

Oracle8i

CORBA Developer's Guide

Release 3 (8.1.7)

July 2000

Part No. A83722-01

ORACLE®

CORBA Developer's Guide, Release 3 (8.1.7)

Part No. A83722-01

Release 3 (8.1.7)

Copyright © 1996, 2000, Oracle Corporation. All rights reserved.

Primary Authors: Sheryl Maring

Contributors: Tim Smith, Ellen Barnes, Matthieu Devin, Steve Harris, Hal Hildebrand, Susan Kraft, Thomas Kurian, Wendy Liau, Angie Long, Sastry Malladi, John O'Duinn, Jeff Schafer, Aniruddha Thakur

The Programs (which include both the software and documentation) contain proprietary information of Oracle Corporation; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Oracle Corporation.

If the Programs are delivered to the U.S. Government or anyone licensing or using the programs on behalf of the U.S. Government, the following notice is applicable:

Restricted Rights Notice Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and Oracle Corporation disclaims liability for any damages caused by such use of the Programs.

Oracle is a registered trademark, and Oracle products mentioned herein are trademarks or registered trademarks of Oracle Corporation. All other company or product names mentioned are used for identification purposes only and may be trademarks of their respective owners.

Contents

Send Us Your Comments	xi
Preface.....	xiii
1 Overview	
Prerequisite Reading.....	1-2
Terminology.....	1-2
About CORBA.....	1-3
CORBA Features.....	1-5
About the ORB.....	1-6
Using JNDI and IIOP.....	1-7
IIOP.....	1-7
For More Information.....	1-8
Books.....	1-8
URLs.....	1-9
2 Getting Started	
A First CORBA Application.....	2-2
Writing Interfaces in IDL.....	2-3
Generate Stubs and Skeletons.....	2-4
Write the Server Object Implementation.....	2-6
Write the Client Code.....	2-8
Compiling the Java Source.....	2-11
Load the Classes into the Database.....	2-11

Publish the Object Name	2-12
Run the Example.....	2-13
The Interface Definition Language (IDL).....	2-14
Using IDL.....	2-15
IDL Types.....	2-19
Exceptions	2-22
Getting by Without IDL.....	2-23
Activating ORBs and Server Objects	2-24
Client Side	2-24
Server Side	2-24
About Object Activation	2-24
CORBA Interceptors.....	2-25
Debugging Techniques.....	2-25
Using a Debug Agent for Debugging Server Applications	2-26

3 Configuring IIOP Applications

Overview.....	3-2
Oracle8i Typical or Minimal Installation.....	3-3
Oracle8i Custom Installation.....	3-4
Manual Install and Configuration	3-8
Configuring Through Tools	3-8
Configuring Through Editing Initialization Files	3-9
Advanced Configuration Options.....	3-11
Database Listeners and Dispatchers	3-11
Dynamic Listener Endpoint Registration.....	3-15
Direct Dispatcher Connection.....	3-16
Configuring SSL for EJB and CORBA.....	3-17

4 JNDI Connections and Session IIOP Service

JNDI Connection Basics	4-2
The Name Space.....	4-3
Execution Rights to Database Objects.....	4-4
URL Syntax.....	4-5
URL Components and Classes.....	4-6
Using JNDI to Access Bound Objects.....	4-7

Importing JNDI Support Classes.....	4-9
Retrieving the JNDI InitialContext.....	4-9
Session IIOP Service	4-13
Session IIOP Service Overview	4-13
Session Management.....	4-15
Service Context Class	4-16
Session Context Class.....	4-17
Session Management Scenarios.....	4-18
Setting Session Timeout.....	4-26
Retrieving JServer Version Number	4-28
Activating In-Session CORBA Objects From Non-IIOP Presentations.....	4-28
Accessing CORBA Objects Without JNDI	4-29
Retrieving the NameService Initial Reference.....	4-29
Retrieving Initial References from ORBDefaultInitRef	4-33

5 Advanced CORBA Programming

Using SQLJ.....	5-2
Running the SQLJ Translator	5-2
A Complete SQLJ Example	5-3
Implementing CORBA Callbacks	5-3
IDL	5-4
Client Code	5-4
Callback Server Implementation	5-5
Callback Client-Server Implementation	5-5
Retrieving Interfaces With The IFR	5-6
Publishing the IDL Interface	5-7
Retrieving Interfaces Implicitly	5-7
Retrieving Interfaces Explicitly.....	5-8
Using the CORBA Tie Mechanism.....	5-10
Migrating from JDK 1.1 to Java 2.....	5-11
Invoking CORBA Objects From Applets.....	5-16
Using Signed JAR Files to Conform to Sandbox Security	5-16
Performing Object Lookup in Applets	5-16
Modifying HTML for Applets that Access CORBA Objects	5-18
Interoperability with non-Oracle ORBs.....	5-21

Java Client using Oracle ORB	5-22
Java Client using non-Oracle ORB	5-22
C++ Client Interoperability	5-23
IIOP Transport Protocol.....	5-25

6 IIOP Security

Overview	6-2
Data Integrity	6-3
Using the Secure Socket Layer	6-3
SSL Version Negotiation.....	6-4
Authentication	6-5
Client-side Authentication	6-6
Using JNDI for Authentication.....	6-8
Providing Username and Password for Client-Side Authentication.....	6-9
Using Certificates for Client Authentication	6-13
AuroraCertificateManager Class.....	6-16
Server-Side Authentication	6-20
Authorization	6-26
Setting up Trust Points.....	6-27
Parsing through the Server's Certificate Chain.....	6-27
AuroraCurrent Class.....	6-28

7 Transaction Handling

Transaction Overview	7-2
Global and Local Transactions.....	7-3
Demarcating Transactions.....	7-3
Transaction Context Propagation.....	7-4
Enlisting Resources.....	7-5
Two-Phase Commit	7-5
JTA Limitations	7-6
JTA Server-Side Demarcation	7-7
JTA Client-Side Demarcation	7-9
Configuring Two-Phase Commit Engine	7-15
Creating DataSource Objects Dynamically	7-17
Setting the Transaction Timeout	7-18

Java Transaction Service	7-19
JTS Client-Side Demarcation.....	7-20
JTS Server-Side Demarcation	7-22
JTS Limitations.....	7-23
Transaction Service Interfaces	7-24
TransactionService.....	7-25
Using The Java Transaction Service.....	7-25
For More Information on JTS	7-29
JDBC Restrictions	7-30

A Example Code: CORBA

Basic Example	A-2
README.....	A-2
Bank.IDL	A-2
Server.....	A-3
Client.java	A-5
IFR Example	A-6
Bank.IDL	A-6
Server.....	A-6
Client	A-9
Callback Example	A-16
README.....	A-16
IDL Files.....	A-20
Server.....	A-20
Client	A-21
TIE Example	A-22
README.....	A-22
Hello.IDL	A-25
Server Code - HelloImpl.java.....	A-25
Client.java	A-25
Pure CORBA Client	A-26
README.....	A-26
Bank.IDL	A-30
Server Code	A-30
Client.java.....	A-32

JTA Examples	A-34
Single-Phase Commit JTA Transaction Example	A-34
Employee.IDL.....	A-34
Client.java	A-35
EmployeeServer.sqlj	A-38
Two-Phase Commit JTA Transaction Example	A-40
Employee.IDL.....	A-40
Client.java	A-40
Server	A-42
JTS Transaction Example	A-48
README.....	A-48
Employee.IDL.....	A-51
Client.java	A-52
Server	A-53
SSL Examples	A-54
Client-Side Authentication	A-54
README.....	A-54
Hello.IDL.....	A-58
Client.java	A-58
Server	A-59
Server-Side Authentication	A-59
README.....	A-60
Hello.IDL.....	A-63
Client.java	A-63
Server	A-66
Session Example	A-66
README.....	A-66
Hello.IDL.....	A-69
Client.java	A-69
Server	A-71
Applet Example	A-71
JDK and Jinitiator Applets	A-71
README.....	A-71
HTML for JDK 1.1	A-72
HTML for JDK 1.2.....	A-73

HTML for Oracle JInitiator.....	A-73
Applet Client.....	A-74
Visigenic Applet.....	A-75
README.....	A-75
HTML for Visigenic Client Applet.....	A-76
Visigenic Client Applet.....	A-76

B Comparing the Oracle8i JServer and VisiBroker VBJ ORBs

Object References Have Session Lifetimes	B-2
The Database Server is the Implementation Mainline	B-3
Server Object Implementations are Deployed by Loading and Publishing.....	B-3
Implementation by Inheritance is Nearly Identical	B-3
Implementation by Delegation is Different.....	B-3
Clients Look Up Object Names with JNDI	B-5
No Interface or Implementation Repository	B-5
The Bank Example in Aurora and VBJ	B-5
The Bank IDL Module.....	B-6
Aurora Client.....	B-6
VBJ Client.....	B-7
Aurora Account Implementation	B-8
VBJ Account Implementation	B-8
Aurora Account Manager Implementation	B-8
VBJ Account Manager Implementation	B-10
VBJ Server Mainline	B-10

C Abbreviations and Acronyms

Send Us Your Comments

CORBA Developer's Guide, Release 3 (8.1.7)

Part No. A83722-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail — jpgcomnt@us.oracle.com
- FAX - 650-506-7225. Attn: Java Platform Group, Information Development Manager
- Postal service:
Oracle Corporation
Information Development Manager
500 Oracle Parkway, Mailstop 4op978
Redwood Shores, CA 94065
USA

Please indicate if you would like a reply.

If you have problems with the software, please contact your local Oracle World Wide Support Center.

Preface

This guide gets you started building CORBA applications for Oracle8i. It includes many code examples to help you develop your application.

Who Should Read This Guide?

Anyone developing server-side CORBA applications for Oracle8i will benefit from reading this guide. Written especially for programmers, it will also be of value to architects, systems analysts, project managers, and others interested in network-centric database applications. To use this guide effectively, you must have a working knowledge of Java and Oracle8i. This guide assumes that you have some familiarity with CORBA. See ["Suggested Reading"](#) on page xv for more information on CORBA concepts.

How This Guide Is Organized

This guide consists of the following chapters and appendices:

[Chapter 1, "Overview"](#), presents a brief overview of the CORBA development model from an Oracle8i perspective.

[Chapter 2, "Getting Started"](#), describes techniques for developing CORBA server objects that run in the Oracle8i data server.

[Chapter 3, "Configuring IIOP Applications"](#) describes how to configure for your CORBA applications.

[Chapter 4, "JNDI Connections and Session IIOP Service"](#) describes how to use JNDI and sessions within your CORBA applications.

[Chapter 5, "Advanced CORBA Programming"](#) details how to program your CORBA application beyond the simple example presented in Chapter 2.

[Chapter 6, "IIOP Security"](#) covers how to implement security within your CORBA application.

[Chapter 7, "Transaction Handling"](#), documents the transaction interfaces that you can use when developing your CORBA applications.

[Appendix A, "Example Code: CORBA"](#), includes examples of CORBA applications. Each example contains both the Java and IDL source code.

[Appendix B, "Comparing the Oracle8i JServer and VisiBroker VBJ ORBs"](#), discusses some of the fundamental differences between developing CORBA applications for VisiBroker and the Oracle8i JServer.

[Appendix C, "Abbreviations and Acronyms"](#), supplies a list of acronyms.

Notational Conventions

This guide follows these conventions:

Italic Italic font denotes terms being defined for the first time, words being emphasized, error messages, and book titles.

`Courier` Courier font denotes Java program names, file names, path names, and Internet addresses.

Java code examples follow these conventions:

`{ }` Braces enclose a block of statements.

`//` A double slash begins a single-line comment, which extends to the end of a line.

`/* */` A slash-asterisk and an asterisk-slash delimit a multi-line comment, which can span multiple lines.

`...` An ellipsis shows that statements or clauses irrelevant to the discussion were left out.

lower case Lower case is used for keywords and for one-word names of variables, methods, and packages.

UPPER CASE Upper case is used for names of constants (static final variables) and for names of supplied classes that map to built-in SQL datatypes.

Mixed Case Mixed case is used for names of classes and interfaces and for multi-word names of variables, methods, and packages. The names of classes and interfaces begin with an upper-case letter. In all multi-word names, the second and succeeding words also begin with an upper-case letter.

Suggested Reading

Programming with VisiBroker, by D. Pedrick et al. (John Wiley and Sons, 1998) provides a good introduction to CORBA development from the VisiBroker point of view.

Core Java by Cornell & Horstmann, second edition, Volume II (Prentice-Hall, 1997) has good presentations of several Java concepts relevant to EJB. For example, this book documents the Remote Method Invocation (RMI) interface.

Online Sources

There are many useful online sources of information about Java. For example, you can view or download guides and tutorials from the Sun Microsystems home page on the Web:

<http://www.sun.com>

Another popular Java Web site is:

<http://www.gamelan.com>

For Java API documentation, see:

<http://www.javasoft.com>

Related Publications

Occasionally, this guide refers you to the following Oracle publications for more information:

Oracle8i Application Developer's Guide - Fundamentals

Oracle8i Java Developer's Guide

Oracle8i JDBC Developer's Guide and Reference

Oracle8i SQL Reference

Oracle8i SQLJ Developer's Guide and Reference

This chapter gives you a general picture of distributed object development in the Oracle8i JServer. As with the more specific chapters that follow, this overview focuses on the aspects of CORBA development that are particular to JServer, giving a brief general description of the standard CORBA development model.

This chapter covers the following topics:

- [Prerequisite Reading](#)
- [Terminology](#)
- [About CORBA](#)
- [Using JNDI and IIOP](#)
- [For More Information](#)

Prerequisite Reading

Before consulting this Guide, you should read the *Oracle8i Java Developer's Guide*, which gives you the technical background information necessary to understand Java in the database server. As well as a comprehensive discussion of the advantages of the JServer implementation for enterprise application development, it explains the fundamentals of the JServer Java virtual machine and gives a technical overview of the tools that JServer provides.

In addition, the *Oracle8i Java Developer's Guide* describes the strategic advantages of the distributed component development model that is implemented by CORBA.

