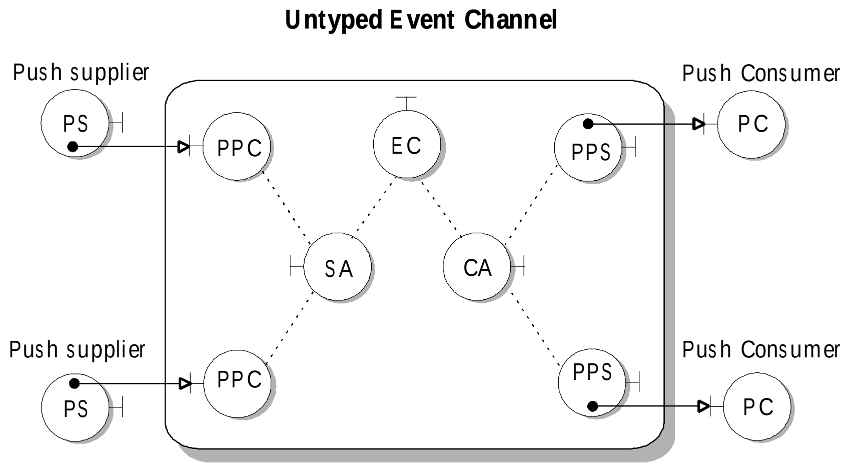
By default, the event service publishes the `ChannelManager` object reference in the naming service, under the following name:

`id`	`"OrbixEventsAdmin ChannelManager"`
`kind`	`""`

The full IDL for the channel manager can be found in Appendix B. For further details consult the *OrbixEvents Programmer's Guide*.

Attaching a Supplier

Once a supplier has got hold of a reference to an event channel, the next step is to attach itself to the event channel so that it can begin supplying events.



Key

EC	`EventChannel`	PPC	`ProxyPushConsumer`
SA	`SupplierAdmin`	PPS	`ProxyPushSupplier`
CA	`ConsumerAdmin`		

Figure 6.4: *Architecture of an Event Channel*

In this context, attaching the supplier means getting hold of a reference to a ProxyPushConsumer (PPC) object. The PPC object acts as a sink for all of the events emanating from the supplier.

The process of attaching a supplier is broken up into a number of steps, in the standard event service. This is illustrated by Figure 6.4. The supplier proceeds via a chain of factory objects before it finally gets hold of a reference to a PPC object. It starts with a reference to an EventChannel object, then gets a reference to the corresponding SupplierAdmin object and then a reference to a PPC object.

The IDL for the factory objects illustrated in Figure 6.4 is as follows:

```
// IDL
module CosEventChannelAdmin {

   exception AlreadyConnected {
   };

   exception TypeError {
   };

   ...

   interface ConsumerAdmin {
      ProxyPushSupplier obtain_push_supplier ();
      ProxyPullSupplier obtain_pull_supplier ();
   };

   interface SupplierAdmin {
      ProxyPushConsumer obtain_push_consumer ();
      ProxyPullConsumer obtain_pull_consumer ();
   };

   interface EventChannel {
      ConsumerAdmin for_consumers ();
      SupplierAdmin for_suppliers ();
      void destroy ();
   };
};
```

Getting a Reference to a ProxyPushConsumer

Sample code for obtaining a reference to a PPC object is shown below. It is assumed that the reference to an event channel `channelV` has already been obtained using one of the methods outlined in the preceding section "Locating an Event Channel" on page 124.

```
// C++

CosEventChannelAdmin::EventChannel_var channelV;
CosEventChannelAdmin::SupplierAdmin_var supAdminV;
CosEventChannelAdmin::ProxyPushConsumer_var ppcV;
...

// Assume that the reference 'channelV' has already
// been initialized

try {
   supAdminV = channelV->for_suppliers();
   ppcV = supAdminV->obtain_push_consumer();
}
catch (CORBA::Exception& ex) {
   // Handle exception...
}

// The reference 'ppcV' is now initialized and
// ready for use.
```

Figure 6.4 on page 130 illustrates the chain of objects invoked to obtain a reference to a `ProxyPushConsumer` object. An invocation is made on the EC (`EventChannel`), that returns a reference to the SA (`SupplierAdmin`). An invocation is then made on SA to return a reference to a PPC (`ProxyPushConsumer`) object.

The object SA is unique per channel and invoking `for_suppliers()` always returns a reference to the same object. By contrast, invoking `obtain_push_consumer()` always returns a reference to a new `ProxyPushConsumer` object.

Connecting to a ProxyPushConsumer

The final step in the attachment of a supplier to the channel is when the supplier invokes the appropriate connect operation on the PPC object. Two IDL interfaces are important at this stage: the `ProxyPushConsumer` interface and the `PushSupplier` interface. The relevant IDL from the event service is as follows:

```
// IDL
module CosEventComm {
   exception Disconnected {
   };

   interface PushConsumer {
      void push (in any data) raises (Disconnected);
      void disconnect_push_consumer ();
   };

   interface PushSupplier {
      void disconnect_push_supplier( );
   };
   ...
};

module CosEventChannelAdmin {
   exception AlreadyConnected { };
   exception TypeError { };

   interface ProxyPushConsumer : CosEventComm::PushConsumer {
      void connect_push_supplier (
         in CosEventComm::PushSupplier push_supplier)
         raises (AlreadyConnected);
   };

   interface ProxyPushSupplier : CosEventComm::PushSupplier {
      void connect_push_consumer (
            in CosEventComm::PushConsumer push_consumer)
         raises (AlreadyConnected, TypeError);
   };
};
```

The supplier finishes attaching itself to the channel by calling `connect_push_supplier()` on the PPC object.

Note however, that the connect_push_supplier() operation requires a reference to a PushSupplier object. Therefore, before the supplier can connect it must provide an implementation of a PushSupplier object.

The PushSupplier object does not play a major role. The function of the PushSupplier object is to facilitate an orderly shutdown of connections to the event server, should the event server go off-line for some reason. It supports just a single operation disconnect_push_supplier(). A sample implementation might look like this:

```
// C++

#include <EventService.hh>

class PushSupplier_i : public virtual
         CosEventComm::PushSupplierBOAImpl {
   // Private member variables
   int m_disconnected;

public:
   // C'tor and D'tor
   PushSupplier_i() : m_disconnected(0) { }

   int isOkToPush() throw() {
      return (m_disconnected) ? 0 : 1;
   }
   //
   // IDL Operations
   //
   void disconnect_push_supplier() {
      m_disconnected = 1;
   }
};
```

Once you have an implementation of a PushSupplier object in the supplier, you are ready to connect to the ProxyPushConsumer in the event channel. Just a few lines of code are needed to accomplish this step:

```
// C++
...
PushSupplier_i * psImpl = new PushSupplier_i();
// The reference to the ProxyPushConsumer 'ppcV'
// has already been initialized above
ppcV->connect_push_supplier(psImpl);
```

The remote ProxyPushConsumer object is now in its connected state and ready to receive events.

Supplying Events

Most of the work associated with using the event service has been done in the preceding sections. Supplying events is simply a matter of invoking the push() operation on the PPC object. The following code extract shows a string event being supplied to the event channel:

```
// C++

...
// The following variables have been initialized above:
//
//    psImpl- reference to a local PushSupplier_i
//            object
//
//    ppcV  - reference to a remote ProxyPushConsumer
//            object in the 'connected' state.

CORBA::Any a;
a <<= "StringToBePushed";

if (psImpl->isOkToPush() ) {
      ppcV->push(a);
}
```

The class CORBA::Any is the C++ representation of the IDL type any. The any can be used to hold any valid IDL type, including basic types and user defined types. To initialize an any, the relevant data is inserted using the <<= (left shift assignment) operator. In this case, a string is inserted into the any and the correct version of the <<= operator is selected by the C++ compiler via operator overloading. The any makes a copy of the inserted string and this copy is owned by the any.

In this example, the status of the connection is checked via isOkToPush() before invoking the push() operation.

Attaching a Consumer

The process of attaching a consumer to an event channel parallels the steps involved in attaching a supplier. After obtaining a reference to the event channel, the consumer attaches itself by getting hold of a reference to a `ProxyPushSupplier` (PPS) object.

The steps involved in attaching a consumer are shown in Figure 6.4 on page 130. The consumer proceeds via a chain of factory objects. It starts with a reference to an `EventChannel` object, then gets a reference to the corresponding `ConsumerAdmin` object and then a reference to a PPS object.

The IDL for the factory objects illustrated in Figure 6.4 on page 130 is given in the preceding section “Attaching a Supplier” on page 130.

Getting a Reference to a ProxyPushSupplier

Sample code for obtaining a reference to a PPS object is shown below. It is assumed that the reference to the event channel `channelV` has already been obtained using one of the methods outlined in the preceding section “Locating an Event Channel” on page 124.

```
// C++

CosEventChannelAdmin::EventChannel_var channelV;
CosEventChannelAdmin::ConsumerAdmin_var consAdminV;
CosEventChannelAdmin::ProxyPushSupplier_var ppsV;
...

// Assume that the reference ’channelV’ has already
// been initialized

try {
   consAdminV = channelV->for_consumers();
   ppsV = consAdminV->obtain_push_supplier();
}
catch (CORBA::Exception& ex) {
   // Handle exception...
}
// The reference ’ppsV’ is now initialized and
// ready for use.
```

Connecting to a ProxyPushSupplier

The final step in the attachment of a consumer to the channel is when the consumer invokes the appropriate connect operation on the PPS object. Two IDL interfaces are important at this stage: the `ProxyPushSupplier` interface and the `PushConsumer` interface. The relevant IDL from the event service is shown in the section "Connecting to a ProxyPushConsumer" on page 133.

The consumer finishes attaching itself to the channel by calling `connect_push_consumer()` on the PPS object.

Note that this operation requires a reference to a `PushConsumer` object. Therefore, before the consumer can connect it must provide an implementation of a `PushConsumer` object.

The implementation of the `PushConsumer` object is of key importance to the consumer application. It provides the implementation of the `push()` operation that the consumer uses to receive events from the event service. This interface is implemented by the class `PushConsumer_i`. A sample implementation is given in the next section.