Terminology

This section defines some of the basic terms used in this chapter. See also [Appendix C, "Abbreviations and Acronyms"](#) for a list of common acronyms used in Java and distributed object computing.

client

A client is an object, an application, or an applet that makes a request of a server object. Remember that a client need not be a Java application running on a workstation or a network computer, nor an applet downloaded by a web browser. A server object can be a client of another server object. "Client" refers to a role in a requestor/server relationship, not to a physical location or a type of computer system.

marshalling

In distributed object computing, marshalling refers to the process by which the ORB passes requests and data between clients and server objects.

object adapter

Each CORBA ORB implements an object adapter (OA), which is the interface between the ORB and the message-passing objects. CORBA 2.0 specifies that a basic object adapter (BOA) must exist, but most of the details of its interface are left up to individual CORBA vendors. Future CORBA standards will require a vendor-neutral *portable object adapter* (POA). Oracle intends to support a POA in a future release.

request

A request is a method invocation. Other names sometimes used in its stead are *method call* and *message*.

server object

A CORBA server object is a Java object activated by the server, typically on a first request from a client.

session

A *session* always means a database session. Although it is conceptually the same kind of session as that established when a tool such as SQL*Plus connects to Oracle, there are differences in the CORBA case, as follows:

- You establish the database session using the IIOP protocol; you establish a SQL*Plus session using the Net8 TTC protocol.
- A Java virtual machine (JVM) that runs in the database server controls an IIOP session.

Note: To use CORBA with Oracle8i, you must configure the database so that the listener can recognize incoming IIOP requests, in addition to TTC requests. DBAs and system administrators should see the [Chapter 3, "Configuring IIOP Applications"](#) for information on setting up the database and the listener to accept incoming IIOP requests.

See "[Session IIOP Service](#)" on page 4-13 for more information about sessions.

About CORBA

CORBA stands for *Common Object Request Broker Architecture*. What is *common* about CORBA is that it integrates ideas from several of the original proposers. CORBA did not just follow the lead of a single large corporation, and it is very deliberately vendor neutral. The CORBA *architecture* specifies a software component, a *broker*, that mediates and directs *requests* to *objects* that are distributed across a network (or several networks), which might have been written in a different language from that of the requestor, and which might be (and in fact, usually are) running on a completely different hardware architecture from that of the requestor.

Note: This section provides a short introduction to CORBA, and should give you some idea of how you typically use CORBA in the Oracle8i server environment. Providing a complete introduction to CORBA is beyond the scope of this Guide. See the references in ["For More Information"](#) on page 1-8 for suggested further reading. This first section gives a very high-level overview of CORBA itself.

CORBA enables your application to tie together components from various sources. Also, and unlike a typical client/server application, a CORBA application is not inherently synchronous. It is not necessarily typical that a CORBA requestor (a client) invokes a method on a server component and waits for a result. Using asynchronous method invocations, event interfaces and callbacks from server object to the client ORB, you can construct elaborate applications that link together many interacting objects and that access one or many data sources and other resources under transactional control. .

CORBA offers a well-supported international standard for cross-platform, cross-language development. CORBA supports cross-language development by specifying a neutral language, Interface Definition Language (IDL), in which you develop specifications for the interfaces that the application objects expose.

CORBA supports cross-platform development by specifying a transport mechanism, IIOP, that allows different operating systems running on very different hardware to interoperate. IIOP supplies a common "software" bus that, together with an ORB running on each system, makes data and request transfer transparent to the application developer.

Although the CORBA standard was developed and promulgated just before the advent of Java and is a standard focused on component development in a heterogeneous application development environment, incorporating systems and languages of varying age and sophistication, it is perfectly possible to develop CORBA applications solely in Java. CORBA and Java are a good match.

For CORBA developers, JServer offers the following services and tools:

- a Java Transaction Service (JTS) interface to the OMG Object Transaction Service (OTS)
- a CosNaming implementation used when publishing objects to an Oracle8i database, retrieving the object references, and activating objects
- a version of the IIOP protocol that supports the JServer session-based ORB, which is compatible with standard IIOP

- a wide range of tools, which assist in developing CORBA applications, that do the following:
 - load Java classes and resource files to the database
 - drop loaded classes
 - publish objects to the CosNaming service
 - manage the session name space

CORBA Features

CORBA achieves its flexibility in several ways:

- It specifies an *interface description language* (IDL) that allows you to specify the interfaces to objects. IDL object interfaces describe, among other things:
 - The data that the object makes public.
 - The operations that the object can respond to, including the complete signature of the operation. CORBA operations map to Java methods, and the IDL operation parameter types map to Java datatypes.
 - Exceptions that the object can throw. IDL exceptions also map to Java exceptions, and the mapping is very direct.

CORBA provides bindings for many languages, including both non-object languages such as COBOL and C, and object-oriented languages such as Smalltalk and Java.
- All CORBA implementations provide an *object request broker* (ORB), that handles the routing of object requests in a way that is largely transparent to the application developer. For example, requests (method invocations) on remote objects that appear in the client code look just like local method invocations. The remote call functionality, including marshalling of parameter and return data, is performed for the programmer by the ORB.
- CORBA specifies a network protocol, the *Internet Inter-ORB Protocol* (IIOP), that provides for transmission of ORB requests and data over a widely-available transport protocol: TCP/IP, the Internet standard.
- A set of fully-specified *services* eases the burden of application development by making it unnecessary for the developer to constantly reinvent the wheel. Among these services are:
 - Naming. One or more services that let you resolve names that are bound to CORBA server objects.

- Transactions. Services that let you manage transaction control of data resources in a flexible and portable way.
- Events.

CORBA specifies over 12 services. Most of these are not yet implemented by CORBA ORB vendors.

The remainder of this section introduces some of the essential building blocks of an Oracle8i JServer CORBA application. These include:

- the ORB—how to talk to remote objects
- IDL—how to write a portable interface
- the naming service (and JNDI)—how to locate a persistent object
- object adapters—how to register a transient object

Note: The Java code examples in this chapter are available on line. You can study the complete examples (see [Appendix A, "Example Code: CORBA"](#)), compile and run them, and then modify them for your own use. All examples are installed in the `$ORACLE_HOME/javavm/demo/demo.zip` file.

About the ORB

The object request broker, or ORB, is the fundamental part of a CORBA implementation. The ORB makes it possible for a client to send messages to a server, and the server to return values to the client. The ORB handles all communication between a client and a server object.

The JServer ORB is based on code from Inprise's VisiBroker 3.4 for Java. The ORB that executes on the server side has been slightly modified from the VisiBroker code, to accommodate the different Oracle8i object location and activation model. The client-side ORB has been changed very little.

Note: The VisiBroker ORB functionality supplied with JServer is only licensed for accessing Oracle8i servers.

In some CORBA implementations, the application programmer and the server object developer must be aware of the details of how the ORB is activated on the client and the server, and they must include code in their objects to start up the

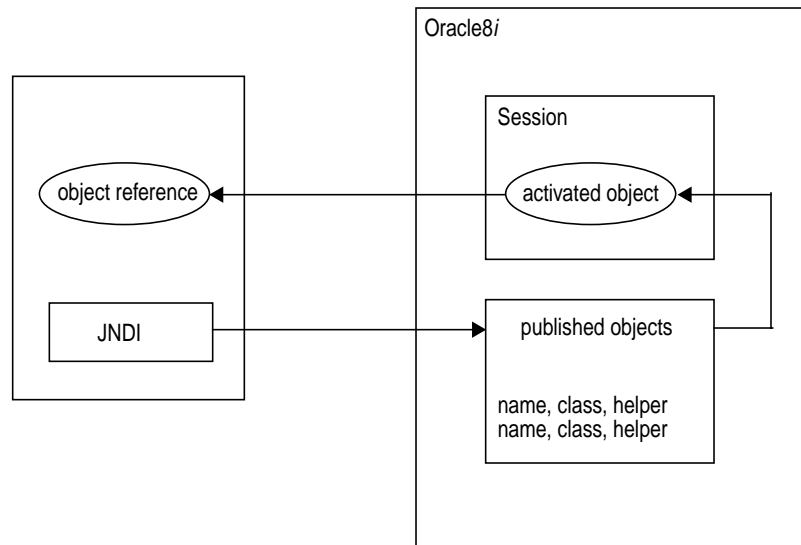
ORBs and activate objects. The Oracle8i ORB, on the other hand, makes these details largely transparent to the application developer. As you will see from the Java code examples later in this chapter and in Appendix A, only in certain circumstances does the developer need to control the ORB directly. These occur, for example, when coding callback mechanisms, or when there is a need to register transient objects with the basic object adapter.

Using JNDI and IIOP

You publish CORBA objects in the Oracle database using the OMG CosNaming service. In addition, you can access these objects using Oracle's JNDI interface to CosNaming.

Figure 1-1 shows, in a schematic way, how applications access remote objects published in the database using JNDI.

Figure 1-1 Remote Object Access



IIOP

Oracle8i offers a Java interpreter for the IIOP protocol. Oracle embeds a pure Java ORB of a major CORBA vendor (VisiBroker for Java version 3.4 by Inprise) and repackaging the Visigenic Java IIOP interpreter to run in the database.

Because Oracle8i is a highly scalable server, only the essential components of the interpreter are necessary—namely, a set of Java classes that do the following:

- decode the IIOP protocol
- find or activate the relevant Java object
- invoke the method the IIOP message specifies
- write the IIOP reply back to the client

Oracle8i does not use the ORB scheduling facilities. The Oracle multi-threaded server performs the dispatching, enabling the server to process IIOP messages efficiently and in a highly scalable manner.

On top of this infrastructure, Oracle8i implements the EJB and CORBA programming models.

For More Information

This section lists some resources that you can access to get more information about CORBA and about CORBA application development using Java.

Books

The ORB and some of the CORBA services supplied with Oracle8i JServer are based on VisiBroker for Java code licensed from Inprise. *Programming with VisiBroker*, by D. Pedrick et al. (John Wiley and Sons, 1998), provides both an introduction to CORBA development from the VisiBroker point of view and an in-depth look at the VisiBroker CORBA environment.

Client/Server Programming with Java and CORBA, by R. Orfali and D. Harkey (John Wiley and Sons, 1998), covers CORBA development in Java. This book also uses the VisiBroker implementation for its examples.

You should be aware that the examples published in both of these books require some modification to run in the Oracle8i ORB. It is better to start off using the examples in the Appendices to this Guide, which are more extensive than the examples in the books cited and demonstrate all the features of Oracle8i CORBA. See also [Appendix B, "Comparing the Oracle8i JServer and VisiBroker VBJ ORBs"](#) for a discussion of the major differences between VisiBroker for Java and the Oracle8i implementation.

URLs

You can download specifications for CORBA 2.0 and for CORBA services from links available at the following web site:

<http://www.omg.org/library/downinst.html>

Documentation on Inprise's VisiBroker for Java product is available at:

<http://www.inprise.com/techpubs/visibroker/visibroker33/>

Getting Started

This chapter introduces the basic procedures for creating CORBA applications for Oracle8i. The emphasis in this chapter is to present the basics for developing an Oracle8i CORBA application. For advanced programming techniques and miscellaneous tips for CORBA applications, see [Chapter 5, "Advanced CORBA Programming"](#).

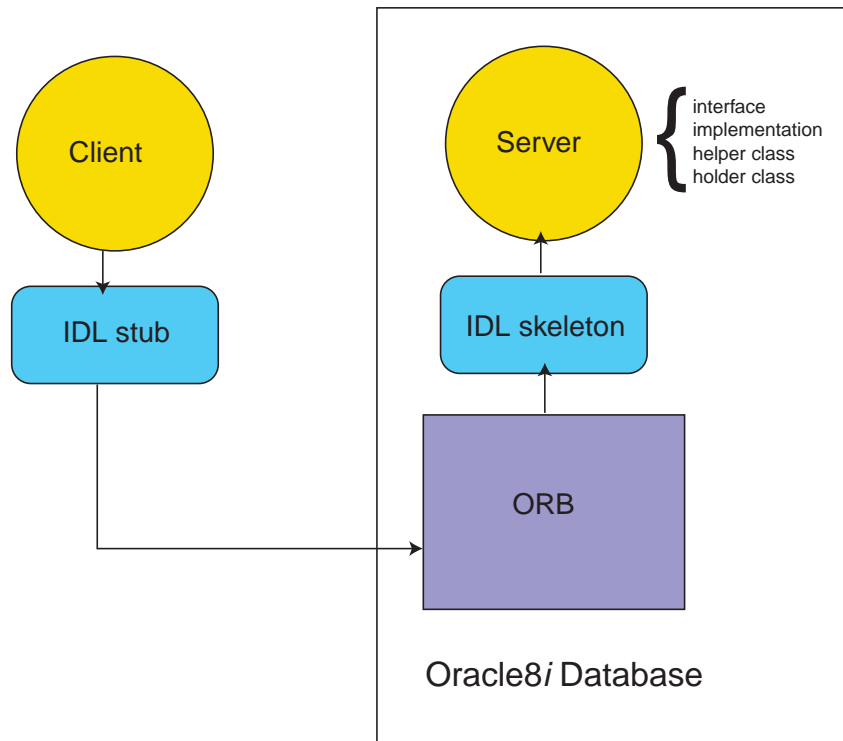
This chapter covers the following topics:

- [A First CORBA Application](#)
- [The Interface Definition Language \(IDL\)](#)
- [Activating ORBs and Server Objects](#)
- [Debugging Techniques](#)

A First CORBA Application

This section introduces the JServer CORBA application development process. It tells you how to write a simple but useful program that runs on a client system, connects to Oracle using IIOP, and invokes a method on a CORBA server object that is activated and runs inside an Oracle8i Java VM.

Figure 2-1 CORBA Application Components



As shown in [Figure 2-1](#), a CORBA application requires that you provide the client implementation, the server interface and implementation, and IDL stubs and skeletons. To create this, perform the following steps:

1. Design and write the object interfaces in IDL.
2. Generate stubs, skeletons, and helper and holder support classes.
3. Write the server object implementations.

4. Write the client implementation. This code runs outside of the Oracle8i data server on a workstation or PC.
5. Compile the Java server implementation with the client-side Java compiler. In addition, compile all the Java classes generated by the IDL compiler. Generate a JAR file to contain these classes and any other resource files that are needed.
6. Compile the client code using the JDK Java compiler.
7. Load the compiled classes into the Oracle8i database, using the `loadjava` tool and specifying the JAR file as its argument. Make sure to include all generated classes, such as stubs and skeletons. Client stubs are required in the server only when the server object acts as a client to another CORBA object.
8. Publish a name for the objects directly-accessible using the CosNaming service, so that you can access them from the client program.

The sample used in this chapter asks the user for an employee number in the EMP table and returns the employee's last name and current salary. It throws an exception if there is no employee in the database with the given ID number.

Writing Interfaces in IDL

When writing a server application, you must create an Interface Definition Language (IDL) file to define the server's interfaces. An interface is a template that defines a CORBA object. As with any object in an object oriented language, it contains methods and data elements that can be read or set. However, the interface is only a definition and so defines what the interface to an object would be if it existed. In your IDL file, each interface describes an object and the operations clients can perform on that object.

Note: For a full description of IDL, see "[The Interface Definition Language \(IDL\)](#)" on page 2-14.

This example contains a file, called `employee.idl`, that contains only a single server-side method. The `getEmployee` method takes an ID number and queries the database for the employee's name and salary.

This interface defines three things:

- a `getEmployee` method that queries the database and returns the information
- an `EmployeeInfo` data structure to hold the returned information
- a `SQLException` exception to be thrown if the employee is not found

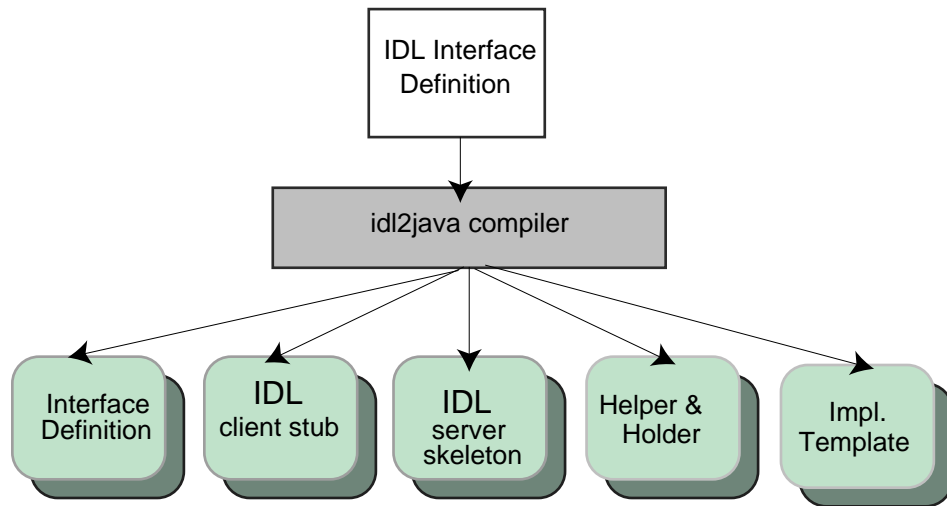
The contents of the `employee.idl` file should look like:

```
module employee {  
  
    struct EmployeeInfo {  
        wstring name;  
        long number;  
        double salary;  
    };  
  
    exception SQLException {  
        wstring message;  
    };  
  
    interface Employee {  
        EmployeeInfo getEmployee (in long ID) raises (SQLException);  
    };  
};
```

Generate Stubs and Skeletons

Use the `idl2java` compiler to compile the interface description. As shown in [Figure 2-2](#), the compiler generates the interface, implementation template, helper, and holder classes for the three objects in the IDL file, as well as a stub and skeleton class for the `Employee` interface. See ["Using IDL"](#) on page 2-15 for more information about these classes and in the *Oracle8i Java Tools Reference* for more information on the `idl2java` compiler.

Note: Because there is no use of the Tie mechanism in this example, you can invoke the compiler with the `-no_tie` option. This means that two fewer classes are generated.

Figure 2–2 IDL Compilation Generates Support Files

Compile the IDL as follows:

```
% idl2java -no_tie -no_comments employee.idl
```

Note: Because developing a CORBA application includes many compilation, loading, and publishing steps, Oracle recommends that if you are working in a command-line oriented environment, you always use a makefile or a batch file to control the process. Or, you can use IDE products such as Oracle's JDeveloper to control the process.

In compiling the `employee.idl` file, you receive the following files:

File name	File type
<code>_example_Employee.java</code>	Implementation template for server object
<code>Employee.java</code>	Employee interface definition
<code>EmployeeInfo.java</code>	EmployeeInfo interface definition
<code>SQLException.java</code>	SQLException interface definition

File name	File type
<code>_st_Employee.java</code>	IDL client stub
<code>_EmployeeImplBase.java</code>	IDL server skeleton
<code>EmployeeHelper.java</code>	Helper class for <code>Employee</code> . The most important methods this class provides are the narrow method for typecasting a returned object to be a <code>Employee</code> object and the <code>id</code> method that returns the interface's identifier.
<code>EmployeeHolder.java</code>	Holder class for <code>Employee</code> . The Holder class enables a java object to pass values back to clients.
<code>EmployeeInfoHelper.java</code>	Helper class for <code>EmployeeInfo</code> .
<code>EmployeeInfoHolder.java</code>	Holder class for the <code>EmployeeInfo</code> structure.
<code>SQLExceptionHelper.java</code>	Helper class for <code>SQLException</code> .
<code>SQLExceptionHolder.java</code>	Holder class for the <code>SQLException</code> exception.

The only file that you modify is the `_example_Employee.java` file. You should rename the `_example_Employee.java` file to a more appropriate name, such as `EmployeeImpl.java`. Once renamed, you modify the file to add your server's implementation. The `EmployeeImpl.java` file extends the IDL server skeleton, `_EmployeeImplBase.java`. Add and implement the `getEmployee` method that is defined in the `Employee.java` interface definition. In addition, you need to create the client application that invokes these methods appropriately. ["Write the Server Object Implementation"](#) on page 2-6 demonstrates how to create the server implementation of `Employee` in `EmployeeImpl.java`.

Write the Server Object Implementation

An implementation is an actual instantiation of an interface. That is, the implementation is code that implements all of the functions and data elements that were defined in the IDL interface. The following shows how to implement the `Employee` interface:

1. Modify the `EmployeeImpl.java` file, which used to be the `_example_Employee.java` file, to add your server implementation. Notice

that the `EmployeeImpl` extends the IDL-generated skeleton, `_EmployeeImplBase`.

As shown in [Figure 2-1](#), the `_EmployeeImplBase` IDL skeleton exists between the ORB and the server application, so that any invocation for the server application is performed through it. The skeleton prepares the parameters, calls the server method, and saves any return values or any out or inout parameters.

2. Implement the `getEmployee` method to query the database for the employee and return the appropriate name and salary in `EmployeeInfo`.

```
package employeeServer;

import employee.*;
import java.sql.*;

public class EmployeeImpl extends _EmployeeImplBase {

    /*constructor*/
    public EmployeeImpl() {
    }

    /*getEmployee method queries database for employee info*/
    public EmployeeInfo getEmployee (int ID) throws SQLException {
        try {
            /*create a JDBC connection*/
            Connection conn =
                new oracle.jdbc.driver.OracleDriver().defaultConnection ();

            /*Create a SQL statement for the database query*/
            PreparedStatement ps =
                conn.prepareStatement ("select ename, sal from emp where empno = ?");
            /*set the employee identifier and execute query. return the
            result in an EmployeeInfo structure */
            try {
                ps.setInt (1, ID);
                ResultSet rset = ps.executeQuery ();
                if (!rset.next ())
                    throw new SQLException ("no employee with ID " + ID);
                return new EmployeeInfo (rset.getString (1), ID, rset.getFloat (2));
            } finally {
                ps.close ();
            }
        }
        /*If a problem occurs, throw the SQLException exception*/
        } catch (SQLException e) {
```

```
        throw new SQLException (e.getMessage ());
    }
}
```

This code uses the JDBC API to perform the database query. The implementation uses a prepared statement to accommodate the variable in the WHERE clause of the query. See the *Oracle8i JDBC Developer's Guide and Reference* for more about Oracle8i JDBC. You can use SQLJ, instead of JDBC, if your statement is static.

Comparing Oracle8i Server Applications to Other ORB Applications

Most ORB applications must provide a server application that instantiates the server implementation and registers this instance with the CORBA object adapter. However, with Oracle8i, JServer instantiates the implementation and registers the resulting instance on demand for you. Thus, you do not need to provide code that initializes the ORB, instantiates the implementation, and registers the instance. The only server code that you provide is the actual server implementation. This means that your client cannot find an active server implementation instance through the ORB, since it is not instantiated until called. You must publish the implementation object in the CosNaming service. The client retrieves the object from the Name Service through a JNDI lookup. Once retrieved, the client invokes the `activate` method, which initializes an instance of the object. At this point, the client can invoke methods on the object.

Write the Client Code

After writing the server, the client needs to be created. To access the server object you must be able to refer to it by name. In order for the server object to be accessed by the client, you publish the server object in the Oracle8i database. The client code looks up the published name and activates the server object as a by-product of the look up. You can look up any server object either through JNDI or CosNaming. The JNDI option is shown in the example below. See "[JNDI Connection Basics](#)" on page 4-2 for more information on JNDI and CosNaming.

When you perform the JNDI lookup, the ORB on the server side is started and the client is authenticated using the environment properties supplied when the initial context object is created. See "[IIOP Security](#)" on page 6-1.

In order to retrieve the object from the Name Service, you must provide the following:

- [Object name](#)

- [IIOP Service Name](#)
- [Client Authentication Information](#)

Object name

The object name specifies the complete path name of the published object that you want to look up. For example: `/test/myServer`.

See "[Retrieving the JNDI InitialContext](#)" on page 4-9 for further information about the `lookup()` method.

IIOP Service Name

The service name specifies a service that an IIOP presentation manages, and it represents a database instance. The format of the service URL is explained in "[Accessing CORBA Objects Without JNDI](#)" on page 4-29. Briefly, the service name specifies the following components:

- URL prefix for the service
- the name of the host that manages the service presentation
- the port number of the listener for the target database instance on that host
- the system identifier (SID) for the database instance on the host

A typical example of a service name is `sess_iiop://localhost:2481:ORCL`, where `sess_iiop` is the URL prefix for the service, `localhost` defaults to the host of the local database, `2481` is the default listener port for IIOP connections, and `ORCL` is the SID.

Client Authentication Information

You must authenticate yourself to the database each time you connect. The type of authentication information depends on how you want to authenticate—through a username/password combination or SSL certificates. See "[IIOP Security](#)" on page 6-1 for more information.

Client Example

The client invokes the `getEmployee` method through the following steps:

1. Instantiates and populates a `JNDI InitialContext` object with the required connect properties, including authentication information. See "[JNDI Connection Basics](#)" on page 4-2.

2. Invokes the `lookup()` method on the initial context, with a URL as a parameter that specifies the service name and the name of the object to be found. The `lookup()` method returns an object reference to the `Employee` CORBA server object. See ["Using JNDI to Access Bound Objects"](#) on page 4-7 for more information.
3. Using the object reference returned by the `lookup()` method invokes the `getEmployee()` method on the object in the server. This method returns an `EmployeeInfo` class, which is derived from the IDL `EmployeeInfo` struct. For simplicity, an employee ID number is hard-coded as a parameter of this method invocation.
4. Prints the values returned by `getEmployee()` in the `EmployeeInfo` class.

```
import employee.*;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class Client {
    public static void main (String[] args) throws Exception {
        String serviceURL = "sess_iiop://localhost:2481:ORCL";
        String objectName = "/test/myEmployee";

        // Step 1: Populate the JNDI properties with connect and authentication
        // information
        Hashtable env = new Hashtable ();
        env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        env.put (Context.SECURITY_PRINCIPAL, "SCOTT");
        env.put (Context.SECURITY_CREDENTIALS, "TIGER");
        env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
        Context ic = new InitialContext (env);

        // Step 2: Lookup the object providing the service URL and object name
        Employee employee = (Employee)ic.lookup (serviceURL + objectName);

        // Step 3 (using SCOTT's employee ID number): Invoke getEmployee
        EmployeeInfo info = employee.getEmployee (7788);

        // Step 4: Print out the returned values.
        System.out.println (info.name + " " + info.number + " " + info.salary);
    }
}
```

When the client code is executed, it prints the following on the client console:

```
SCOTT 7788 3000.0
```

Compiling the Java Source

You run the client-side Java byte code compiler, `javac`, to compile all the Java source that you have created. The Java source includes the client and server object implementations, as well as the Java classes generated by the IDL compiler.

For the `Employee` example, you compile the following files:

- `employee/Employee.java`
- `employee/EmployeeHelper.java`
- `employee/EmployeeHolder.java`
- `employee/EmployeeInfo.java`
- `employee/EmployeeInfoHelper.java`
- `employee/EmployeeInfoHolder.java`
- `employee/SQLException.java`
- `employee/SQLExceptionHelper.java`
- `employee/SQLExceptionHolder.java`
- `employee/_EmployeeImplBase.java`
- `employee/_st_Employee.java`
- `EmployeeImpl.java`
- `Client.java`

Other generated Java files are compiled following the dependencies that the Java compiler uses.

Oracle8i JServer supports the Java JDK compiler, releases 1.1.6 or 1.2. Alternatively, you might be able to use other Java compilers, such as a compiler incorporated in an IDE.

Load the Classes into the Database

CORBA server objects, such as the `EmployeeImpl` object created for this example, execute inside the Oracle8i database server. You must load all your classes into the server—through the `loadjava` command-line tool—so that they can be activated

by the ORB upon demand. You also load all dependent classes, such as IDL-generated Holder and Helper classes, and classes the server object uses, such as the `EmployeeInfo` class of this example.

Use the `loadjava` tool to load each of the server classes into the Oracle8i database. For the `Employee` example, issue the `loadjava` command in the following way:

```
% loadjava -resolve -user scott/tiger
employee/Employee.class employee/EmployeeHolder.class
employee/EmployeeHelper.class employee/EmployeeInfo.class
employee/EmployeeInfoHolder.class employee/EmployeeInfoHelper.class
employee/SQLException.class employee/SQLExceptionHolder.class
employee/SQLExceptionHelper.class employee/_st_Employee.class
employee/_EmployeeImplBase.class employeeServer/EmployeeImpl.class
```

Note: You do not load any client implementation classes or any other classes not used on the server side.

It is sometimes more convenient to combine the server classes into a JAR file, and simply use that file as the argument to the `loadjava` command. In this example, you could issue the command:

```
% jar -cf0 myJar.jar employee/Employee.class employee/EmployeeHolder.class \
employee/EmployeeHelper.class employee/EmployeeInfo.class \
employee/EmployeeInfoHolder.class employee/EmployeeInfoHelper.class \
employee/SQLException.class employee/SQLExceptionHolder.class \
employee/SQLExceptionHelper.class employee/_st_Employee.class \
employee/_EmployeeImplBase.class employeeServer/EmployeeImpl.class
```

Then, execute the `loadjava` command as follows:

```
% loadjava -resolve -user scott/tiger myJar.jar
```

Publish the Object Name

The final step in preparing the application is to publish the name of the CORBA server object implementation in the Oracle8i database. See ["The Name Space"](#) on page 4-3 and the `publish` section in the *Oracle8i Java Tools Reference* for information about publishing objects.