Assuming that the implementation `PushConsumer_i` is given, the following few lines of code can be used to connect to the `ProxyPushSupplier` in the event channel:

```
// C++

...
PushConsumer_i * pcImpl = new PushConsumer_i();

// The reference to the ProxyPushSupplier 'ppsV'
// has already been initialized above
ppsV->connect_push_consumer(pcImpl);

// The remote ProxyPushSupplier object is now in
// its 'connected' state and ready to supply events
```

Consuming Events

The consumer receives events by providing an implementation of the PushConsumer interface. The code that is responsible for receiving events is found in the implementation of the push() operation. The class PushConsumer_i provides a sample implementation, as follows:

```
// C++

#include <EventService.hh>

class PushConsumer_i : public virtual
        CosEventComm::PushConsumerBOAImpl {
   // Private member variables
   int   m_disconnected;

public:
   // C'tor and D'tor
   PushConsumer_i() : m_disconnected(0) { }

   //
   // IDL Operations
   //
   virtual void disconnect_push_supplier() {
      m_disconnected = 1;
   }
   virtual void push (CORBA::Any& any) {
      char * msg;

      if (a >>= msg)
         cout << "Event: " << msg << endl;
      else
         cout << "Event: Unexpected data type" << endl;
   }
};
```

Note the use of the operator >>= (right shift assignment) to extract the string value from the any. The above implementation expects the value in the any to be a string. If the type of data contained in the any is *not* a string, the return value of the expression (a >>= msg) is false and the value is not extracted. To learn more about how to extract values from an any consult the *Orbix C++ Programmer's Guide*.

As explained in “Callback Objects” on page 121, the `PushConsumer` object is an example of a callback object and should be ready to receive events at any time. An Orbix event loop must be running in the consumer even if it is just a client application. Typically, in a client application, this event loop would be set running in a subthread or else integrated with another event loop. These options are described in the *Orbix C++ Programmer’s Guide*.

Extending the StockWatch Example

In the StockWatch demonstration, example 2, the event service is used to monitor changes in stock prices. The StockWatch server plays the role of event supplier, propagating news of price changes to clients via the event service. Clients play the role of consumers and receive events via `PushConsumer` callback objects.

Figure 6.5 on page 140 shows the basic architecture of the StockWatch example when it uses events. When a client of the StockWatch application records the sale of a stock, the server generates an event that represents the new price and pushes the event onto that stock’s event channel. At this point the server is acting as an event supplier.

Figure 6.5 on page 140 illustrates the implementation of an event propagation in a CORBA system as follows:

1. When a client of the StockWatch application records a sale of a stock, the server updates the database with the new price.
2. The server then generates an event that represents the new price, and pushes the event onto that stock’s event channel. In this case the server is acting as a supplier to the event channel.
3. The OrbixEvents server maintains an event channel for each stock. When an event is pushed onto a channel, the channel propagates it to all consumers that are connected to the channel.

   Both the StockWatch server and the event consumers, in this example, are clients of the OrbixEvents service.

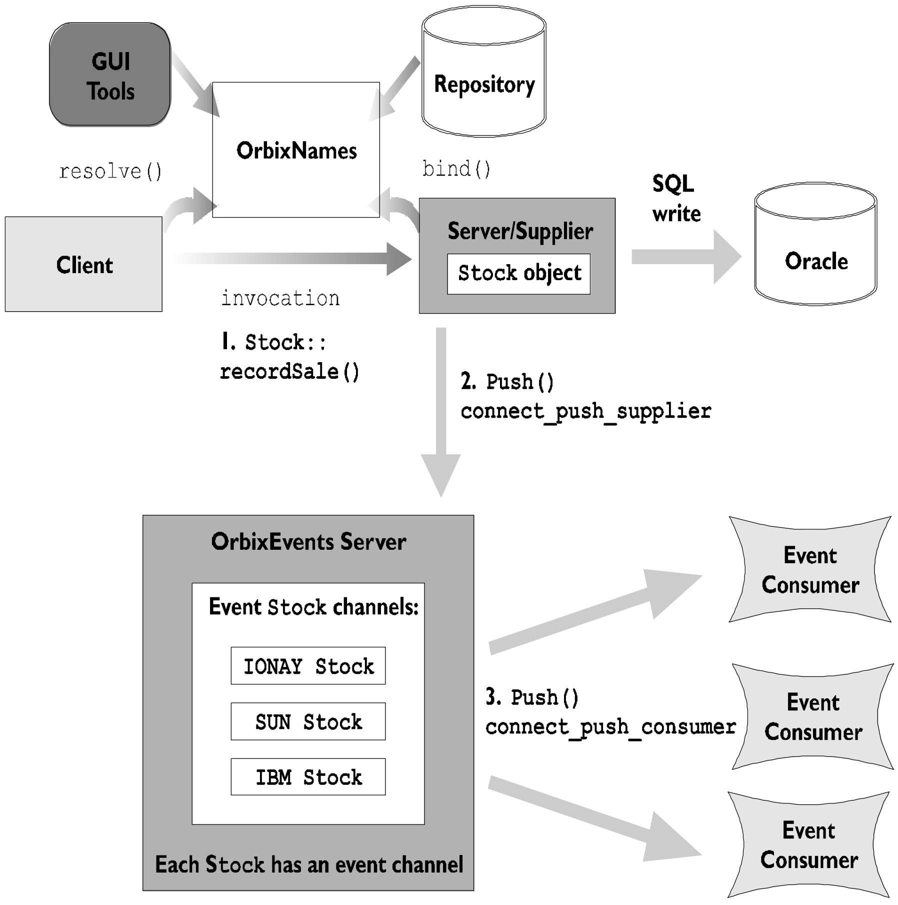


Figure 6.5: *The Architecture of StockWatch When Using Events*

Initializing the StockWatch Server

The StockWatch server locates the event service by using either `_bind()` or the naming service to obtain a reference to the `ChannelManager` object in the event server.

As each `Stock` object is created in the server's address space, a check is made to see if an event channel exists for that stock by invoking `find()` on the remote `ChannelManager` object. If not, a new event channel is created for the stock by invoking `create()` on the remote `ChannelManager` object.

The server attaches itself to each of the active event channels so that it is ready to begin supplying events.

Initializing the StockWatch Consumers

A different approach to locating event channels is taken by the clients of StockWatch. With the help of some additions to the StockWatch IDL file, a simpler approach to bootstrapping the clients (consumers) is possible. The relevant extract of IDL from the `StockWatch.idl` file is as follows:

```
      // CORBA IDL for the StockWatch example
      //
1     #include "EventService.idl"
      //
      // Basic type definitions, Exceptions, Structs
      ...
      // Interfaces
         interface Stock {
            Symbol getSymbol ();
            string getDescription ()
               raises (rejected);
            Money getCurrentPrice ()
               raises (rejected);
            PriceInfoSeq getRecentPrices ()
               raises (rejected);
            void recordSale(in Money price)
               raises (rejected);
2           CosEventChannelAdmin::ConsumerAdmin getFeed ();
         };
      ...
```

The code is described as follows:

1. The IDL file includes the `EventService.idl` to make available the data types and definitions required for OrbixEvents.
2. The `getFeed()` operation provides a convenient short cut for clients to attach themselves to the event channel corresponding to this `Stock` object. For example, when `getfeed()` is invoked on the `"IONA"` `Stock` object it returns a reference to the `ConsumerAdmin` object for the `"IONA"` event channel.

   This simplified approach to finding an event channel has the benefit that clients can use standard event service interfaces.

7

The Object Transaction Service

The concept of a transaction and a distributed transaction is explained. The programming principles of OrbixOTS are explained and code examples presented. Other topics such as the two-phase commit protocol and multithreaded programming with OrbixOTS are also discussed.

OrbixOTS brings transactional capability to developers creating enterprise-wide applications. So far, the example applications in this guide have used a single database. Figure 7.1 shows that although the client invokes a function on an object in the server, only the server and the database need to be involved in the transaction. From the client's point of view, the transaction is completely hidden.

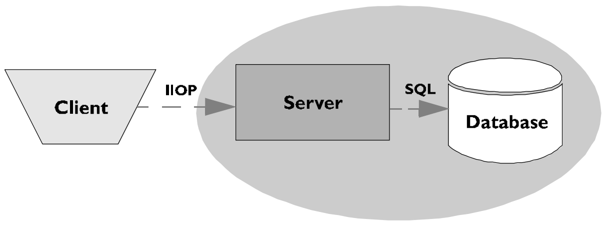


Figure 7.1: *A Simple Transaction*

When applications need to manage transactions across multiple, distributed data resources, transaction demarcation (the beginning and end of a transaction) becomes the responsibility of the client. Also, the Object Transaction Service, OrbixOTS, is needed to co-ordinate all the components involved in the transaction.

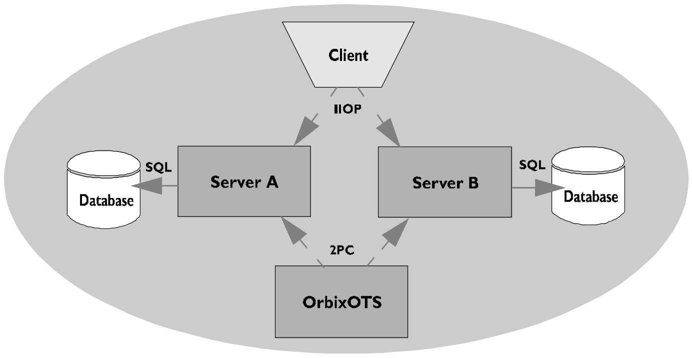


Figure 7.2: *A Distributed Transaction*

The classical illustration of a transaction is that of funds transfer in a banking application. This involves two operations: a debit of one account and a credit of another (perhaps after extracting an appropriate fee). To combine these operations into a single unit of work, the following properties are required:

- If the debit operation fails, the credit operation ought to fail, and vice-versa: that is, they should both work or both fail.
- The total amount of money in the system should be the same, before and after the transaction.
- The system goes through an inconsistent state during the process (between the debit and the credit). This inconsistent state should be hidden from other parts of the application.
- It is implicit that committed results of the whole operation are permanently stored.

These points illustrate the so-called *ACID* properties of a transaction:

Atomic—A transaction is an all or nothing procedure; individual updates are assembled and either all committed or all aborted (rolled back) simultaneously when the transaction completes.

Consistent—A transaction is a unit of work that takes a system from one consistent state to another.

Isolated—While a transaction is executing, its partial results are hidden from other entities accessing the system.

Durable—The results of a transaction are persistent.

Thus a transaction is an operation on a system that takes it from one persistent, consistent state to another.