For the example in this section, you can publish the server object using the `publish` command as follows:

```
% publish -republish -user scott -password tiger -schema scott
```

```
-service sess_iiop://localhost:2481:ORCL  
/test/myEmployee employeeServer.EmployeeImpl employee.EmployeeHelper
```

This command specifies the following:

- `publish`—run the publish command
- `-republish`—overwrite any published object of the same name
- `-user scott`—scott is the username for the schema doing the publishing
- `-password tiger`—Scott's password
- `-schema scott`—the name of the schema in which to resolve classes
- `-service sess_iiop://localhost:2481:ORCL`—establishes the service name (see also "[Service Context Class](#)" on page 4-16)
- `/test/myEmployee`—the name for the published object
- `employeeServer.EmployeeImpl`—the name of the class, loaded in the database, that implements the server object
- `employee.EmployeeHelper`—the name of the helper class

Run the Example

To run this example, execute the client class using the client-side JVM. For this example, you must set the CLASSPATH for the `java` command to include:

- the standard Java library archive (`classes.zip`)
- any class files the client ORB uses, such as those in VisiBroker for Java `vbjapp.jar` and `vbjorb.jar`
- the Oracle8i-supplied JAR files: `mts.jar` and `aurora_client.jar`

If you are using JDBC, include one of the following JAR files:

- `classes111.zip` for JDBC 1.1 support
- `classes12.zip` for JDBC 1.2 support

If you are using SSL, include one of the following JAR files:

- `javax-ssl-1_1.jar` and `jssl-1_1.jar` for SSL 1.1 support
- `javax-ssl-1_2.jar` and `jssl-1_2.jar` for SSL 1.2 support

You can locate these libraries in the `lib` and `jlib` directories under the Oracle home location in your installation.

The following invocation of the JDK `java` command runs this example.

Note: The UNIX shell variable `$ORACLE_HOME` might be represented as `%ORACLE_HOME%` on Windows NT. The `JDK_HOME` is the installation location of the Java Development Kit (JDK).

```
% java -classpath .:${ORACLE_HOME}/lib/aurora_client.jar
:${ORACLE_HOME}/lib/mts.jar
:${ORACLE_HOME}/jdbc/lib/classes111.zip:
${ORACLE_HOME}/sqlj/lib/translator.zip:${ORACLE_HOME}/lib/vbjorb.jar:
${ORACLE_HOME}/lib/vbjapp.jar:${JDK_HOME}/lib/classes.zip
Client
sess_iiop://localhost:2481:ORCL
/test/myEmployee
scott tiger
```

This example assumes that you invoke the client with the following arguments on the command line:

- CLASSPATH libraries
- client object
- service name
- name of the published object to activate
- username
- password

Note: From the `java` command you can see why it is almost always better to use a makefile or a batch file to build CORBA applications.

The Interface Definition Language (IDL)

CORBA provides language independence. CORBA objects written in one language can send requests to objects implemented in a different language. Objects implemented in an object-oriented language such as Java or Smalltalk can talk to objects written in C or COBOL, and the converse.

Language independence is achieved through the use of a specification meta-language that defines the interfaces that an object—or a piece of legacy code wrapped to look like an object—presents to the outside world. As in any object-oriented system, a CORBA object can have its own private data and its own private methods. The specification of the public data and methods is the interface that the object presents to the outside world.

IDL is the language that CORBA uses to specify its objects. You do not write procedural code in IDL—its only use is to specify data, methods, and exceptions.

Each CORBA vendor supplies a compiler that translates IDL specifications into a specific language. Oracle8i JServer uses the `idl2java` compiler from Inprise. The `idl2java` compiler translates your IDL interface specifications into Java classes. See the *Oracle8i Java Tools Reference* for more information on this tool.

Note: The `idl2java` compiler accepts only ASCII characters. Do not use ISO Latin-1 or other non-ASCII NLS characters in IDL files.

Using IDL

The following example demonstrates the IDL for the `HelloWorld` example. See "[Basic Example](#)" on page A-2 for the complete example.

```
module hello {  
    interface Hello {  
        wstring helloWorld();  
    };  
};
```

IDL consists of a *module*, which contains a group of related object interfaces. The IDL compiler uses the module name to name a directory where the Java classes are placed after generation. Also, the module name is used to name the Java package for the resulting classes.

This module defines a single interface: `Hello`. The `Hello` interface defines a single operation: `helloWorld`, which takes no parameters and returns a `wstring` (a wide string, which is mapped to a Java `String`).

Note: IDL data and exception types, such the `wstring` shown in the preceding example, are not specified in this guide. Although some of the IDL to Java bindings are listed in this guide (for example see ["IDL Types"](#) on page 2-19), CORBA developers should refer to the OMG specification for complete information about IDL and IDL types. See ["For More Information"](#) on page 1-8.

The module and interface names must be valid Java identifiers and valid file names for your operating system. When naming interfaces and modules, remember that both Java and CORBA objects are portable, and that some operating systems are case sensitive and some are not, so be sure to keep names distinct in your project.

Nested Modules

You can nest modules. For example, an IDL file that specifies the following modules maps to the Java package hierarchy `package org.omg.CORBA`.

```
module org {
  module omg {
    module CORBA {
      ...
    };
    ...
  };
  ...
};
```

Running the IDL Compiler

Assume that the HelloWorld IDL is saved in a file called `hello.idl`. When you run `idl2java` to compile the `hello` module, eight Java class files are generated and are placed in a subdirectory named `hello` in the same directory as the IDL file:

```
% idl2java hello.idl
Traversing hello.idl
Creating: hello/Hello.java
Creating: hello/HelloHolder.java
Creating: hello/HelloHelper.java
Creating: hello/_st_Hello.java
Creating: hello/_HelloImplBase.java
Creating: hello/HelloOperations.java
Creating: hello/_tie_Hello.java
Creating: hello/_example_Hello.java
```

The ORB uses these Java classes to invoke a remote object, pass and return parameters, and perform various other things. You can control the files generated, where they are put, and other aspects of IDL compiling—such as whether the IDL compiler generates comments in the Java files. See the complete description of the `idl2java` compiler in the *Oracle8i Java Tools Reference*.

Each of the files generated is described below.

Hello This specifies, in Java, what the interface to a Hello object looks like. In this case, the interface is:

```
package hello;
public interface Hello extends org.omg.CORBA.Object {
    public java.lang.String helloWorld();
}
```

Because the file is put in a `hello` directory, it takes the package spec from that name. All CORBA basic interface classes subclass, directly or indirectly, the following:

```
org.omg.CORBA.Object.
```

You must implement the methods in the interface. It is recommended that the implementation class for the `hello.java` interface be named `helloImpl`, but this naming convention is not a requirement.

HelloHolder The application uses the holder class when parameters in the interface operation are of types `out` or `inout`. Because the ORB passes Java parameters by value, special holder classes are necessary to provide for parameter return values.

HelloHelper The helper classes contain methods that read and write the object to a stream, and cast the object to and from the type of the base class. For example, the helper class has a `narrow()` method that is used to cast an object to the appropriate type, as in the following code:

```
LoginServer lserver = LoginServerHelper.narrow
    (orb.string_to_object (loginIOR));
```

Note that when you get an object reference using the JNDI `InitialContext lookup()` method, you do not have to call the helper `narrow()` method. The ORB calls it automatically for you.

<code>_st_Hello</code>	<p>The generated files that have <code>_st_</code> prefixed to the interface name are the stub files or client proxy objects. (<code>_st_</code> is a VisiBroker-specific prefix.)</p> <p>These classes are installed on the client that calls the remote object. In effect, when a client calls a method on the remote object, it is really calling into the stub, which then performs the operations necessary to perform a remote method invocation. For example, it must marshall parameter data for transport to the remote host.</p>
<code>_HelloImplBase</code>	<p>Generated source files of the form <code>_<i><interfaceName></i>ImplBase</code> are the skeleton files. A skeleton file is installed on the server and communicates with the stub file on the client, in that it receives the message on the ORB from the client and upcalls to the server. The skeleton file also returns parameters and return values to the client.</p>
<code>HelloOperations</code> <code>_tie_Hello</code>	<p>The server uses these two classes for Tie implementations of server objects. See "Using the CORBA Tie Mechanism" on page 5-10 for information about Tie classes.</p>
<code>_example_Hello</code>	<p>The <code>_example_<i><interfaceName></i></code> class provides you with a template for your server object implementation. You can copy the example code to the directory where you will implement the <code>Hello</code> server object, rename it, (<code>HelloImpl.java</code> is used in the examples in this Guide), and implement the methods.</p>

IDL Interface Body

An IDL interface body contains the following kinds of declarations:

constants	The constant values that the interface exports.
types	Type definitions.
exceptions	Exception structures that the interface exports.
attributes	Any associated attributes exported by the interface.
operations	Operations are the methods that the interface supports.

IDL Types

This section gives a brief description of IDL datatypes and their mapping to Java datatypes. For more information about IDL types not covered here, see the CORBA specifications and the books cited in "[For More Information](#)" on page 1-8.

Basic Types

Mapping between IDL basic types and Java primitive types is straightforward. [Table 2-1](#) shows the mappings, as well as possible CORBA exceptions that can be raised on conversion.

Table 2-1 IDL to Java Datatype Mappings

CORBA IDL Datatype	Java Datatype	Exception
boolean	boolean	
char	char	CORBA::DATA_CONVERSION
wchar	char	
octet	byte	
string	java.lang.String	CORBA::MARSHAL CORBA::DATA_CONVERSION
wstring	java.lang.String	CORBA::MARSHAL
short	short	
unsigned short	short	
long	int	
unsigned long	int	
long long	long	
unsigned long long	long	
float	float	
double	double	

The IDL character type `char` is an 8-bit type, representing an ISO Latin-1 character that maps to the Java `char` type, which is a 16-bit unsigned element representing a Unicode character. On parameter marshalling, if a Java `char` cannot be mapped to an IDL `char`, a CORBA `DATA_CONVERSION` exception is thrown.

The IDL `string` type contains IDL chars. On conversion between Java `String`, and IDL `string`, a `CORBA DATA_CONVERSION` can be thrown. Conversions between Java strings and bounded IDL `string` and `wstring` can throw a `CORBA MARSHALS` exception if the Java `String` is too large to fit in the IDL `string`.

Constructed Types

Perhaps the most useful IDL constructed (aggregate) type for the Java developer is the struct. The IDL compiler converts IDL structs to Java classes. For example, the IDL specification:

```
module employee {
    struct EmployeeInfo {
        long empno;
        wstring ename;
        double sal;
    };
    ...
}
```

causes the IDL compiler to generate a separate Java source file for an `EmployeeInfo` class. It looks like this:

```
package employee;
final public class EmployeeInfo {
    public int empno;
    public java.lang.String ename;
    public double sal;
    public EmployeeInfo() {
    }
    public EmployeeInfo(
        int empno,
        java.lang.String ename,
        double sal
    ) {
        this.empno = empno;
        this.ename = ename;
        this.sal = sal;
    }
    ...
}
```

The class contains a public constructor with parameters for each of the fields in the struct. The field values are saved in instance variables when the object is constructed. Typically, these are passed by value to CORBA objects.

Collections

The two types of ordered collections in CORBA are sequences and arrays. An IDL sequence maps to a Java array with the same name. An IDL array is a multidimensional aggregate whose size in each dimension must be established at compile time.

The ORB throws a CORBA MARSHAL exception at runtime if sequence or array bounds are exceeded when Java data is converted to sequences or arrays.

IDL also generates a holder class for a sequence. The holder class name is the sequence's mapped Java class name with `Holder` appended to it.

The following IDL code shows how you can use a sequence of structs to represent information about employees within a department:

```
module employee {
    struct EmployeeInfo {
        long empno;
        wstring ename;
        double sal;
    };

    typedef sequence <EmployeeInfo> employeeInfos;

    struct DepartmentInfo {
        long deptno;
        wstring dname;
        wstring loc;
        EmployeeInfos employees;
    };
};
```

The Java class code that the IDL compiler generates for the `DepartmentInfo` class is:

```
package employee;
final public class DepartmentInfo {
    public int deptno;
    public java.lang.String dname;
    public java.lang.String loc;
    public employee.EmployeeInfo[] employees;
    public DepartmentInfo() {
    }
    public DepartmentInfo(
        int deptno,
        java.lang.String dname,
        java.lang.String loc,
```

```
        employee.EmployeeInfo[] employees
    ) {
        this.deptno = deptno;
        this.dname = dname;
        this.loc = loc;
        this.employees = employees;
    }
```

Notice that the sequence `employeeInfos` is generated as a Java array `EmployeeInfo[]`.

Specify an array in IDL as follows:

```
const long ArrayBound = 12;
typedef long larray[ArrayBound];
```

The IDL compiler generates this as:

```
public int[] larray;
```

When you use IDL constructed and aggregate types in your application, you must make sure to compile the generated `.java` files and load them into the Oracle8i database when the class is a server object. You should scan the generated `.java` files, and make sure that all required files are compiled and loaded. Study the `Makefile` (UNIX) or the `makeit.bat` batch file (Windows NT) of CORBA examples that define these types to see how the set of IDL-generated classes is compiled and loaded into the data server.

Exceptions

You can create new user exception classes in IDL with the exception key word. For example:

```
exception SQLError {
    wstring message;
};
```

The IDL can declare that operations raise user-defined exceptions. For example:

```
interface employee {
    attribute name;
    exception invalidID {
        wstring reason;
    };
    ...
    wstring getEmp(long ID)
        raises(invalidID);
```



```
};
};
```

CORBA System Exceptions

Mapping between OMG CORBA system exceptions and their Java form is also quite straightforward. These mappings are shown in [Table 2-2](#).

Table 2-2 CORBA and Java Exceptions

OMG CORBA Exception	Java Exception
CORBA::PERSIST_STORE	org.omg.CORBA.PERSIST_STORE
CORBA::BAD_INV_ORDER	org.omg.CORBA.BAD_INV_ORDER
CORBA::TRANSIENT	org.omg.CORBA.TRANSIENT
CORBA::FREE_MEM	org.omg.CORBA.FREE_MEM
CORBA::INV_IDENT	org.omg.CORBA.INV_IDENT
CORBA::INV_FLAG	org.omg.CORBA.INV_FLAG
CORBA::INTF_REPOS	org.omg.CORBA.INTF_REPOS
CORBA::BAD_CONTEXT	org.omg.CORBA.BAD_CONTEXT
CORBA::OBJ_ADAPTER	org.omg.CORBA.OBJ_ADAPTER
CORBA::DATA_CONVERSION	org.omg.CORBA.DATA_CONVERSION
CORBA::OBJECT_NOT_EXIST	org.omg.CORBA.OBJECT_NOT_EXIST
CORBA::TRANSACTIONREQUIRED	org.omg.CORBA.TRANSACTIONREQUIRED
CORBA::TRANSACTIONROLLEDBACK	org.omg.CORBA.TRANSACTIONROLLEDBACK
CORBA::INVALIDTRANSACTION	org.omg.CORBA.INVALIDTRANSACTION

Getting by Without IDL

The Oracle8i JVM development environment offers the Inprise Caffeine tools, which enable development of pure Java distributed applications that follow the CORBA model. You can write your interface specifications in Java and use the `java2iiop` tool to generate CORBA-compatible Java stubs and skeletons.

Developers can also use the `java2idl` tool to code in pure Java, yet still have IDL available that can be shipped to customers who are using a CORBA server that does not support Java. This tool generates IDL from Java interface specifications.

See the *Oracle8i Java Tools Reference*, for more information about `java2iiop` and `java2idl`.

Activating ORBs and Server Objects

A CORBA application requires that an ORB be active on both the client system and the system running the server. In looking at the examples shown so far in this chapter, it is not obvious how the ORB is activated, either on the client or the server. This section presents more information about that topic.

Client Side

The client-side ORB is normally initialized in one of two ways:

- The ORB is implicitly instantiated when the client instantiates the server object either through the `JNDI.lookup()` method on the `JNDI.InitialContext` object.
- The ORB is explicitly instantiated when a pure CORBA client invokes the CORBA ORB `init` method. See "[Aurora ORB Interface](#)" on page 5-12 for a full explanation of the `init` method.

Note: The only other time that you explicitly initialize the ORB on the client through the `ORB.init` method is when you are in a callback scenario. See "[Implementing CORBA Callbacks](#)" on page 5-3 for a full discussion of callbacks. This discussion also includes an example that shows the ORB is initialized within the object that is called back to by the server.

Server Side

The presentation that manages IIOP requests starts the ORB on the server when the session is created. If you want to retrieve the ORB instance, use the CORBA `oracle.aurora.jndi.orb_dep.Orb.init` method. See "[Aurora ORB Interface](#)" on page 5-12 for a full explanation of this method.

About Object Activation

Objects are activated on demand. When a client looks up an object, the ORB loads the object into memory and caches it. To activate the object, the ORB looks up the class by the fully-qualified class name under which the object was published. The

class name is resolved in the schema defined at publication time, rather than the caller's schema. See the description of the command-line tool `publish` in the *Oracle8i Java Tools Reference* for more information.

When the class is located, the ORB creates a new instance of the class, using `newInstance()`. For this reason, the no-argument constructor of a persistent object class must be **public**. If the class implements the `oracle.aurora.AuroraServices.ActivatableObject` interface (as determined by the Java reflection API), then the `_initializeAuroraObject()` message is sent to the instance. (See "[Using the CORBA Tie Mechanism](#)" on page 5-10 for an example that requires `_initializeAuroraObject()`).

There is no need for the server implementation to register its published objects with the object adapter using a `boa.obj_is_ready()` call—the JServer ORB performs this automatically.

You register transient objects generated by other objects, such as persistent published objects, with the BOA using `obj_is_ready()`. For an example, see the factory demo in the `examples/corba/basic/factory` directory of the product CD.

CORBA Interceptors

Visibroker enables you to implement interceptors. The Visibroker documentation provides details for how to create them.

Debugging Techniques

Until Java IDEs and JVMs support remote debugging, you can adopt several techniques for debugging your CORBA client and server code.

1. Use JDeveloper for debugging any Java applications. JDeveloper has provided a user interface that utilizes JServer's debugging facilities. You can successfully debug an object loaded into the database by using JDeveloper's debugger. See the JDeveloper documentation for instructions.
2. Use a prepublished `DebugAgent` object for debugging objects executing on a server. See "[Using a Debug Agent for Debugging Server Applications](#)" on page 2-26 for more information.
3. Perform stand-alone ORB debugging using one machine and ORB tracing.

Debug by placing both the client and server in a single address space in a single process. Use of an IDE for client or server debugging is optional, though highly desirable.

4. Use Oracle8i trace files.

In the client, the output of `System.out.println()` goes to the screen. However, in the Oracle8i ORB, all messages are directed to the server trace files. The directory for trace files is a parameter specified in the database initialization file. Assuming a default install of the product into a directory symbolically named `$ORACLE_HOME`, then the trace file would appear as

```
`${ORACLE_HOME}/admin/<SID>/bdump/ORCL_s000x_xxx.trc
```

where ORCL is the SID, and x_xxx represents a process ID number. Do not delete trace files after the Oracle instance has been started, or no output is written to a trace file. If you do delete trace files, stop and then restart the server.

5. Use a single Oracle MTS server.

For debugging only, set the `MTS_SERVERS` parameter in your `INITSID.ORA` file to `MTS_SERVERS = 1`, and set the `MTS_MAX_SERVERS` to 1. Having multiple MTS servers active means that a trace file is opened for each server process, and, thus, the messages get spread out over several trace files, as objects get activated in more than one session.

6. Use the `printback` example to redirect `System.out`. This example is available in the demo directory, `demo/examples/corba/basic/printback`.

Using a Debug Agent for Debugging Server Applications

The procedure for setting up your debugging environment is discussed fully in the Oracle8i Java Developer's Guide. However, it discusses starting the debug agent using a `DBMS_JAVA` procedures. Within a CORBA application, you can start, stop, and restart the debug agent using the `oracle.aurora.debug.DebugAgent` class methods. These methods perform exactly as their `DBMS_JAVA` counterparts perform.

```
public void start( java.lang.String host, int port, long timeout_seconds)
                 throws DebugAgentError
public void stop() throws DebugAgentError
public void restart(long timeout) throws DebugAgentError
```

Example 2-1 Starting a DebugAgent on the Server

The following example shows how to debug an object that exists on the server. First, you need to start a debug proxy through the `debugproxy` command-line tool. This example informs the `debugproxy` to start up the `jdb` debugger when contacted by the debug agent.

Once you execute this command, start your client, which will lookup the intended object to be debugged, lookup the `DebugAgent` that is prepublished as `"/etc/debugagent"`, and start up the `DebugAgent`.

Once the `DebugAgent` starts, the `debugproxy` starts up the `jdb` debugger and allows you to set your breakpoints. Since you have a specified amount of time before the `DebugAgent` times out, the first thing you should do is suspend all threads. Then, set all of your breakpoints before resuming. This suspends the timeout until you are ready to execute.

proxy window on tstHost

```
% debugproxy -port 2286 start jdb -password
. (wait until a debug agent starts up and
.  contact this proxy... when it does, jdb
.  starts up automatically and you can set
.  breakpoints and debug the object, as follows:)
> suspend
> load SCOTT:Bank
> stop in Bank:updateAccount
> resume
> ...
```

client code

```
main( ... )
{
  //retrieve the object that you want to debug
  Bank b = (Bank)ic.lookup(sessURL + "/test/Bank");
  //lookup debugagent from JNDI
  DebugAgent dbagt = (DebugAgent)ic.lookup(svcURL + "/etc/debugagent");
  //start the debug agent and give the proxy host, port, and a timeout
  dbagt.start("tstHost",2286,30);
  ...
  //execute methods within Bank)
  ...
  //stop the agent when you want to
  dbagt.stop();
  //restart the agent when you want to
  dbagt.restart(30);
}
```

Configuring IIOP Applications

Configuring IIOP-based applications, whether EJB or CORBA applications, involves configuring the appropriate listener and MTS server for session-based IIOP communications. The process for configuring IIOP-based applications involves both database and network configuration. These elements are discussed in the sections below:

- [Overview](#)
- [Oracle8i Typical or Minimal Installation](#)
- [Oracle8i Custom Installation](#)
- [Manual Install and Configuration](#)
- [Advanced Configuration Options](#)

Overview

Clients access EJB and CORBA applications in the database over an Internet Inter-Orb Protocol (IIOP) connection. IIOP is an implementation of General Inter-Orb Protocol (GIOP) over TCP/IP. All IIOP connections for CORBA or EJB clients that communicate with the database must have IIOP configured on the database and within the Net8 listener, unless you have the following scenario:

- the listener that you want to connect to is on the same node as the database
- the database that you want to use has configured a generic dispatcher
- you use the dynamic registration tool described in "[Dynamic Listener Endpoint Registration](#)" on page 3-15 to enable the listener to manage IIOP requests

Otherwise, you must configure the database and the listener, as follows:

Entity	Description	Configuration Tool
Database	To support an IIOP connection, you must configure the database for GIOP in MTS mode.	Configure the database MTS dispatchers for IIOP through the Database Configuration Assistant. This tool is started under the covers by the Typical and Custom Oracle8i install.
Net8 Listener	To support an IIOP connection, you must configure the Net8 listener to accept an IIOP connection over defined ports 2481 or 2482.	Configure the Net8 listener for IIOP through the Net8 Assistant.

The database supports incoming requests through a presentation. The presentation protocol is responsible for making sure data is represented in a format the application and session layers can accommodate. Both the listener and the dispatcher accept incoming network requests based upon the presentation that is configured. For IIOP, you configure a GIOP presentation.

Note: For security concerns, you must decide if your IIOP connection will be Security Socket Layer (SSL) enabled.

- See ["Using the Secure Socket Layer"](#) on page 6-3 for information on SSL.
 - See ["Configuring SSL for EJB and CORBA"](#) on page 3-17 for information on how to configure SSL.
-
-

The configuration for an IIOP connection can be handled in one of three ways:

- [Oracle8i Typical or Minimal Installation](#)—If you choose the Typical or Minimal Oracle8i installation, you receive configuration for session-based, non-SSL IIOP connections for both the database and the listener.
- [Oracle8i Custom Installation](#)—If you choose the JServer option within a "Custom" Oracle8i installation, you receive configuration for session-based, non-SSL IIOP connections for the database. You have to invoke the Net8 Assistant to configure IIOP for the listener.
- [Manual Install and Configuration](#)—If you install JServer by invoking the `initjvm.sql` script, you must manually configure your IIOP connection. All configuration is done manually either by invoking the Database Configuration and Net8 Assistants directly or by editing the various initialization parameter files.

Oracle8i Typical or Minimal Installation

During a Typical installation of the server, JServer is installed and configured for you. You automatically receive configuration for an MTS database with session-based IIOP connections through the listener using non-SSL TCP/IP.

After the typical install is complete, the following line is added to your database initialization file:

```
mts_dispatchers="(protocol=tcp)(presentation=oracle.aurora.server.SGiopServer)"
```

If, instead, you installed the Advanced Security Option and you want the SSL-based TCP/IP connection, edit your database initialization file to remove the hash mark (#) from the following line:

```
mts_dispatchers="(protocol=tcps)(presentation=oracle.aurora.server.SGiopServer)"
```

Note: The `(protocol=tcps)` attribute identifies the connection as SSL-enabled.

In addition, the listener is configured for IIOP. The following is placed within your `listener.ora` file:

```
listener=
  (description_list =
    (description=
      (address=(...))
      (protocol_stack=
        (presentation=GIOP)
        (session=RAW)
      )
    )
  )
```

After configuration, the client directs its request to a URL that includes the host and port, which identifies the listener, and either the SID or database service name, which identifies the database. The following shows the syntax for this request:

```
session_iiop://<hostname>/:<portnumber>/:<SID | service_name>
```

Oracle8i Custom Installation

If, within a Custom install, you choose the JServer option (as shown in [Figure 3-1](#)), the Database Configuration Assistant configures an MTS database for session-based IIOP connections, using non-SSL TCP/IP.

Note: If you choose the Typical or Minimal options within the custom choices, your configuration is the same as defined in "[Oracle8i Typical or Minimal Installation](#)" on page 3-3.

Figure 3–1 Choosing the JServer option

This places the following line within your database initialization file:

```
mts_dispatchers="(protocol=tcp)(presentation=oracle.aurora.server.SGIopServer)"
```

If, instead, you installed the Advanced Security Option and you want the SSL-based TCP/IP connection, edit your database initialization file to remove the hash mark (#) from the following line:

```
mts_dispatchers="(protocol=tcps)(presentation=oracle.aurora.server.SGIopServer)"
```

Note: The (protocol=tcps) attribute identifies the connection as SSL-enabled.

After the installation is complete, you must bring up the Net8 Assistant to configure the listener for IIOP connections.