Example of a Distributed Transaction

Consider the example of a funds transfer between two accounts. The actions of debiting a sum of money from one account and crediting that sum of money to another account should take place within the context of a single transaction. A sample definition of an account object is given by the following IDL[1]:

```
// IDL
#include <OrbixOTS.idl>
exception DBError { string reason; };

interface TransAccount : CosTransactions::TransactionalObject {
     void makeWithdrawal(in float amount) raises (DBError);
     void makeDeposit(in float amount) raises (DBError);
     void query(out string accName, out float accBalance)
        raises (DBError);
};
```

1. This IDL is extracted from the `transbank` demonstration, which can be found in the OrbixOTS demos directory.

The TransAccount interface provides basic operations for manipulating an account, allowing you to query the account balance, withdraw and deposit money. In order to participate properly in a distributed transaction, the account interface must be defined to be transactional—that is why TransAccount inherits from CosTransactions::TransactionalObject.

The transaction under consideration involves making a debit from one account, by invoking makeWithdrawal(), and a credit to a second account, by invoking makeDeposit(). If the reference to the first account is called srcAccount and the reference to the second account is called destAccount, the transaction can be described by the following steps:

```
// Pseudo-code
// Transaction: Transfer funds from srcAccount to destAccount

// Beginning of transaction
srcAccount->makeWithdrawal(amountToTransfer);
destAccount->makeDeposit(amountToTransfer);
// End of transaction
```

Different scenarios for the execution of transactions are considered in the following sections:

1. One OTS client invoking on two OTS servers.
2. One ordinary client invoking on two OTS servers.
3. One OTS client invoking on one OTS server.

One OTS Client Invoking on Two OTS Servers

The first scenario for carrying out the funds transfer is shown in Figure 7.3. All of the participants are transactional in this scenario, with one OTS client invoking on two OTS servers. The client transfers money by invoking makeWithdrawal() on the object srcAccount and invoking makeDeposit() on the object destAccount.

Both of the account objects involved in the transaction are specified as transactional, which is indicated in Figure 7.3 as a dark outline around the transactional objects.

In this example the client invokes *directly* on the two transactional objects. To make the sequence of invocations atomic, the client must bracket them between calls to *begin* and *commit* the transaction. The client is thus responsible for transaction demarcation and plays the role of *transaction originator*.

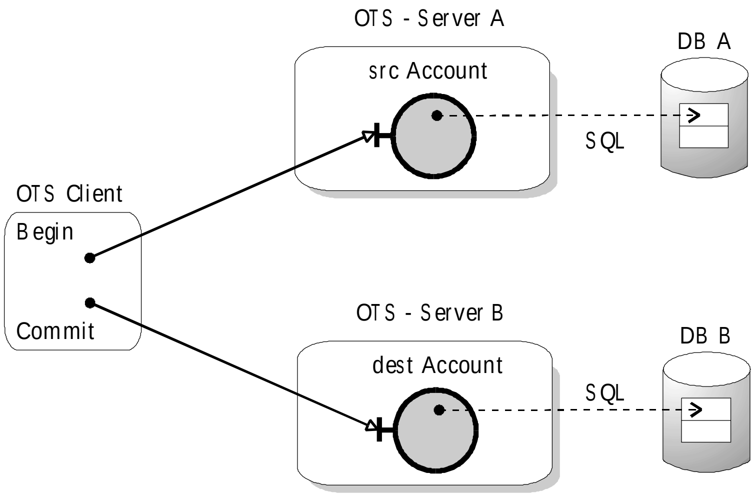


Figure 7.3: *Distributed Transaction with OTS Client.*

In order to synchronize the transaction extra information, known as *transaction context*, passes back and forth between the participants. The extra messages needed to transfer transaction context are sent automatically by OrbixOTS and are transparent to the application developer.

This scenario is most appropriate for the case where both servers and clients are physically close together and connected by a high quality network.

In the case where clients are connected to the OTS servers by a poor quality network, for example clients connected by a dialup network or via the Internet, this approach to transactions is best avoided. The transaction monitor would then be attempting to commit a transaction over an unreliable network, giving rise to frequent rollbacks and aborted transactions.

One Ordinary Client Invoking on Two OTS Servers.

The second scenario for carrying out a funds transfer is shown in Figure 7.4. In this example, only the two OTS servers are transactional. The client is an ordinary non-transactional client (it does not need to be linked with any OTS libraries).

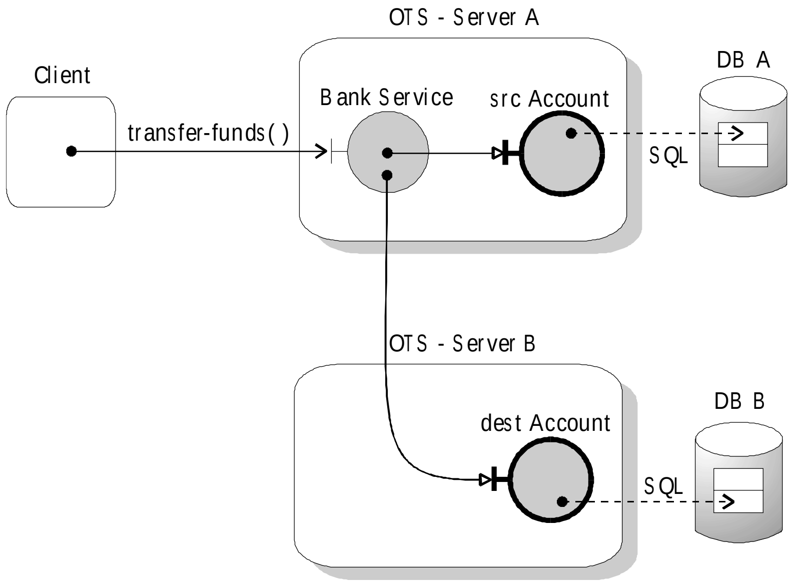


Figure 7.4: *Distributed Transaction with Ordinary Client.*

As pointed out in the previous section, it is better to limit the scope of a distributed transaction to include the smallest number of participants possible. This approach minimizes the risk of network problems interfering with the transaction.

In this example, a new object of type `BankService` is introduced in Server A. The `BankService` is a non-transactional CORBA object that exposes a single operation `transfer_funds()`. It is the implementation of `transfer_funds()`

that carries out the individual steps of the transaction in this example, including bracketing the invocations between calls to *begin* and *commit*. The BankService object plays the role of transaction originator in this scenario.

The client can now carry out the whole transaction by making a single invocation of operation transfer_funds() on the ordinary object BankService. In general, it is a good idea to expose to remote clients only IDL operations that represent a single atomic transaction.

One OTS Client Invoking on One OTS Server.

The third scenario shown in Figure 7.5 has an OTS client invoking on a single OTS server. Strictly speaking, this is not a distributed transaction at all because there is only one transactional server and one database involved. If the invocations on this transactional object can be made independently of other transactional servers then there is no need for the object to be transactional.

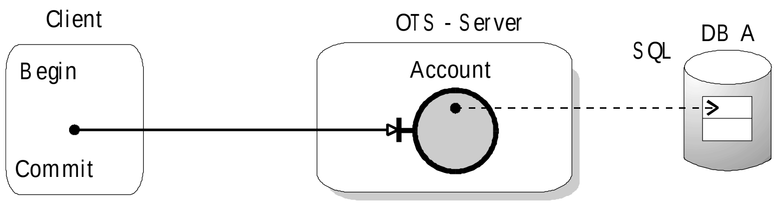


Figure 7.5: *OTS Client Invoking on a Single OTS Server.*

However, it often arises that a transactional object exposes some operations that are intended to be executed in the context of a distributed transaction, while other operations are atomic and can be invoked individually.

For example, an OTS client that wants to query the balance of a TransAccount object can execute the following pseudo-code:

```
// Pseudo-code
// Transaction: Query an account balance

// Beginning of transaction
Account->query(accountName,returnedBalance);
// End of transaction
```

This transaction is trivial; there is only one invocation bracketed between the *begin* and *commit*. Although there is no real coordination needed to commit this transaction the call to *begin* and *commit* is still required in OTS. The transaction service normally requires that a transaction begin before an operation is invoked on a transactional object. Otherwise, a CORBA system exception is raised by the ORB. In OrbixOTS, however, an option can be set to relax this restriction.

StockWatch Example

The transactional StockWatch demonstration of example 3 is organized along the lines of the scenario depicted in Figure 7.5. That is, an OTS client talks to a single OTS server StockWatch.

The default setup of StockWatch does not, therefore, illustrate a truly distributed transaction. It has the practical advantage of being easy to set up because only one database has to be configured. If you want to experiment with distributed transactions you can modify the example to connect to two transactional servers at once. Alternatively, you can run one of the demos from the OrbixOTS directory that does connect to more than one database (scenario depicted in Figure 7.3).

Modification of StockWatch IDL for OTS

Before your clients and servers can handle distributed transactions, the objects involved have to be made transactionally aware. The objects that participate in transactions are called *transactional objects*.

The following IDL file shows the interface for two transactional interfaces: `StockWatchOTS` and `StockOTS`. Transactional objects must receive transaction context with each invocation. The programmer can pass the transaction context as additional arguments (explicit propagation). However, the easiest approach is to have the IDL definition of your interface inherit from `CosTransactions::TransactionalObject`. This instructs OrbixOTS to include the transaction context with each IIOP request (implicit propagation).

```
     // StockWatchOTS.idl
     //
1    #include <OrbixOTS.idl>

     // Basic type definitions
     typedef string Date;
     typedef float  Money;
     typedef string Symbol;
     typedef sequence<Symbol> SymbolSeq;

     // Exceptions
     exception rejected {
        string m_reason;
     };

     // Structs
     struct PriceInfo {
        Money m_price;
        Date  m_when;
     };
     typedef sequence<PriceInfo> PriceInfoSeq;

     // Interfaces
2    interface StockOTS : CosTransactions::TransactionalObject {

        Symbol getSymbol();
        string getDescription()
           raises (rejected);
        Money getCurrentPrice()
           raises (rejected);
        PriceInfoSeq getRecentPrices()
           raises (rejected);
        void recordSale (in Money price)
           raises (rejected);
     };

3    interface StockWatchOTS : CosTransactions::TransactionalObject {

        SymbolSeq getSymbols ()
           raises (rejected);
        Stock getStockBySymbol (in Symbol aSymbol)
           raises (rejected);
     };
```

The code is described as follows:

1. Include in your interface the IDL file that specifies the interfaces for OrbixOTS.
2. Make the `StockOTS` interface transactional by specifying that the interface is derived from the `CosTransactions::TransactionalObject` interface.
3. Make the `StockWatchOTS` interface transactional by specifying that the interface is derived from the `CosTransactions::TransactionalObject` interface.