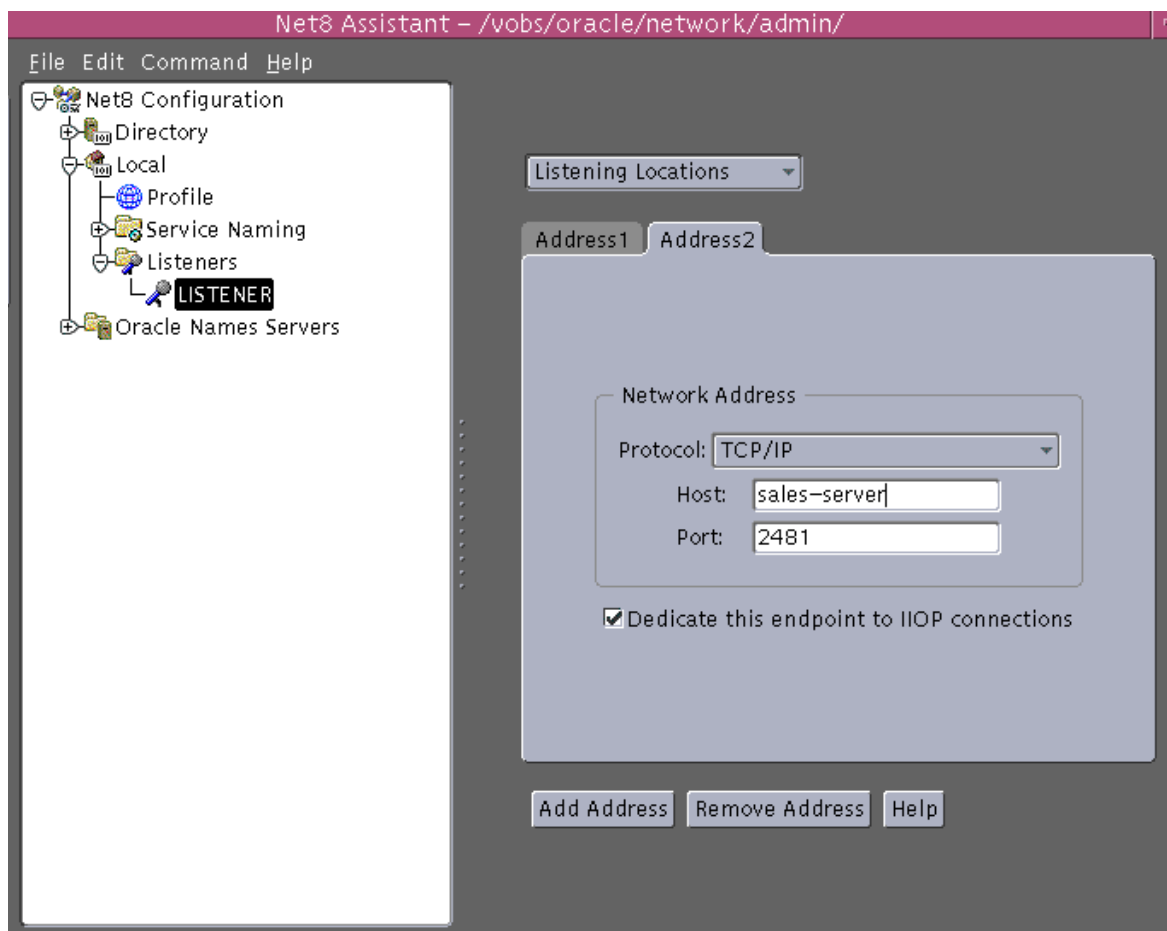
Net8 Assistant

The Net8 Assistant can modify any of the listener settings. The following is a brief description of the task you must do to configure the listener through the Net8 Assistant. For a fuller explanation, see the *Net8 Administrator's Guide*.

1. Start Net8 Assistant
 - On UNIX, run `netasst` at `$ORACLE_HOME/bin`.
 - On Windows NT, choose Start > Programs > Oracle - *HOME_NAME* > Network Administration > Net8 Assistant.

2. In the navigator pane, expand Local > Listeners.

This brings you to the listener location panel, as shown in [Figure 3-2](#).

Figure 3–2 IIOIP Listening Port Configuration

3. Select a listener.
4. From the list in the right pane, select Listening Locations.
5. Select the TCP/IP or TCP/IP with SSL protocol from the Protocol list.
6. Enter the host name of the database in the Host field.
7. Enter port 2481 in the Port field if the chosen protocol is TCP/IP, or enter port 2482 if the chosen protocol is TCP/IP with SSL.
8. Click "Dedicate this endpoint to IIOIP connections".

9. Choose File > Save Network Configuration.

This places the following within your `listener.ora` file:

```
listener=
  (description_list =
    (description=
      (address=(protocol=tcp)(host=sales-server)(port=2481)))
      (protocol_stack=
        (presentation=GIOP)
        (session=RAW)
      )
    )
  )
)
```

After configuration, the client directs its request to a URL that includes the host and port, which identifies the listener, and either the SID or database service name, which identifies the database. The following shows the syntax for this request:

```
session_iiop://<hostname>/:<portnumber>/:<SID | service_name>
```

Manual Install and Configuration

If you did not install JServer through either the Typical or Custom install options, you can add JServer to an existing database with the `initjvm.sql` script. See the *Oracle8i Java Developer's Guide* for more information on this script.

Once you have installed JServer, you can configure your IIOP connections either through the tools—Database Configuration and Net8 Assistants—or by manually editing the initialization files.

Configuring Through Tools

1. Configure the database for IIOP through the Database Configuration Assistant. To start up the Database Configuration Assistant, do the following:
 - On UNIX, run `dbassist` at `$ORACLE_HOME/bin`.
 - On Windows NT, choose Start > Programs > Oracle - `HOME_NAME` > Database Administration > Database Configuration Assistant.

After starting up the Database Configuration Assistant, choose the JServer option. For information on what this does to your initialization files, see "[Oracle8i Custom Installation](#)" on page 3-4

2. Configure the listener for IIOp through the Net8 Assistant. These steps are described in "[Net8 Assistant](#)" on page 3-6.

Configuring Through Editing Initialization Files

The presentation layer within the database identifies the type of connection your client is using to access the database. To identify the GIOP presentation, you use `oracle.aurora.server.SGiopServer`, which is the configuration for session-based IIOp connections. CORBA applications can activate objects within multiple sessions and are not limited to objects within the single session that the client initiated. These connections identify both a session and the standard IIOp semantics.

To configure an IIOp connection, you specify the GIOP presentation in the following initialization files:

1. Configure the IIOp connection in the database initialization file—You configure the PRESENTATION attribute of the MTS_DISPATCHERS parameter.

This section describes only the PRESENTATION attribute for the MTS_DISPATCHERS parameter. For a full description of MTS configuration, see the *Net8 Administrator's Guide*.

2. Configure the Net8 listener for IIOp connections.

Both steps are described in more detail below.

1. Configure the IIOp Connection in the Database Initialization File

To configure an IIOp connection within the database, you can manually edit the database initialization file.

The following is the syntax for the MTS_DISPATCHERS parameter:

```
mts_dispatchers="(protocol=tcp | tcps)
                (presentation=oracle.aurora.server.SGiopServer)"
```

The attributes for MTS_DISPATCHER are described below:

Attribute	Description
PROTOCOL (PRO or PROT)	Specifies the TCP/IP or TCP/IP with SSL protocol, which the dispatcher will generate a listening end point for. Valid values: TCP (for TCP/IP) or TCPS (for TCP/IP with SSL)

Attribute	Description
PRESENTATION (PRE or PRES)	<p data-bbox="696 262 1265 314">Enables support for GIOP. Supply the following value for a GIOP presentation:</p> <ul data-bbox="696 331 1265 434" style="list-style-type: none"> <li data-bbox="696 331 1265 434">■ <code>oracle.aurora.server.SGiopServer</code> for session-based GIOP connections. This presentation is valid for TCP/IP and TCP/IP with SSL.

Note: If you configure several `MTS_DISPATCHERS` within your database initialization file, each MTS definition must follow each other. You should not define any other configuration parameters between the `MTS_DISPATCHER` definitions.

For example, to configure MTS for session-based IIOIP connections through the listener using non-SSL TCP/IP, add the following within your database initialization file:

```
mts_dispatchers="(protocol=tcp)(presentation=oracle.aurora.server.SGiopServer) "
```

2. Configure a Listener for the Incoming Connection

Each listener is configured to listen on a well-known port number, and the client communicates with the listener using this port number. To support CORBA and EJB, the listener must be configured to listen for IIOIP clients on either ports 2481 or 2482.

You can either use the Net8 Assistant to configure your listener or manually configure the listener within the `listener.ora` file. Oracle recommends that you use the Net8 Assistant. See "[Net8 Assistant](#)" on page 3-6 for information on the Net8 Assistant.

To configure the listener manually, you must modify the listener's `DESCRIPTION` parameter within the `listener.ora` file.

Modify the LISTENER.ORA DESCRIPTION Parameter You must configure the listener with a GIOP listening address. The following example configures a GIOP presentation for non-SSL TCP/IP with port number 2481. You use port 2481 for non-SSL and port 2482 for SSL.

For GIOP, the `PROTOCOL_STACK` parameter is added to the `DESCRIPTION` when configuring an IIOIP connection to `sales-server`:


```

listener=
  (description_list=
    (description=
      (address=(protocol=tcp)(host=sales-server)(port=2481))
      (protocol_stack=
        (presentation=giop)
        (session=raw))))

```

The following table gives the definition for each of the GIOP parameters:

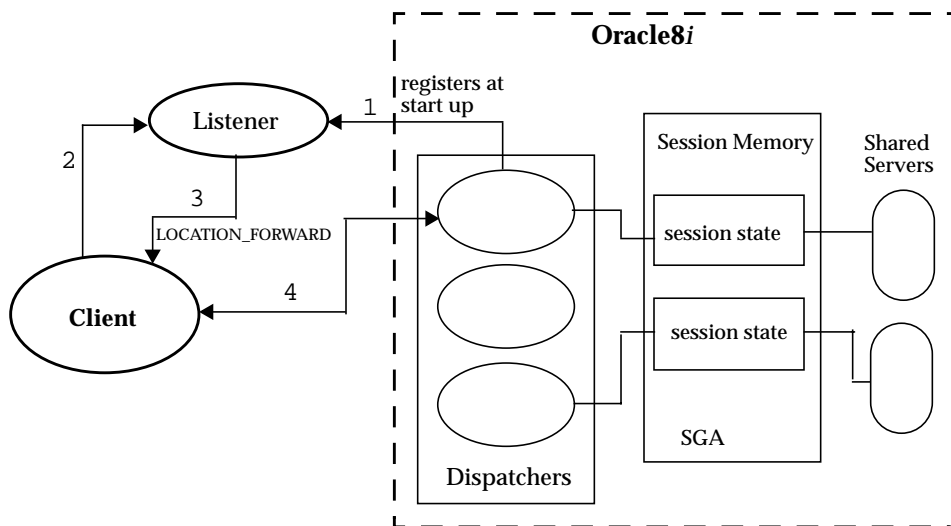
Attribute	Description
PROTOCOL_STACK	Identifies the presentation and session layer information for a connection.
(PRESENTATION=GIOP)	Identifies a presentation of GIOP for IIOP clients. GIOP supports <code>oracle.aurora.server.SGiopServer</code> using TCP/IP.
(SESSION=RAW)	Identifies the session layer. There is no specific session-layer for IIOP clients.
(ADDRESS=...)	Specifies a listening address that uses TCP/IP on either port 2481 for non-SSL or port 2482 for SSL. If non-SSL, the protocol should be TCP; for SSL, the protocol should be defined as TCPS.

Advanced Configuration Options

- [Database Listeners and Dispatchers](#)
- [Dynamic Listener Endpoint Registration](#)
- [Direct Dispatcher Connection](#)
- [Configuring SSL for EJB and CORBA](#)

Database Listeners and Dispatchers

[Figure 3-3](#) shows the interaction between the listener and the dispatchers, and also illustrates how an Oracle8i ORB session is activated.

Figure 3–3 Listener/Dispatcher Interaction

1. Upon database startup, the dispatcher registers itself with the listener.
2. The client invokes a method, giving the listener's URL address as the destination.
3. The listener sends back a `LOCATION_FORWARD` response to the client's ORB layer informing it of the dispatcher's address. This redirects the request to the appropriate dispatcher.

Note: The client is unaware of the redirection logic, which is performed by the ORB runtime layer that supports the client.

4. The underlying ORB runtime layer resends the initial request to the dispatcher. All future method invocations are directed to the dispatcher. The listener is no longer a part of the communication.

The incoming request is examined by the shared server services to see if the request is for an existing session. If so, the request is forwarded to the indicated session. If not, the service creates a new database session for the request and activates the ORB in the session. This session is very similar to the database sessions created for incoming Net8 connections. In the session, the ORB reads the incoming IIOP messages, authenticates the client, finds and activates the corresponding server-side

objects, and sends IIOp messages as needed to reply to the connected client. Subsequent messages from the client are directed to the existing session.

When you configure a listener, you need to configure separate ports as listening endpoints for both Net8 and IIOp connections. Similarly, if you want any endpoint to use the secure socket layer (SSL), you will also need a separate endpoint for an SSL-enabled IIOp endpoint. See ["Using the Secure Socket Layer"](#) on page 6-3 for more information about connecting using IIOp and SSL.

Handling Incoming Requests

The administrator for your database configures an MTS server with a dispatcher that is GIOP enabled. In addition, the administrator configures a listener that this dispatcher registers with upon database startup.

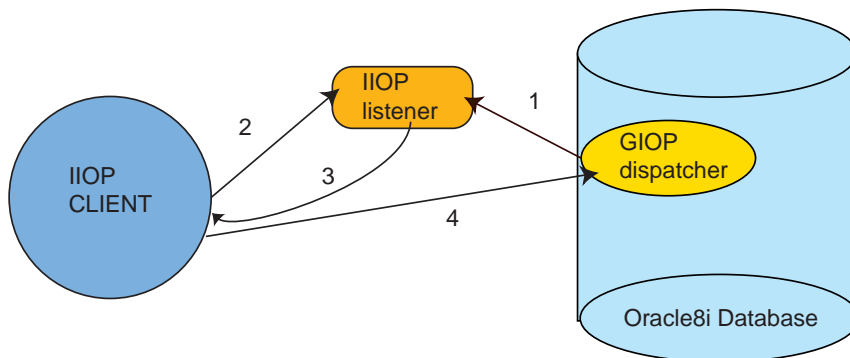
When the database does start up, all dispatchers register with all listeners configured within the same database initialization file. However, when an IIOp client invokes a request, the listener will only redirect the request to a GIOP dispatcher or hand off to a generic dispatcher.

These are discussed fully in the following sections:

- [Redirect to GIOP Dispatcher](#)
- [Hand Off to Generic Dispatcher](#)

Redirect to GIOP Dispatcher The listener recognizes the IIOp protocol and redirects the request to a registered GIOP dispatcher.

Figure 3–4 IIOp Listener Redirect to GIOP Dispatcher



1. GIOP dispatcher registers itself with the listener.
2. IIOP client—an EJB or CORBA client—invokes a method, giving the address of the listener. For redirection to occur, the listener must be statically configured to receive IIOP requests.
3. The listener sends back a response to the client informing it of the GIOP dispatcher's address.

The listener redirects if a GIOP dispatcher is configured. If no GIOP dispatcher is configured, the listener can hand off the request to a generic dispatcher. See ["Hand Off to Generic Dispatcher"](#) on page 3-14 for more information.

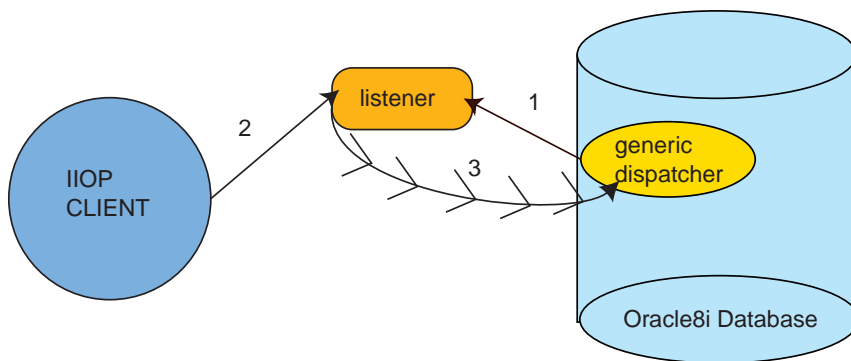
4. The underlying ORB runtime layer on the client resends its initial request to the GIOP dispatcher. All future method invocations are directed to the dispatcher. The listener is no longer a part of the communication.

Hand Off to Generic Dispatcher If there is no GIOP dispatcher statically configured, but there is a generic dispatcher configured, the listener can hand off the request to this dispatcher. The only restrictions are that the listener and dispatcher must exist on the same node in order for the hand off to occur and that the listener must be either statically or dynamically configured to receive an IIOP request.

For hand off to occur, the listener forfeits the socket to the dispatcher. Thus, this can only occur within a single node.

[Figure 3-5](#) shows the dispatcher and listener combination in a hand off environment.

Figure 3-5 *Hand Off to Dispatcher*



1. When the database starts, the generic dispatcher registers itself with the configured listener.

Note: The listener must be configured to receive IIOp requests. You can either statically configure the listener through the Net8 configuration, or you can dynamically configure the listener through a dynamic registration tool, `regep`. See "[Dynamic Listener Endpoint Registration](#)" on page 3-15 for more information.

2. The client sends a request to the listener.
3. The listener hands off the request to the generic dispatcher. The listener negotiates with the generic dispatcher on a separate channel. On this channel, the socket is handed off to the dispatcher through the operating system mechanisms.

The client communicates directly with the dispatcher from this point on. The client is never made aware that the socket was handed off.

Dynamic Listener Endpoint Registration

As discussed in "[Hand Off to Generic Dispatcher](#)" on page 3-14, a listener will hand off the socket to an existing generic dispatcher. In order for a hand off to occur for an IIOp incoming request, the listener must have an IIOp endpoint registered. You can register any listening endpoint through either of the following:

- static configuration—configured by the Net8 configuration tool
- dynamic configuration—registered by the dynamic registration tool, `regep`

The dynamic registration tool, `regep`, adds any type of listening endpoint to your listener. This includes an IIOp listening endpoint. The following describes how to use the dynamic registration tool for an IIOp listening endpoint.

The restrictions for this scenario are as follows:

- Both the listener and generic dispatcher always exist on the same node.
- A GIOP configured dispatcher cannot exist.

Note: If a GIOP configured dispatcher does exist, the listener will redirect the request, rather than hand off the request, to the configured dispatcher.

The advantage for dynamically registering a listener endpoint is that you do not need to restart your database for this listener to be IIOP enabled. The listening endpoint is active immediately.

For full details on the `regep` tool, see the *Oracle8i Java Tools Reference Guide*.

Example 3–1 Dynamically Registering a Listener at Port 2241

The following line dynamically registers a listener on the `SUNDB` host on endpoint port number 2241. This tool logs on to the `SUNDB` host.

```
regep -pres oracle.aurora.server.SGiopServer -host sundb -port 2241
```

Direct Dispatcher Connection

If you want your client to go to a dispatcher directly, bypassing the listener, you direct your client to the dispatcher's port number. Do one of the following to discover the dispatcher's port number:

- Configure a port number for the dispatcher by adding the `ADDRESS` parameter that includes a port number.
- Discover the port assigned to the dispatcher by invoking `lsnrctl service`.

If you choose to configure the port number, the following shows the syntax:

```
mts_dispatchers="(address=(protocol=tcp | tcps)
                  (host=<server_host>) (port=<port>))
                  (presentation=oracle.aurora.server.SGiopServer)"
```

The attributes are described below:

Attribute	Description
ADDRESS (ADD or ADDR)	Specifies the network address that the dispatchers will listen on. The network address may include either the TCP/IP (TCP) or the TCP/IP with SSL (TCPS) protocol, the host name of the server, and a GIOP listening port, which may be any port you choose that is not already in use.

Attribute	Description
PRESENTATION (PRE or PRES)	<p>Enables support for GIOP. Supply the following value for a GIOP presentation:</p> <ul style="list-style-type: none"> oracle.aurora.server.SGiopServer for session-based GIOP connections. This presentation is valid for TCP/IP and TCP/IP with SSL.

The client supplies the port number on its URL, as follows:

```
session_iiop://<hostname>/:<portnumber>
```

Notice that the URL excludes a SID or service name. The dispatcher does not need the SID instance or service name because it is a directed request.

Configuring SSL for EJB and CORBA

Oracle8i also supports the use of authentication data such as certificates and private keys required for use by SSL in combination with GIOP. To configure your transport to be SSL-enabled with GIOP, do the following:

1. Enable the MTS_DISPATCHERS to be SSL-enabled.
2. Specify the SSL wallet to be used when configuring both the listener and database.
3. Configure the listener to accept SSL.

The following sections detail how to accomplish these three steps.

Enable the MTS_DISPATCHERS for SSL

You must edit the database initialization file to add an SSL-enabled dispatcher. Uncomment the MTS_DISPATCHERS parameter in the database initialization file that defines the TCPS port. During installation, the Database Configuration Assistant always includes a commented out line for SSL TCP/IP. This line is as follows:

```
mts_dispatchers="(protocol=tcps)(presentation=oracle.aurora.server.SGiopServer)"
```

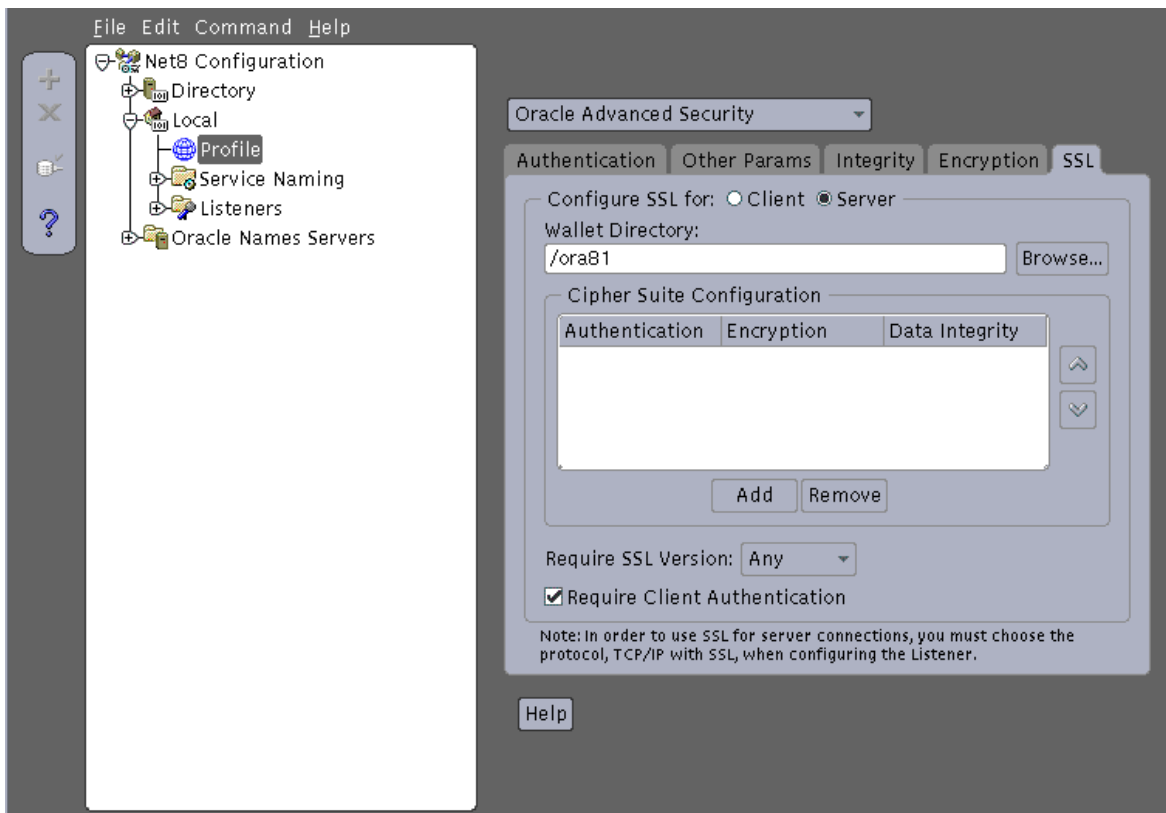
Configure the Wallet Location through Net8 Assistant

Modify the listener to accept SSL requests on port 2482.

1. Start Net8 Assistant.

- On UNIX, run `netasst` at `$ORACLE_HOME/bin`.
 - On Windows NT, choose Start > Programs > Oracle - *HOME_NAME* > Network Administration > Net8 Assistant.
2. In the navigator pane, expand Local > Profile.
 3. From the pull-down list, select Oracle Advanced Security > SSL.
- This brings you to the listening port panel, as shown in [Figure 3-6](#).

Figure 3-6 I/OP listening port configuration



4. On the "Configure SSL for:" line, select the "Server" radio button.
5. Under "Wallet Directory", enter the location for the wallet.

6. If you desire a certain SSL version, choose the appropriate version on the SSL version pulldown list.
7. If you want the client to authenticate itself by providing certificates, select the "Require Client Authentication" checkbox.
8. Choose File > Save Network Configuration.

These steps will add wallet and SSL configuration information into both the listener and database configuration files. The SSL wallet location must be specified in both the listener and database configuration files. Both entities must locate the wallet for certificate handshake capabilities.

The listener.ora file:

```
ssl_client_authentication=false
ssl_version=undetermined
```

The default is for the database to authenticate the client. If you want the listener to authenticate the client, change the `ssl_client_authentication` parameter to true.

The database's sqlnet.ora file:

```
ssl_client_authentication=true
ssl_version=0
sqlnet.crypto_seed=<seed_info>
```

If you did not request client authentication, the `ssl_client_authentication` parameter will be false. The default value is for client authentication to be true. In addition, you can specify a specific SSL version number, such as 3.0, in the `ssl_version` parameter. The `ssl_version` value of 0 means that the version is undetermined and will be agreed upon during handshake.

Within both the listener's `listener.ora` and database's `sqlnet.ora` files, the wallet location is specified:

```
oss.source.my_wallet=
  (source=
    (method=file)
    (method_data=
      (directory=wallet_location)))
```

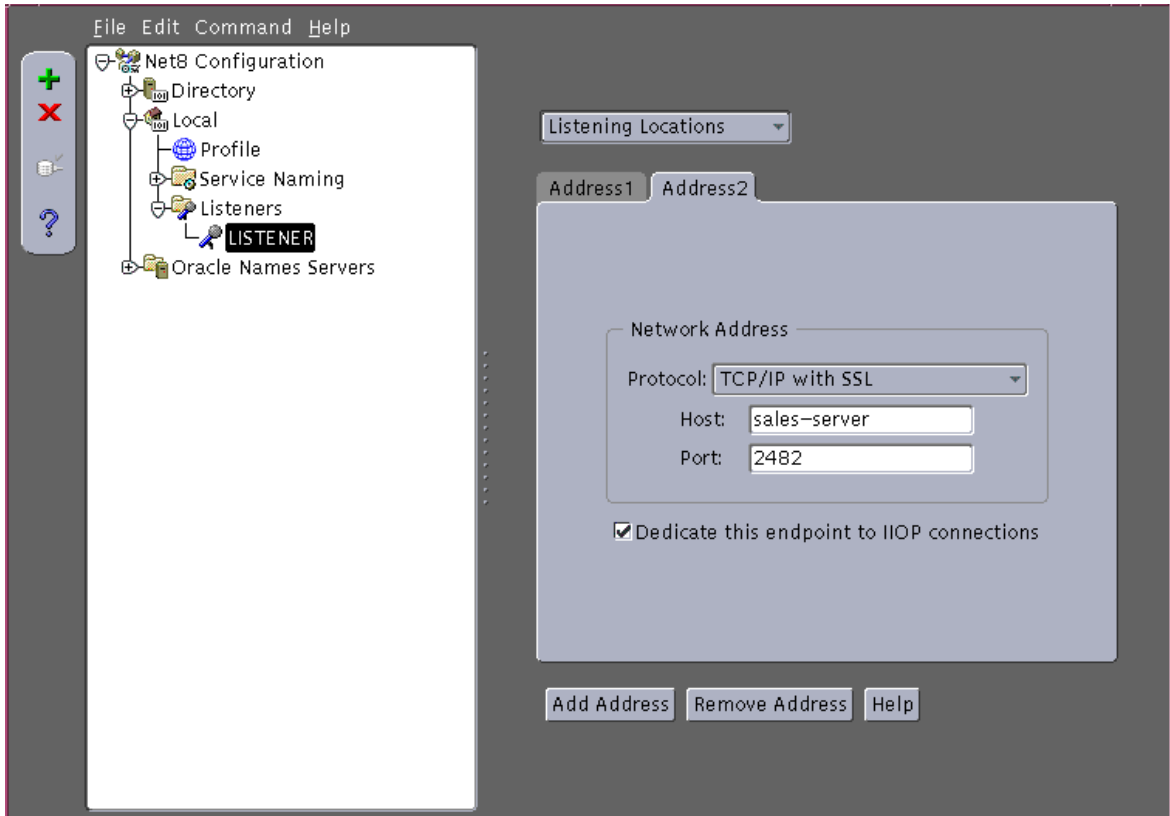
The *Oracle Advanced Security Administrator's Guide* describes how to set up the SSL wallet with the appropriate certificates.

Configure an SSL-Enabled Listener through Net8 Assistant

1. Back in the navigator pane, expand Local > Listener.

This brings you to the listener location panel, as shown in [Figure 3-2](#).

Figure 3-7 IIOIP listening port configuration



2. Select a listener.
3. From the list in the right pane, select Listening Locations. If none of the current listening addresses are feasible for your SSL listening address, you can add a new address by clicking on the "Add Address" button.
4. Select the TCP/IP with SSL protocol from the Protocol list.
5. Enter the host name of the database in the Host field.

6. Enter port 2482 in the Port field.
7. Click "Dedicate this endpoint to IIOP connections".
8. Choose File > Save Network Configuration.