The definition of interface `TransactionalObject` is given in the file `OrbixOTS.idl` as follows:

```
// IDL
// File: OrbixOTS.idl
module CosTransactions {
   interface TransactionalObject {};
   ...
};
```

The definition of the `TransactionalObject` interface is trivial. Its sole purpose is to act as a marker. Interfaces that inherit from it are treated by the OTS as transactional.

Controlling Database Transactions

There are typically three options available for controlling database transactions:

1. Embedded SQL
2. Database Native Interface
3. XA Interface

The first two are used for controlling local transactions.

Embedded SQL

If the interaction with the database is coded in embedded SQL, control of transactions is available via the following SQL statements:

SQL	Effect on Transaction
(implicit)	Begin transaction
`EXEC SQL COMMIT WORK`	Commit (successful outcome)
`EXEC SQL ROLLBACK WORK`	Rollback (abort transaction)

Generally, a transaction is implicitly begun whenever you access the database. The `EXEC SQL COMMIT WORK` statement is called after a successful outcome to the transaction and indicates that you want the changes to the database to be made permanent. The `EXEC SQL ROLLBACK WORK` statement indicates a failed outcome and signals the database to forget details of the transaction.

These are not the only embedded SQL statements that can affect a transaction. For example, there are some SQL statements that can commit a pending transaction as a side effect of their execution. You must consult your database documentation for a complete list of SQL statements that affect transactions.

Database Native Interface

In addition to embedded SQL, most databases provide the option of using a native interface. This includes, for example, function calls for committing and rolling back transactions. The effect of calling these functions is typically the same as using their embedded SQL equivalents.

XA Interface

Most modern databases provide an XA interface for controlling transactions (an industry standard defined by the X/Open group). The motivation behind this standard was a desire to open up databases to the control of distributed transaction monitors, such as OrbixOTS. It turns out that the traditional interfaces for steering transactions are unsuitable for this purpose because they allow an inadequate degree of control. In particular, explicit support is needed for the *two-phase commit protocol* (described below) in distributed transaction systems. The necessary level of support is provided by the XA interface.

XA Operation	Purpose
`xa_open()`	Opens the connection to the resource manager. This is called during initialization.
`xa_close()`	Closes the connection to the resource manager.
`xa_start()`	Informs the resource manager that a thread or process has started working on behalf of a transaction.
`xa_end()`	Informs the resource manager that a thread or process has finished working on behalf of a transaction.
`xa_rollback()`	Rolls back modifications made by a transaction to the resource manager.
`xa_prepare()`	Prepares the resource manager for eventual commitment of a transaction. The resource manager returns its vote.
`xa_commit()`	Commits modifications made by a transaction to the resource.
`xa_recover()`	Retrieves the identifiers of transactions for which the resource manager needs to know the final outcome. This is called during initialization.
`xa_forget()`	Informs the resource manager that a heuristic decision may be forgotten.
`xa_complete()`	Completes an asynchronous call.

Table 7.1: *Methods of the XA Interface*

The form in which the XA interface is expressed is object-oriented but, for pragmatic reasons, it had to be defined in a non-object oriented language 'C'. The X/open committee therefore used the 'C' style approximation to an object: a struct containing a collection of pointers to functions. This pseudo-object is known as an *XA switch*. The methods of the XA switch are summarized in Table 7.1 on page 154.

In general, it is not intended for this interface to be called directly by the application developer. Usually the interface is controlled by a distributed transaction monitor instead.

In most cases, the XA interface is not meant to be used at the same time as other means of transaction control (embedded SQL or native interface) from within the same process.

The OTS as Transaction Manager

When there is more than one database in a system and two or more of these databases are involved in a transaction, it becomes essential to place these databases under the control of a single transaction manager. The transaction manager is needed to properly coordinate a transaction involving multiple databases or, more generally, *resources*.

The database, or resource, must export its XA interface to the transaction manager and, in this way, it gives up control of its transactions to the transaction manager (in this case, OrbixOTS). It is useful to remember that all of the interaction between transaction manager and database happens via this XA interface. There are other things going on between the application and the database, but the rest of this activity is due to code written by the application developer (for example, SELECTs and UPDATEs to read from and write to the database).

To take a specific example, look at how the database Oracle exports its XA switch to the OrbixOTS transaction manager. This is known as *registering the resource* with OrbixOTS. The code appears in an OTS server process (only OTS server processes are able to register resources):

```
// C++
#include <OrbixOTS.hh>

extern "C" {
  // The Oracle XA switch.
  extern struct xa_switch_t xaosw;
}

int main()
{
  OrbixOTS::Server_var ots;

  //--------------------------------------------
  // Various initialization steps (not shown)
  //--------------------------------------------
  // ...

  //--------------------------------------------
  // Register XA resource with OTS txn manager
  //--------------------------------------------
  char * openString =
         "Oracle_XA+Acc=P/scott/tiger+SesTm=60";
  char * closeString = "";

  CORBA::Long rm_id = ots->register_xa_rm(
                 &xaosw,
                 openString,
                 closeString,
                 0);
  // ...
}
```

The XA switch is provided by Oracle in the form of a global static instance of the struct called xaosw. Since the struct is instantiated within the Oracle libraries, it means that xaosw is an Oracle-specific identifier that is always used to identify the Oracle XA switch.

Initialization of OrbixOTS is carried out via the OrbixOTS::Server[2] object. The key step is the call to register_xa_rm(). The address of the switch xaosw is passed as the first argument, establishing the link between transaction manager

2. This object is non-standard because the CORBA OTS specification does not specify the details of how an OTS initializes itself. That is an implementation detail.

and database resource. The OTS transaction manager is now able to make calls on `xa_open()`, `xa_close()`, and so on via the pointers to functions defined in the struct `xaosw`.

The `openString` is the string that will be passed as an argument to `xa_open()` when it is called, later on, by the transaction manager. Likewise `closeString` is the string argument that will be passed to `xa_close()`. The last argument of `register_xa_rm()` is used to indicate whether the database is operating in concurrent (multithreaded) mode. In this example a `0` (zero) is passed because the server is accessing the database in single-threaded mode.

Format of the Open String

The format of the `openString` is specified neither by the X/Open group nor by OrbixOTS. To determine the correct format of the `openString` you must consult the documentation for the particular database you are using, and check the required format of the string passed to `xa_open()`. For example, in the case of Oracle the most basic `openString` includes the following mandatory fields

`Oracle_XA`
`Acc=P/`*user/password*
`SesTm=`*session_time_limit*

joined by a '+' (plus sign). Putting these mandatory fields together for the Oracle account `scott/tiger` gives the following sample `openString`:

```
"Oracle_XA+Acc=P/scott/tiger+SesTm=60"
```

Note that Oracle allows for other optional fields in the open string. In particular, extra options are required if you want to run the database in concurrent mode.

The effect of the call `xa_open()` (a call made automatically by the transaction manager) is to establish a session with the database. However the connection to the database might not be opened right away, depending on how your database vendor has implemented `xa_open()`.

The call to `xa_open()` also establishes the fact that the XA interface is being used to control database transactions.

Format of the Close String

The close string is passed as an argument to `xa_close()`. However, Oracle does not make use of this string and it can be left empty. Most other databases ignore the close string as well.

Writing an OTS Server

In this section, the StockWatch server is extended to use OrbixOTS as the transaction co-ordinator. The backend database's XA interface is registered with OrbixOTS.

Server programming is divided into two sections:

- The first code sample initializes the server so that it can use distributed transactional objects.
- The second section briefly describes how the implementation of a transactional object interface is different from a non-transactional implementation.

Initializing a Transactional Server

Servers are implemented as objects in OrbixOTS applications. The OrbixOTS C++ interface supplies the server class that you use to initialize servers. The following code sample shows the `main()` function of the `example3/serverOracle.cxx` file.

```
// serverOracle.cxx
//
// Orbix OTS header files
#include <OrbixOTS.hh>
...
int main(int argc, char **argv) {
   if (!argv[1] || !argv[2])
      syntax_error();

   // OTS required variables for registering an OTS
   oracle_backend orac;
```

```
        // Instantiate my instance of StockWatchOTS
        StockWatch_i swatch(&orac,1);

        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "Orbix");
        CORBA::BOA_var boa = orb->BOA_init(argc, argv, "Orbix_BOA");

        try {
           ...
           // Create an instance of OTS::Server
1          OrbixOTS::Server_var ots = OrbixOTS::Server::IT_create();
           ots->serverName(serverName);
           ots->logDevice(logfile);
           ots->restartFile(recoveryFile);
           ots->mirrorRestartFile(mirrorRecoveryFile);


           // Register XA/Resource
2          int serverid = ots->register_xa_rm
                    (&xaosw,
                     openString,
                     closeString,
                     0);


           // initialize OTS
3          ots->init();


           // Now wait for incoming invocations
4          // OTS::Server::impl_is_ready calls
           // CORBA::ORB::impl_is_ready
           ots->impl_is_ready(TIMEOUT);

        } catch (...)

5       ots->shutdown();
     }
```

The code is described as follows:

1. Create and initialize an instance of `OrbixOTS::Server` The `OrbixOTS::Server` object needs the following configuration information:

Server name	The string that is actually passed to `CORBA::Orbix::impl_is_ready()`.
Log device	As explained earlier, there is no centralized transaction co-ordinator in the implementation of OrbixOTS. Instead any server that participates in a transaction can potentially become transaction co-ordinator. A co-ordinator requires a logging facility to avoid problems arising due to resource failure during two-phase commit.
	OrbixOTS allows both ordinary files or raw devices to be used for the transaction log. Typically, ordinary files are used in the early development stages; however in a deployed system, a raw device is usually required.
	OrbixOTS supports transaction log mirroring.
Restart file	This is the path for a file that contains information about the log. It is created the first time the server is run and the log file is formatted. On subsequent runs it is read and used to read the log during recovery processing.
Restart mirror	This is a mirror copy of the restart file, which is used for redundancy. This must be specified.
	In a deployed system the restart file and mirror restart file should be on separate disks.

2. An XA-compliant resource, in this case Oracle, is registered with OrbixOTS. All transactional client requests that visit this server process cause OrbixOTS to register Oracle with the transaction co-ordinator. `OrbixOTS::Server::register_xa_rm()` requires the following arguments:

XA structure	A pointer to an xa_switch_t (as detailed below).
A database open string	As stated earlier, when an XA resource is registered with OrbixOTS, connection to the database is controlled via XA and not by SQL. In Oracle, the XA open string has the following format: `"Oracle_XA+Acc=P/<account> <passwd>+SesTm=60"`
A database close string	Typically empty.
Association flag	TRUE indicates that the XA resource supports *multiple* association. That is, several threads can call XA functions concurrently. FALSE indicates that the XA resource supports only single association. That is, calls to XA functions must be serialized.

An XA client resource is required to export a struct of type xa_switch_t. The following is a 'C' definition of xa_switch_t:

```
   struct xa_switch_t {
      char name[RMNAMESZ];
      long flags;//resource mgr specific options
      long version;
      int(*xa_open_entry)(char*,int,long);
      int(*xa_close_entry)(char*,int,long);
      int(*xa_start_entry)(XID*,int,long);
      int(*xa_end_entry)(XID*,int,long);
      int(*xa_rollback_entry)(XID*,int,long);
      int(*xa_prepare_entry)(XID*, int,long);
      int(*xa_commit_entry)(XID*,int,long);
      int(*xa_recover_entry)(XID*,long,int,long);
      int(*xa_forget_entry)(XID*,int,long);
      int(*xa_complete_entry)(int*,int*,int,long);
};
```

3. The `OrbixOTS::Server::init()` function initializes the underlying OrbixOTS components and services. This must be done after any XA resources have been registered. This function also drives OTS recovery (see the *OrbixOTS Programmer's and Administrator's Guide*).
4. After server objects are created, the server must tell Orbix that its implementation is ready to service requests.

   The operation `OrbixOTS::Server::impl_is_ready()` can optionally take a concurrency mode, as a second parameter. Possible values are as follows:

   - `OrbixOTS::Server::serializeRequestsAndTransactions` (default). This concurrency mode only allows one transaction and one request to be active in the server at a time. That is, once a transaction accesses a server, no other transaction can enter that server until that transaction is complete.
   - `OrbixOTS::Server::concurrent`. This concurrency mode applies no serialization constraints. That is, requests that are part of different global transactions run concurrently in the server.
   - `OrbixOTS::Server::serializeRequests`. This mode is a compromise between the two previous modes. Requests that are part of different global transactions are serialized as they arrive at the server.

   The operation `OrbixOTS::Server::impl_is_ready()` allows you to specify an inactivity timeout as the first argument. This timeout is specified in milliseconds, just like the regular `CORBA::BOA::impl_is_ready()` function.

   The inactivity timeout for the Orbix event loop can also be set via the environment variable `OTS_LISTEN_TIMEOUT` or via the configuration variable `OrbixOTS.OTS_LISTEN_TIMEOUT`, specifying the duration of the timeout in milliseconds.
5. Before terminating the server application, you must call `OrbixOTS::Server::shutdown()` to shut down the OrbixOTS services. The application continues as normal after the call to `shutdown()`, allowing you to perform miscellaneous housekeeping tasks before the server terminates[3].

3. The use of `OrbixOTS::Server::exit()` is now deprecated.

Implementing a Transactional Class

You implement transactional classes much like normal interface classes, using standard Orbix programming and standard embedded SQL for the backend. However, what is different about implementing transactional classes is the coding you do *not* do.

- By registering an XA resource manager for the database in the `main()` server function, the resource manager opens a connection to the database. Therefore, your code should not use SQL to open a connection to the database. In this implementation, the `db_connect()` function, shown in the file `db_oraapi.pc`, is never called.
- Since distributed transaction demarcation is controlled by clients, your SQL code does not need to explicitly begin and end transactions. SQL can still be used to access the database, but transaction control is with the client and the transaction manager.

Writing an OTS Client

A client program that invokes directly on transactional objects is known as an OTS client or a transactional client. It must be linked with an OTS client library.

Initializing a Transactional Client

The following code sample shows how to initialize clients using the client class supplied with the OrbixOTS C++ interface. This code sample is from the `main()` function of the file `clientots.cxx`.

```
      // C++
      // clientots.cc

      // OTS header file
1     #include <OrbixOTS.hh>
      ...
      int main(int argc, char **argv) {
         StockWatchOTS_var swatch;

         CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "Orbix");
         ...
```

```
        // Initialize OTS
2       OrbixOTS::Client_var clientots = OrbixOTS::Client::IT_create();
        clientots->init();

        try {
           // Main application loop
        } catch (...) { }

3       clientots->shutdown();
     }
```

The code is described as follows:

1. Include the `OrbixOTS.hh` header file to gain access to OTS definitions.
2. Create an instance of an `OrbixOTS::Client` object by invoking the static method `IT_Create()`. The `OrbixOTS::Client` class is a singleton class, so a call to `IT_Create()` always returns a reference to the same object.

   The member function `OrbixOTS::Client::init()` is then invoked to initialize the OrbixOTS client application.
3. Before terminating, the client application must call `OrbixOTS::Client::shutdown()` to shut down the OrbixOTS services. All of the transactions in progress are completed (either committed or rolled back). The application continues as normal after the call to `shutdown()`, allowing you to perform miscellaneous housekeeping tasks before the client terminates[4].

Making a Transactional Invocation

Distributed transactions expose transaction programming to the client application. The rules of basic transaction programming are straightforward: after initializing a client object, the client has to do three basic steps for each transaction:

1. Begin the transaction.
2. Invoke functions within the transaction.
3. Initiate a commit or rollback to end the transaction.

4. The use of `OrbixOTS::Server::exit()` is now deprecated.

The following extract is taken from the `clients/cxx/menuots.cxx` file from the StockWatch demonstration:

```
     // C++
     void printSymbols(StockWatchOTS_ptr swatch){

        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "Orbix");
        CORBA::BOA_var boa = orb->BOA_init(argc, argv, "Orbix_BOA");

        CosTransactions::Current_var current;

        try {
           SymbolSeq_var ss;

           // Get 'Current' using initialization service
1          CORBA::Object_ptr objP =
                 orb->resolve_initial_references("TransactionCurrent");
           current = CosTransactions::Current::_narrow(objP);
           if (CORBA::is_nil(current)) {
              cerr << "Narrow failed on Current" << endl;
              exit(1);
           }

2          current->begin();

3          ss = swatch->getSymbols();

4          current->commit(TRUE);

           for (CORBA::ULong i = 0; i < ss->length(); i++){
              cout << "   " << ss[i] << endl;
           }
           cout << endl;
        }
5    } catch(CORBA::TRANSACTION_ROLLEDBACK){
           cerr << "transaction rolledback" << endl;
           throw;
        }
6       catch(...){
           current->rollback(); // end the transaction
           throw;
        }
     }
```

The code is described as follows:

1. An instance of the class `CosTransactions::Current` is created using the initialization service. A different `Current` object is implicitly associated with each thread in an application.
2. Clients begin a transaction by calling the function `CosTransactions::Current::begin()`. The `Current` object can be used to manage different concurrent transactions, one per calling thread.
3. The application-specific function `getSymbols()` executes within the scope of the transaction.

Note: For simplicity, this example has only one participating server and database.

4. Call the `CosTransactions::Current::commit()` function to commit the current transaction. This call ends the transaction and starts the two-phase commit processing. The transaction is committed only if all of the participants in the transaction agree to commit.

   The boolean argument to `commit()` is the `report_heuristics` flag that specifies whether heuristic decisions should be reported for the transaction associated with the calling thread. Normally it is best for this flag to be set to `TRUE`.
5. If a commit fails, the OrbixOTS runtime system throws the `CORBA::TRANSACTION_ROLLEDBACK` exception indicating that the transaction rolled back and the caller is disassociated from the transaction.
6. If any other exception is thrown the caller should call `CosTransactions::Current::rollback()` to explicitly rollback the current transaction.

The association between the transaction and the client process ends when the client calls either of the functions `CosTransactions::Current::commit()` or `CosTransactions::Current::rollback()`.

Implicit Indirect and Explicit Direct Modes

Implicit indirect mode, as used in the preceding example, is the usual mode in which the OTS is used and it is recommended for the majority of applications. It allows transactionality to be configured at an object level of granularity. In this mode, therefore, you can talk of *transactional* and *non-transactional* objects. This mode is also the easiest to use as it demands minimal coding by the developer. Transaction context is implicitly propagated to all participants in a transaction.

In contrast, the explicit direct mode is much harder to use because transaction context must be explicitly propagated. Explicit coding is required by the developer to make a particular process (or thread) into a participant of the transaction. This mode does allow a more fine-grained configuration of transactionality—allowing you to specify whether individual IDL operations are transactional or not. Generally, this mode is recommended in special cases where the additional flexibility proves indispensable.

Two-Phase Commit Protocol

Consider a client that wants to execute a transaction that requires two servers to update their respective databases. Figure 7.6 through Figure 7.9 depict this situation—two servers, each with its own database, are distributed using OrbixOTS. OrbixOTS mediates between the applications, ensuring that the database updates are performed atomically.

Client Begins the Distributed Transaction

A client begins a transaction by making a call on OrbixOTS (Figure 7.6). The client is now in the context of a created transaction.

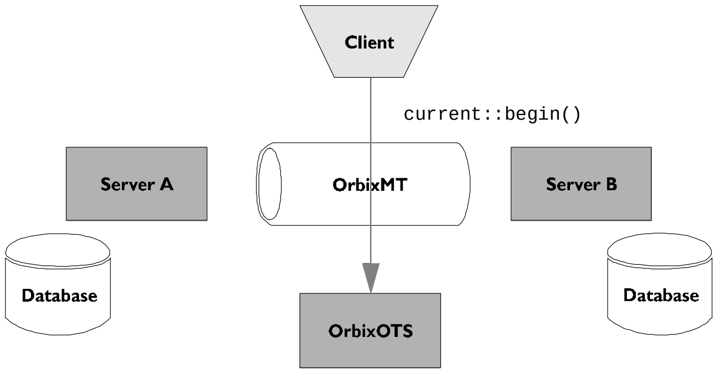


Figure 7.6: *Client Begins a Transaction*

OrbixOTS is shown here as separate from the applications; however, this is only conceptual. In fact the transaction manager is implemented as a linked-in library; hence, transactional applications each have an instance of the transaction manager that co-operate to implement distributed transactions. This architecture has the advantage that there is no dedicated ‘transaction server’ that could be a central point of failure and bottleneck.

Client Invokes Operations within the Transaction

When a client invokes a transactional object in a server, some transactional context accompanies each request. Transactional contexts contain such details as a global transaction identifier and a reference to a transaction co-ordinator.

Using *implicit* context propagation the transaction context is automatically included in the service context field of each IIOP request message. Alternatively, *explicit* context propagation requires the programmer to pass the transaction context as an argument to IDL operations.

On invocation of an IDL operation that uses a `Resource`, the resource must be registered with the transaction `Coordinator`. A resource must only be registered once with the co-ordinator: subsequent invocations within the context of the same transaction should not re-register the resource.

Registration of XA-compliant resources is much simpler. An XA resource is registered once with OrbixOTS, typically when the server starts up.

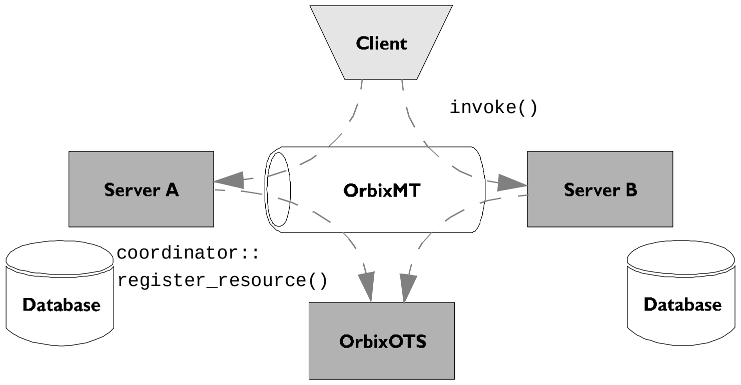


Figure 7.7: *Server Registers with OrbixOTS when Client First Invokes*

Client Ends the Transaction

The client ends the transaction by invoking a commit or rollback operation on OrbixOTS (Figure 7.8).

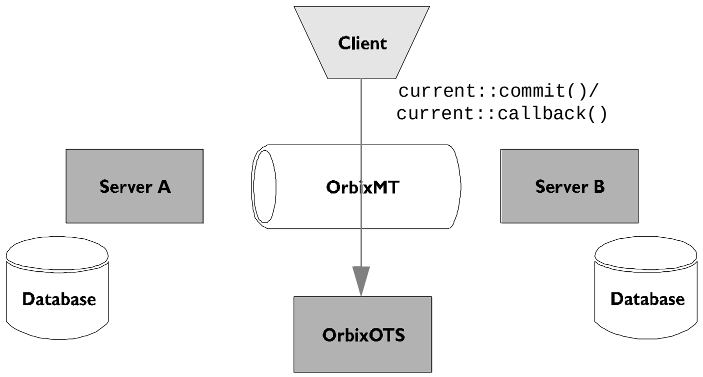


Figure 7.8: *Client Ends a Transaction*

OrbixOTS Co-ordinates a Two-Phase Commit

After the application code has invoked `commit()` on the OTS transaction manager, a number of extra steps are executed automatically by the OTS in order to complete the transaction and fully commit the participants. It is at this stage that the differences between an ordinary local transaction (*single-phase commit*) and a distributed transaction (two-phase commit) become apparent.

In the *first phase* of the two-phase commit, the transaction manager invokes `prepare()` on all of the resources participating in the transaction. This gives the participating resources an opportunity to save the current status of the transaction in their respective *transaction logs*. These transaction logs are used by the automatic failure recovery mechanism if something goes wrong before the transaction completes. Resources reply to the `prepare()` operation by sending back a vote on the outcome of the transaction. The alternatives available for the `Vote` response are:

- Return `VoteReadOnly` if the resource's data is not modified by the transaction. This would be the case, for example, if the interaction with the database only involved `SQL SELECT` statements.
- Return `VoteCommit` if the resource's data is written to stable storage by the transaction and the transaction is properly prepared. The response indicates that this particular participant is happy to proceed with committing the transaction.
- Return `VoteRollback` if this particular participant wants to abort the transaction for any reason.

The transaction manager waits for all of the votes to be returned before initiating the *second phase* of the two-phase commit.

The action taken by the transaction manager in the second phase of the two-phase commit depends upon the outcome of the voting by participants. If the consensus of the participants is to commit, the transaction manager proceeds by invoking `commit()` on all of the resources. However, if there are one or more negative outcomes, the transaction manager invokes `rollback()` on all participating resources.

The OTS operates on a model of presumed-abort. In other words, if a particular resource fails to receive the confirming `commit()` it will ultimately roll back the transaction (after a certain timeout).

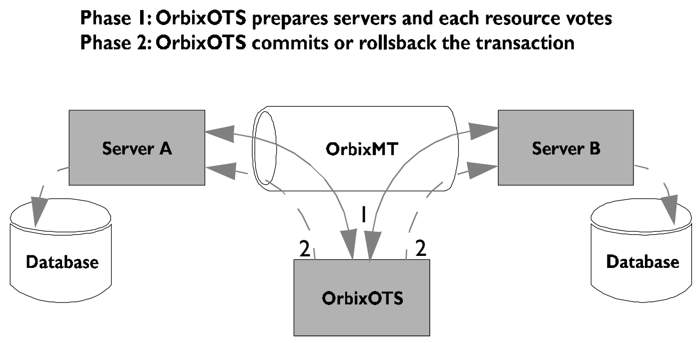

Phase 1: OrbixOTS prepares servers and each resource votes
Phase 2: OrbixOTS commits or rollsback the transaction
Server A
OrbixMT
Server B
Database
1
2
2
Database
OrbixOTS

Figure 7.9: *A Two-Phase Commit*

Recoverability and Log Files

Recoverability of transactions relies crucially on careful management of a transaction log file. Every recoverable OTS server is associated with such a log file, which stores the state of pending transactions. In the course of executing a two-phase commit, a recoverable server will not proceed to the commit phase until data needed for recovery has been physically written to the log file.

In the event of a server crash, the log file will enable the restarted server to resolve transactions pending before the crash.

Threading and Concurrency

The X/Open Distributed Transaction Processing (DTP) model does not directly address issues of threading and concurrency. It was developed originally to describe resources accessed from a single thread of control. This applies, for example, to the case where a process opens only a single connection to a database. In the original X/Open DTP model, there was no need to identify which database connection was being used at any one time, nor which thread of control was associated with a particular transaction.

In the meantime, as threading has become the norm across a wide range of applications, database vendors have added multithreading capability to their products. What this means is that a single process, which is a client of the database, may open multiple connections to the database and have multiple threads running, all of which are accessing the database simultaneously. Several transactions may be active at once. Each active transaction must be associated with *a database connection* and *a thread of control.*

In the case of multithreaded database access, the following convention is followed by the database vendors:

- Each database connection is uniquely associated with a particular thread of control.

  This implies that a particular database connection must be opened and closed by a particular thread. Moreover, the database connection must only be accessed via its associated thread and not by any other thread.
- Each transaction is uniquely associated with a database connection and, therefore, also uniquely associated with a particular thread of control.

> A transaction is begun when a database is accessed via a particular thread, and it remains associated with this thread for the duration of the transaction. All work on a particular transaction must therefore be carried out via its unique associated thread.

When a transaction manager, such as OrbixOTS, is used in multithreaded mode, it must be aware of this threading convention. That is why there is a final `boolean` argument to the method `OrbixOTS::Server::register_xa_rm()` to indicate to OrbixOTS whether the resource is multithreaded.

Programming in Multithreaded Mode

The developer who wants to use OrbixOTS with a resource in multithreaded mode has to carry out the following tasks:

- Tell OrbixOTS that the resource is concurrent.
- Configure the resource to run in concurrent mode.
- Choose a concurrency model for Orbix event processing.
- Set the OrbixOTS thread pool size.

Tell OrbixOTS that the Resource is Concurrent

The operation `OrbixOTS::Server::register_xa_rm()` takes an association flag as its final argument. This association flag should be set to `TRUE` if you want to use OrbixOTS in multithreaded mode. The important point is that by telling the OTS transaction manager it is being used in multithreaded mode, you ensure that it adopts the proper threading convention in its interaction with the database.

Configure the Resource

It is necessary to consult the documentation from your resource or database vendor for detailed instructions on setting up the resource to operate in concurrent mode. You must follow the instructions specifically relating to the use of an XA interface. Most of the hard work for the multithreading case must be done in the part of the application code that accesses the database. This work consists of implementing thread synchronization and following the directions of the database vendor in relation to concurrent database access.

Choose a Concurrency Model

Part of the support for multithreading in OrbixOTS is a feature that allows a developer to choose a concurrency model for the processing of incoming CORBA invocations. OrbixOTS uses a thread pool by default for processing incoming CORBA invocations. This thread pool can be operated according to one of three alternative threading models: `serializeRequestsAndTransactions`, `concurrent` or `serializeRequests`.

- `OrbixOTS::Server::serializeRequestsAndTransactions` (default) is used by single-threaded applications. This concurrency mode only allows one transaction and one request to be active in the server at a time. That is, once a transaction accesses a server, no other transaction can enter that server until that transaction is complete.
- `OrbixOTS::Server::concurrent` is used by multithreaded applications using a database in multithreaded mode.[5] This concurrency mode applies no serialization constraints. That is, requests that are part of different global transactions run concurrently in the server.
- `OrbixOTS::Server::serializeRequests` can be used by applications using a database in multithreaded mode, but it is not a very common option. This mode is a compromise between the two previous modes. Requests that are part of different global transactions are serialized as they arrive at the server.

The concurrency model is specified as an argument to the operation `OrbixOTS::Server::impl_is_ready()`.

Set the OrbixOTS Thread Pool Size

OrbixOTS uses a pool of threads to process incoming requests and this thread pool is activated once `OrbixOTS::Server::impl_is_ready()` is called. It is important for the scalability of your application to ensure that the size of the thread pool is large enough to cope with the volume of incoming requests.

The variables `OTS_TPOOL_LWM` and `OTS_TPOOL_HWM` are used to control the number of threads available in the OrbixOTS thread pool.

5. It is possible for an application to be multithreaded but use the database in single-threaded mode. This implies that only one of the threads is ever used to access the database. The appropriate concurrency mode for such a case is `serializeRequestsAndTransactions`.

The low water mark OTS_TPOOL_LWM defines the minimum number of active threads available in the thread pool. It should be set to a value roughly equal to the average number of concurrent requests pending on a server. The default value is 5.

The high water mark OTS_TPOOL_HWM defines the maximum number of threads that can be active in the thread pool at any time. It should be set to a value greater than the peak number of concurrent requests made on an OTS server. The default value is 10*OTS_TPOOL_LWM.

The number of active threads in the pool varies dynamically between the limits set by OTS_TPOOL_LWM (minimum) and OTS_TPOOL_HWM (maximum) depending upon the load.

These variables can be set either as environment variables in the environment of the server, or as configuration variables in the scope OrbixOTS of your Orbix configuration.

8

Security

This chapter presents the basic concepts of OrbixSSL security and shows you how to add security to the StockWatch demonstration.

OrbixSSL integrates Orbix and Secure Sockets Layer (SSL) security. Using OrbixSSL, distributed applications can transfer confidential data securely across a network.

Normal Orbix applications communicate using the CORBA standard Internet Inter-ORB Protocol (IIOP). This application-level protocol is layered above the transport-level protocol TCP/ IP.

OrbixSSL offers application developers the option of layering IIOP on top of SSL, which in turn is layered on top of TCP/IP.

All Orbix components, including the Orbix daemon and Orbix utilities, and all OrbixSSL applications can communicate using SSL. OrbixSSL imposes few requirements on administrators and programmers who want to support SSL communications in Orbix applications.

OrbixSSL Concepts

SSL provides *authentication*, *privacy*, and *integrity* for communications across TCP/IP connections:

- Authentication allows an application to verify the identity of another application with which it communicates.
- Privacy ensures that data transmitted between applications can not be eavesdropped on or understood by an intermediary.
- Integrity allows applications to detect whether data was modified during transmission.

This section provides a brief introduction to these features of SSL.

Authentication in SSL

SSL uses Rivest Shamir Adleman (RSA) public key cryptography for its authentication phase. In public key cryptography, each application has an associated public key and private key. Data encrypted with the public key can be decrypted only with the private key. Data encrypted with the private key can be decrypted only with the public key.

Public key cryptography allows an application to prove its identity by encoding data with its private key. Since no other application has access to this key, the encoded data must derive from the true application. Any application can confirm the content of the encoded data by decoding it with the application's public key.

Consider the example of two applications, a client and a server. The client connects to the server and wants to send some confidential data. Before sending the data, the client must ensure that it is connected to the required server and not to an impostor.

When the client connects to the server, it confirms the server identity using the following protocol:

1. The client asks the server to transmit the server's public key.
2. The server sends its public key to the client.
3. The client challenges the server to send a message encrypted with the server's private key.

4. The server sends a message to the client both in plaintext format and encrypted with the server's private key. The format of this message is such that it would be infeasibly difficult for an application other than the true server to generate.
5. The client decrypts the encrypted version of the message with the corresponding public key. The client compares this decoded message with the plaintext message to verify the identity of the server.

The protocol also allows the server to authenticate the client. Client authentication, which is supported by OrbixSSL, is also optional in SSL communications.

As any application can have a public and private key pair, the transfer of the public key in step two must be accompanied by additional information that proves the key is associated with the true server, and not some other application. For this reason, the key is transmitted as part of a certificate thereby contributing to the process of owner verification.

Certificates in SSL Authentication

The International Telecommunications Union (ITU) recommendation X.509 defines a standard format for certificates. SSL authentication uses X.509 certificates to transfer information about an application's public key.

An X.509 certificate includes the following data:

- The name of the entity identified by the certificate.
- The public key of the entity.
- The name of the certification authority that issued the certificate.

For more information on X.509 certificates, see the *OrbixSSL C++ Programmer's and Administrator's Guide*.

The role of a certificate is to match an entity name to a public key. A certification authority (CA) is a trusted authority that verifies the validity of the combination of entity name and public key in a certificate.

You have to specify trusted CAs in order to use OrbixSSL.

Privacy of SSL Communications

Immediately after authentication, an SSL client application sends an encoded data value, encrypted using the server's public key, to the server. This unique session encoded value is a key to a symmetric cryptographic algorithm.

A symmetric cryptographic algorithm is an algorithm in which a single key is used to encode and decode data. Once the server has received such a key from the client, all subsequent communications between the applications can be encoded using the agreed symmetric cryptographic algorithm.

Examples of symmetric cryptographic algorithms used to maintain privacy in SSL communications are the Data Encryption Standard (DES) and RC4.

Integrity of SSL Communications

The authentication and privacy features of SSL ensure that applications can exchange confidential data that cannot be understood by an intermediary. However, these features do not protect against the modification of encrypted messages transmitted between applications.

To detect if an application has received data modified by an intermediary, SSL adds a message authentication code (MAC) to each message. This code is computed by applying a function to the message content and the secret key used in the symmetric cryptographic algorithm.

An intermediary cannot fake the MAC for a message without knowing the secret key used to encrypt it. If the message is corrupted during transmission, the message content will not match the MAC. SSL automatically detects this error and rejects corrupted messages.

Validation of Certificates

Validation is a key step in the process of authentication where it is decided whether or not a received certificate can be considered trustworthy. OrbixSSL provides you with a number of options for certificate validation, allowing you to customize it to a greater or lesser degree. The basic options are as follows:

- *Default validation.* By default, OrbixSSL checks that a certificate has been signed by a CA that you trust (a list of CAs can be supplied in the configuration of OrbixSSL). This policy can be useful if you know that the application can trust all certificates signed by a particular CA or CAs.
- *Customized validation.* For more fine-grained validation, you have the option of adding code to your secure application to check the validity of certificates. OrbixSSL allows you to register a callback function that will be called during the authentication phase of connection establishment. The callback function is passed a copy of the peer certificate and can decide whether to accept or reject the connection based on the contents of this certificate. The default validation is still carried out by OrbixSSL.

Certificate Revocation List (CRL)

A CRL is a blacklist of certificates that are denied access to secure applications. Once a certificate is revoked it is denied access to secure applications. This requirement might arise, for example, if certain certificates were no longer considered trustworthy or if certificates lost the privilege of accessing the secure system.

The mechanism for revoking certificates is easy to administer. The certificate to be revoked is appended to a file containing the certificate revocation list. OrbixSSL automatically consults this file when a connection attempt is made. Connection attempts by clients using one of the revoked certificates will fail. No extra coding is required to enable this feature.

Installing OrbixSSL

OrbixSSL is delivered on a separate CD-ROM to the other Orbix components. OrbixSSL includes the following components:

- Link libraries and header files for C++ developers.
- Java classes for Java developers.
- Utility programs for creating and managing X.509 certificates.

OrbixSSL installation integrates SSL components with your existing Orbix installation. It adds SSL capabilities to your system without affecting existing applications.

Enabling SSL Security

There are number of steps involved in making an Orbix application secure:

1. Set up X.509 certificates for the applications. These can either be obtained from a public CA or generated by a private CA.
2. Add some code to your applications to enable SSL.
3. Configure which certificates the applications use to identify themselves and which certificates they accept.
4. Configure security for each of the Orbix components and services. This typically includes the Orbix daemon, the interface repository, the naming service, and the event service.

You also want to be sure that the runtime environment is set up so that your application picks up dynamic libraries from the correct location.

Extending the StockWatch Example for SSL

Example 4 of the StockWatch demonstration adds SSL capability to the StockWatch clients and servers. The demonstration configures security with the client authentication option enabled.

Making the Server Secure

When extending an application to use SSL security you must:

- Initialize the OrbixSSL library.
- Ensure OrbixSSL knows where the application's X.509 certificate file and private key are located.
- Ensure OrbixSSL knows the password associated with the private key.

- Define an OrbixSSL security policy. This instructs Orbix whether insecure client connections will be accepted, or whether the application will connect to insecure server applications.

These issues are dealt with in the following code sample.

```
      // C++
1     #include <IT_SSL.h>
      ...
      // Initialize the toolkit and check for any errors
      //
2     CORBA(Orbix).filterBadConnectAttempts(1);

3     if (OrbixSSL.init() == IT_SSL_SUCCESS) {
         cout << "Specifing secure, insecure servers and clients"
               << endl;
         //
         //  Associate a particular certificate with this application.
         //
4        if ( OrbixSSL.setPrivateKeyPassword("demopassword")
                  != IT_SSL_SUCCESS) {
            cout << "Error setting private key password" << endl;
            return 1;
         }

         if (OrbixSSL.setSecurityName("demos/demo_server")
                  != IT_SSL_SUCCESS) {
            cout << "Error setting security name" << endl;
            return 1;
         }

         //
         // set the server policy to connect to insecure servers and
         // accept insecure and secure connections
         //
5        if ( OrbixSSL.setInvocationPolicy(
               IT_INSECURE_ACCEPT | IT_SECURE_ACCEPT ) != IT_SSL_SUCCESS
      ) {
            cout << "Error setting security policy" << endl;
            return 1;
         }
      }
```

The code is explained as follows:

1. Include the header file `IT_SSL.h` to gain access to the declarations for the OrbixSSL programming interface.
2. If a client connection attempt is rejected, this would normally cause an exception to be raised on the server side during event processing (`CORBA::BOA::impl_is_ready()` or `CORBA::BOA::processEvents()`).

   In a secure environment, however, connection attempts are routinely rejected when the SSL authentication step fails. In most cases, it would be inconvenient to handle these exceptions, therefore the option `filterBadConnectAttempts()` is set to true to suppress them. This behaviour can also be specified by an administrator via the `orbixssl.cfg` file.
3. Basic OrbixSSL initialization call. This initialization call is invoked on the `OrbixSSL` object, which is a predefined instance of class `IT_SSL` constructed in the SSL libraries.
4. Specify the application's X.509 certificate, and its associated password. This example application uses a demonstration certificate which is located in the OrbixSSL demonstration certificate repository.

   The function `setSecurityName()` specifies the name of the file containing both the X.509 certificate and its associated private key (the private key is stored in encrypted form). The name of this file is specified relative to a root certificate directory, `IT_CERTIFICATE_PATH`, specified in the Orbix SSL configuration file.
5. A variety of client and server policies can be specified as arguments to `setInvocationPolicy()`. In this example, the chosen invocation policy allows both secure and insecure applications to connect to the StockWatch application.

Making the Client Secure

A client can be made secure using similar code to that given in the preceding section. The only difference is that the client specifies its own X.509 certificate in the call to `setSecurityName()` and passes the corresponding password to `setPrivateKeyPassword()`. These calls are only necessary, however, if client authentication is enabled.

The client is likely to choose a different set of options in the call to `setInvocationPolicy()`. A typical client policy is `IT_SECURE_CONNECT`, specifying that the client always attempts to use SSL security when it opens a new connection to a server.

In the case of the StockWatch clients, for example `clients/cxx/clientots_ssl.cxx`, extra code is used to perform customized validation of the server's X.509 certificate. Detailed explanation of this feature is beyond the scope of this chapter.

Administration of OrbixSSL

An understanding of OrbixSSL administration is essential in order to run secure applications. Consult the *OrbixSSL C++ Programmer's and Administrator's Guide* for a full discussion of the configuration issues surrounding security before running an SSL demonstration.

The main configuration issues that you should be aware of are:

- Selection of a CA.
- Selecting an appropriate security policy. For example, should all communications be secure or should a server allow secure and insecure clients to attach to it. Likewise, should a client insist on only connecting to secure servers or should it selectively open secure and insecure connections.
- Specifying the security policy of the Orbix daemon process.
- Enabling client authentication by OrbixSSL servers.
- Specifying the list of supported message encryption algorithms in order of preference.

Setting up a secure Orbix system includes configuring the following:

- A secure Orbix daemon.
- A secure interface repository (IFR).
- A secure naming service.
- A secure event service.
- A secure management service.

9

Load Balancing

Load balancing extensions to the naming service are introduced and a complete programming example is presented.

As your application scales up from a test to a production environment, more and more clients will cause the load to become high on a single server. However, you can design your server so that as you monitor its activity, an administrator can replicate it to facilitate load balancing. Requests can then be routed according to the load on the server process. Additional client requests can be taken care of automatically with minimal delay.

To optimize system throughput, Orbix uses the object group feature of OrbixNames first to register pools of servers and then to distribute requests according to the load on the server processes.

Load balancing is invisible to the client. When a client requests an object from OrbixNames, OrbixNames selects, from a group of objects, an object to resolve that request. The algorithm for choosing an object is set when the object group is initially created in the naming service, and can be either random or round-robin.

Example 5 introduced here is simply a modification of example 1. The tasks to consider in the revision are as follows:

- Creation of an object group in the naming service.
- Introduction of a mechanism whereby the server can be replicated. This is done via the OrbixNames object groups feature and can be implemented by modifying the server `main()` function.

Object Groups in OrbixNames

OrbixNames gives application developers the opportunity to hide object location details from client applications. The CORBA services naming specification describes a model whereby a *name* maps to an associated object. OrbixNames optionally extends this model to allow a name to map to a group of objects (see Figure 9.1).

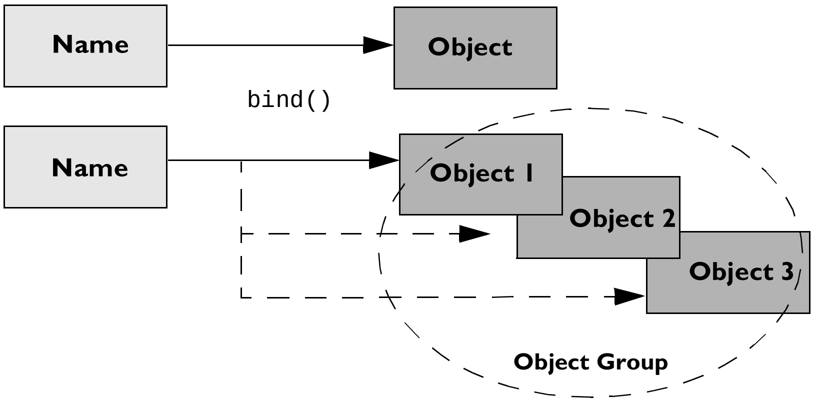


Figure 9.1: *Object Group*

In addition to implementing the `CosNaming` IDL module, OrbixNames also supports a new IDL module called `LoadBalancing` as follows:

```
module LoadBalancing {
   exception no_such_member{};
   exception duplicate_member{};
   exception duplicate_group{};
   exception no_such_group{};
   typedef string memberId;
   typedef sequence<memberId> memberIdList;
   struct member{
      Object obj;
      memberId id;
   };
   interface ObjectGroup{
      readonly attribute string id;
      Object pick();
      void addMember(in member mem) raises (duplicate_member);
      void removeMember(in memberId id) raises (no_such_member);
      Object getMember(in memberId id) raises (no_such_member);

      memberIdList members();
      void destroy();
   };
   typedef string groupId;
   typedef sequence<groupId> groupList;

   interface RandomObjectGroup : ObjectGroup {};
   interface RoundRobinObjectGroup : ObjectGroup {};
   interface ObjectGroupFactory{
      RoundRobinObjectGroup createRoundRobin(in groupId id)
              raises (duplicate_group);
      RandomObjectGroup createRandom(in groupId id)
              raises (duplicate_group);
      ObjectGroup findGroup(in groupId id) raises (no_such_group);
      groupList rr_groups();
      groupList random_groups();
   };
};
```

Typically, an instance of object group is *bound* to a name in OrbixNames. Application objects can be added to, or removed from, this object group.

Internally, the implementation of `CosNaming::NamingContext::resolve()` has been modified to check whether the name being resolved maps to an object of type `ObjectGroup`. If it does not, then the object is simply returned as normal. However, if the object is of type `ObjectGroup`, then the operation `ObjectGroup::pick()` is called. This operation *picks* and returns a member of the group.

Object groups offer basic load balancing functionality to developers (see Figure 9.2). They are invisible to client side applications which simply call `CosNaming::NamingContext::resolve()`, as before. However, server code modifications are required.

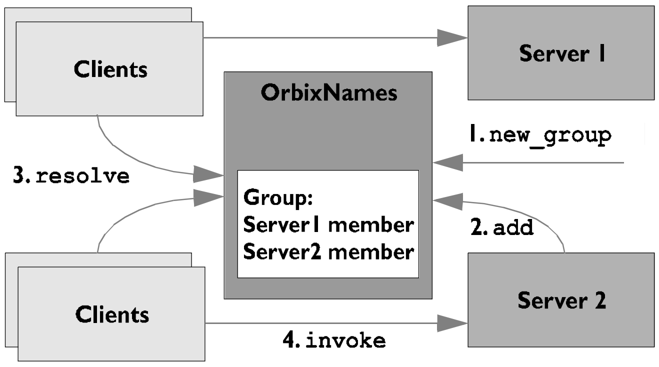


Figure 9.2: *Naming Service and Load Balancing*

OrbixNames load balancing functionality is summarized as follows:

1. The administrator creates an object group and binds it to a name typically by using the new_group utility.
2. Servers *add* members to the group.
3. Clients *resolve* names in the usual manner. However, because the name being resolved maps to an object group, one member is chosen for that client.
4. Clients invoke a member of the object group.

Creating an Object Group in OrbixNames

The simplest way to create an object group is to use the new group utility as follows:

```
new_group oracle example5.oracle -round_robin
```

This creates a new round-robin load balancing group with group ID `oracle`. It binds the group to the name `oracle` in the `example5` naming context which is in the `root` context.

It is also possible to create an object group programmatically. Using the interfaces given in the IDL `LoadBalancing` module you can write code directly in your server that creates an object group as needed. This is the approach taken in example 5 of the StockWatch demonstration.

Modifications to the Client

The client must be linked with the stub code for `LoadBalancing.idl`. If the stub code is not already part of the client, the client has to be relinked.

Usually no programming modifications are required on the client side. The syntax used to access a load-balancing names server is identical to the syntax used to access an ordinary names server. In both cases the client invokes the method `CosNaming::NamingContext::resolve()` to look up a name and obtain an object reference.

However, the semantic difference between the load-balancing and non-load-balancing cases may have an impact on some clients. If a client assumes that the naming service returns a reference to the same server instance every time, and if the client relies on this assumption in a non-trivial way, then the client might be affected by the introduction of load balancing. This is a general semantic consideration which can affect any load balanced system.

Modifications to the Server

Previously the application simply bound a name to a single `StockWatch` object. Using object groups the server should *add* a member to an object group instead.

```
// serverOracle.cc

int main(int argc, char **argv) {

      if (!argv[1] || !argv[2] || argv[3])
         syntax_error();

      char servername [64];
      sprintf(serverName, "%s/%s_%s", DEMO, "oracle", argv[3]);
      CosNaming::Name_var tmpName;
```

```
    CORBA::ORB_var orb = CORBA::ORB_init(argc,argv,"Orbix");

    oracle_backend orac (argv[1], argv[2]);
    try {
          CORBA::Orbix.setServerName(serverName);

          // Instantiate my instance of StockWatch
          StockWatch_i swatch(&orac);

       cout << "Resolve to OrbixNames" << endl;
       CosNaming::NamingContext_var rootCtx;
       CORBA::Object_var objectV =
          orb->resolve_initial_references(
                                             "NameService"
                                          );
       rootCtx = CosNaming::NamingContext::_narrow(objectV);
       if (CORBA::is_nil(rootCtx)) {
          cerr << "Narrow failed for NamingContext" << endl;
          exit(1);
       }
       cout << "resolved successfully to OrbixNames" << endl;

       // Construct a Name
       tmpName = new CosNaming::Name(2);
       tmpName->length(2);
       tmpName[0].id = CORBA::string_dup("example5");
       tmpName[0].kind = CORBA::string_dup("");
       tmpName[1].id = CORBA::string_dup("oracle");
       tmpName[1].kind = CORBA::string_dup("");
       try {
             CORBA::Object_ptr tmp =
1               rootCtx->resolve_object_group(tmpName);

             LoadBalancing::ObjectGroup_var og =
                LoadBalancing::ObjectGroup::_narrow(tmp);

             LoadBalancing::member mem;
             mem.obj = CORBA::Object::_duplicate(&swatch);
             mem.id = CORBA::string_dup(argv[3]);
2            og->addMember(mem);

       } catch (LoadBalancing::duplicate_member) {
                cout << "Already a member of" << tmpName << endl;
```

```
                     cout << "Ignoring... " << endl;
            }
         // Now wait for incoming invocations.
3        CORBA::Orbix.impl_is_ready(serverName,TIMEOUT);
      } catch(...)
```

The code is explained as follows:

1. Resolve an object group which should be bound to the name `example5.oracle`. This object group was created and bound using the `new_group` utility.
2. Use the `addMember()` function to add your server's object to the naming service's object group. This function takes a `LoadBalancing::member` structure as an argument. It includes the object reference you want associated with the object group (that is, `swatch`) and an `id` for the object. This example uses a simple `id` (`argv[3]`) to represent this server instance.
3. The `impl_is_ready()` function lets Orbix know that the server is ready to service client requests. In this case the Orbix server name is `example5/oracle_`*id*, where *id* is taken from `argv[3]`.

   Object group members reside in different Orbix servers. This means that each server needs to be individually registered with the Orbix Implementation Repository. For example:

   ```
   putit example5/oracle_server_1 ...
   putit example5/oracle_server_2 ...
   ```

Enabling Load Balancing

The naming service must be configured to enable load balancing before you can run this application. Load balancing is enabled by supplying the `-l` command line switch. Register the naming service with the Orbix daemon as follows:

`putit -h` *NSHost* `NS "`*IONARoot*`/bin/ns -l"`

where *NSHost* is the host where the naming service is run. This ensures that load balancing is enabled whenever the naming service is started automatically.

Index