This places the following in the `listener.ora` file: it modifies the listener to specify TCPS —instead of TCP—as the protocol with port number 2482. The following shows an example of an SSL-enabled listener on the `sales-server` host.

```
listener=
  (description_list=
    (description=
      (address=(protocol=tcps)(host=sales-server)(port=2482)))
      (protocol_stack=
        (presentation=giop)
        (session=raw)))
```

JNDI Connections and Session IOP Service

This chapter describes in detail how clients connect to an Oracle8i server session and how they authenticate themselves to the server. The term *client*, as used in this chapter, includes client applications and applets running on a network PC or a workstation, as well as distributed objects such as EJBs and CORBA server objects that are calling other distributed server objects and, thus, acting as clients to these objects.

In order to execute CORBA objects, you must first publish these objects in a Oracle8i database instance, using a CORBA CosNaming service. Then, you can retrieve the object reference either through a URL-based JNDI interface to CosNaming or straight to the CosNaming service. JNDI is recommended as it is easy for clients written in Java to locate and activate published objects.

In addition to authentication, this chapter discusses security of access control to objects in the database. A published object in the data server has a set of permissions that determine who can access and modify the object. In addition, classes that are loaded in the data server are loaded into a particular schema, and the person who deploys the classes can control who can use them.

This chapter covers the following topics:

- [JNDI Connection Basics](#)
- [The Name Space](#)
- [Execution Rights to Database Objects](#)
- [URL Syntax](#)
- [Using JNDI to Access Bound Objects](#)
- [Session IOP Service](#)

- [Retrieving JServer Version Number](#)
- [Activating In-Session CORBA Objects From Non-IIOP Presentations](#)
- [Accessing CORBA Objects Without JNDI](#)

JNDI Connection Basics

The client example in Chapter 2 showed how to connect to Oracle, start a database server session, and activate an object using a single URL specification. This was performed through the following steps:

```
1. Hashtable env = new Hashtable();
2. env.put(javax.naming.Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
3. env.put(javax.naming.Context.SECURITY_PRINCIPAL, username);
4. env.put(javax.naming.Context.SECURITY_CREDENTIALS, password);
5. env.put(javax.naming.Context.SECURITY_AUTHENTICATION,
           ServiceCtx.NON_SSL_LOGIN);
6. Context ic = new InitialContext(env);
7. myHello hello =
   (myHello) ic.lookup("sess_iiop://localhost:2481:ORCL/test/myHello");
8. System.out.println(hello.helloWorld());
```

In this example, there are four basic operations:

- Lines 1-5 set up an environment for the JNDI initial context.
- Line 6 creates the JNDI initial context.
- Line 7 looks up a published object. (See ["URL Syntax"](#) on page 4-5 for a discussion of the URL syntax.)
- Line 8 invokes a method on the object.

When a client looks up an object through the JNDI `lookup` method, the client and server automatically perform the following logic:

- A session IIOP connection is created to the ORCL instance of the local host database.
- The server establishes a database session.
- The client is authenticated, using the NON_SSL_LOGIN protocol, with the username and password specified in the environment context.
- The published object, `/test/myHello`, is located in the session namespace and a reference to it is returned to the client.

When the client invokes a method—such as `helloWorld()`—on the returned reference, the server activates the object in the server.

The Name Space

The name space in the database looks just like a typical file system. You can examine and manipulate objects in the publishing name space using the session shell tool. See the "sess_sh" tool in the *Oracle8i Tools Reference Guide* for information about the session shell.

There is a root directory, indicated by a forward slash (`/`). The root directory is built to contain three other directories: `bin`, `etc`, and `test`. The `/test` directory is where most objects are published for the example programs. You can create new directories under root to hold objects for separate projects; however, you must have access as database user `SYS` to create new directories under the root.

There is no effective limit to the depth that you can nest directories.

Note: The initial values in the publishing name space are set up when the JServer product for Oracle8i is installed.

The `/etc` directory contains objects the ORB uses. Do not delete objects in the `/etc` directory. The objects contained in `/etc` are:

`deployejb` `execute` `loadjava` `login` `transactionFactory`

The entries in the name space are represented by objects that are instances of the following classes:

- `oracle.aurora.AuroraServices.PublishingContext`—represents a class that can contain other objects (a directory)
- `oracle.aurora.AuroraServices.PublishedObject`—used for the leafs of the tree, that is the object names themselves.

These classes are documented in the JavaDoc on the product CD.

Published names for objects are stored in a database table. Each published object also has a set of associated permissions. Each class or resource file can have a combination (union) of the following permissions:

read The holder of read rights can list the class or the attributes of the class, such as its name, its helper class, and its owner.

write The holder of write for a context can bind new object names into a context. For an object (a leaf node of the tree), write allows the holder to republish the object under a different name.

execute You must have execute rights to resolve and activate an object represented by a context or published object name.

You use the `chmod` command of the session shell tool to view and change object rights.

Execution Rights to Database Objects

In addition to authentication and privacy, Oracle8i supports controlled access to the classes that make up CORBA and EJB objects. Only users or roles that have been granted execute rights to the Java class of an object stored in the database can activate the object and invoke methods on it.

You can control execute rights on Java classes with the following tools:

- At load time with the `-grant` argument to `loadjava`. See the *Oracle8i Java Developer's Guide* for more information about `loadjava` and execution rights on Java classes in the database.
- Using SQL commands—You use the SQL DDL `GRANT EXECUTE` command to grant execute permission on a Java class loaded in the database. For example, if SCOTT has loaded a class Hello, then SCOTT (or SYS) can grant execute privileges on that class to another user, say OTTO, by issuing the SQL command:

```
SQL> GRANT EXECUTE ON "Hello" TO OTTO;
```

Use the SQL command `REVOKE EXECUTE` to remove execute rights for a user from a loaded Java class.

- At publish time—Published objects are not restricted to a specific schema; they are potentially available to all users in the instance. Published objects have permissions that can differ from underlying classes. For example, if user SCOTT has execute permission on a published object name, but does not have execute permission on the class that the published object represents, SCOTT will not be able to activate the object.

You can control permissions on a published object through the following:

1. Using the `-grant` option with the `publish` tool.

2. Using the `chmod` and `chown` commands within the Session Shell. You must be connected to the Session Shell as the user `SYS` to use the `chown` command.

Use the `ls -l` command in the session shell to view the permissions (EXECUTE, READ, and WRITE) and the owner of a published object.

There are three "built-in" server objects that a client can access without being authenticated, as shown below:

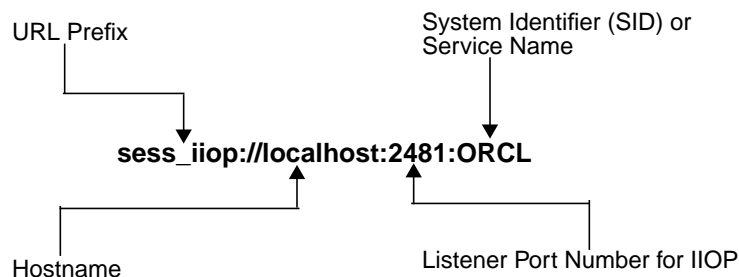
- the Name Service
- the `InitialReferences` object (the boot service)
- the `Login` object

You can activate these objects using `serviceCtx.lookup()` without authentication. See the "[Logging In and Out of the JServer Session](#)" on page 6-11 for an example that access the `Login` object explicitly.

URL Syntax

Oracle8i provides universal resource locator (URL) syntax to access services and sessions. The URL lets you use JNDI requests to start up services and sessions, and also to access components published in the database instance. An example service URL is shown in [Figure 4-1](#).

Figure 4-1 Service URL



The service URL is composed of four components:

1. The URL prefix followed by a colon and two slashes: `sess_iiop://` for a session IIOP request.

2. The system name (the hostname). For example: `myPC-1`. You can also use `localhost` or the numeric form of the IP address for the host.
3. The listener port number for IIOP services. The default is 2481.
4. The system identifier (SID)—for example, `ORCL`—or the service name—for example, `mySID.myDomain`.
 - **SID**—The system identifier is defined in your database initialization file as the `db_name`. This identifies the database instance you are connecting to. If you choose to add the SID to your service URL, the listener will load balance incoming requests across multiple dispatchers for the database instance.
 - **Service name**—The service name is equivalent to either the `service_name` or the `db_name.db_domain` parameters defined in your database initialization file. If you use the service name within your service URL, the listener will load balance incoming requests across multiple database instances: that is, all database instances registered with the listener. This options is good when you are using parallel servers.

Note: If you do use the service name, you must specify the `-useServiceName` flag on any tool that takes in the URL. If you do not specify this flag, the tool assumes that the last string is a SID.

Always use colons to separate the hostname, port, and SID or service name.

Note: If you specify a dispatcher port instead of a listener port, and you specify a SID, an `ObjectNotFoundException` is thrown by the server. Because applications that connect directly to dispatcher ports do not scale well, Oracle does not recommend direct connection to dispatchers.

URL Components and Classes

When you make a connection to Oracle and look up a published object using JNDI, you use a URL that specifies the service (service name, host, port, and SID), as well as the name of a published object to look up and activate. For example, a complete URL could look like:

```
sess_iiop://localhost:2481:ORCL/:default/projectAurora/Plans816/getPlans
```

where `sess_iiop://localhost:2481:ORCL` specifies the service name, `:default` indicates the default session (when a session has already been established), `/projectAurora/Plans816` specifies a directory path in the namespace, and `getPlans` is the name of a published object to look up.

Note: You do not specify the session name when no session has been established for that connection. That is, on the first look up there is no session active; therefore, `:default` as a session name has no meaning. In addition, `:default` is implied, so you can use a URL without a session name.

Each component of the URL represents a Java class. For example, the service name is represented by a `ServiceCtx` class instance, the session by a `SessionCtx` instance. See ["Using JNDI to Access Bound Objects"](#) and ["Session IOP Service"](#) starting on page 4-7 for more information on the service and session names within the URL.

CosNaming Restriction for JNDI Name

The JNDI bound name for the published object must use JNDI syntax rules. The underlying naming service that JServer JNDI uses is `CosNaming`. Thus, if your name includes a dot (".") in one of the names, the behavior diverges from normal `CosNaming` rules, as follows:

- The substring before the dot is treated as a `CosNaming NameComponent id`.
- The substring after the dot is treated as a `CosNaming NameComponent kind`.
- Both `id` and `kind` are concatenated into a full JNDI name.

Normally, in retrieving a `CosNaming` object, you supply the `id` and `kind` as separate entities. The JServer implementation concatenates both `id` and `kind`. Thus, to retrieve the object, your application refers to the full name with the dot included as part of the JNDI name, rather than as a separator.

Using JNDI to Access Bound Objects

Clients use the *Java Naming and Directory Interface* (JNDI) interface to look up published objects in the JServer namespace. JNDI is an interface supplied by Sun Microsystems that gives the Java application developer a methodology to access name and directory services. This section discusses only those parts of the JNDI API that are needed to look up and activate published objects. To obtain a complete set

of documentation for JNDI, see the web site URL
<http://java.sun.com/products/jndi/index.html>.

Note: It is also possible to access the session namespace without using JNDI. Instead, you can use CosNaming methods.

As described in "[URL Syntax](#)" on page 4-5, the JNDI URL required to access any bound name in the JServer namespace requires a compound name consisting of the following two components:

- Service name—the service name is used by JServer to retrieve a handle to the correct namespace.

Several namespaces will exist within your network. The service specifies which namespace to retrieve the JNDI bound object from. Service names can be one of the following:

Service Name	Description
<code>sess_iiop:// <hostname>:<port>:<SID></code>	Specifies the host, port, and SID that the desired namespace is located. Specifying this service name only without a session name returns a <code>ServiceCtx</code> object. The session IIOP service is the main service used by IIOP applications. It retrieves objects and object references bound in JNDI namespaces on different database hosts. See " Session IIOP Service " on page 4-13 for a full description.
<code>jdbc_access:</code>	Specifies that the desired object exists in a well-known namespace. Used primarily to retrieve JTA <code>UserTransaction</code> and <code>DataSource</code> objects from the namespace.
<code>java:</code>	Used to specify that the bound name is actually an EJB environment variable that was specified within its deployment descriptor.

- Session name—the actual JNDI bound name of the object within the designated namespace. The syntax mimics a UNIX file system syntax. The session name can be represented by a `SessionCtx` object.

You can utilize the service and session contexts to perform some advanced techniques, such as opening different sessions within a database or enabling several clients to access an object in a single session. These are discussed further in the

"[Session IOP Service](#)" on page 4-13. However, for simple JNDI lookup invocations, you should use the URL syntax specified in "[URL Syntax](#)" on page 4-5.

Importing JNDI Support Classes

When you use JNDI in your client or server object implementations, be sure to include the following import statements in each source file:

```
import javax.naming.Context;    // the JNDI Context interface
import javax.naming.InitialContext;
import oracle.aurora.jndi.sess_iop.ServiceCtx; // JNDI property constants
import java.util.Hashtable;    // hashtable for the initial context environment
```

Retrieving the JNDI InitialContext

`Context` is an interface in the `javax.naming` package that is used to retrieve the `InitialContext`. All Oracle8i EJB and CORBA clients use the `InitialContext` for `JNDI.lookup()`. Before you perform a `JNDI.lookup()`, you set the environment variables, such as authentication information into the `Context`. You can use a hash table or a properties list for the environment. Then, this information is made available to the naming service when the `lookup()` is performed. The examples in this guide always use a Java `Hashtable`, as follows:

```
Hashtable environment = new Hashtable();
```

Next, you set up properties in the hash table. You must always set the `Context` `URL_PKG_PREFIXES` property, whether you are on the client or server. The remaining properties are used for authentication, which primarily are used by clients or by a server authenticating itself as another user.

- `javax.naming.Context.URL_PKG_PREFIXES`
- `javax.naming.Context.SECURITY_PRINCIPAL`
- `javax.naming.Context.SECURITY_CREDENTIALS`
- `javax.naming.Context.SECURITY_ROLE`
- `javax.naming.Context.SECURITY_AUTHENTICATION`
- `USE_SERVICE_NAME`

URL_PKG_PREFIXES

`Context.URL_PKG_PREFIXES` holds the name of the environment property for specifying the list of package prefixes to use when loading in URL context factories.

The value of the property should be a colon-separated list of package prefixes for the class name of the factory class that will create a URL context factory.

In the current implementation, you must always supply this property in the Context environment, and it must be set to the String "oracle.aurora.jndi".

SECURITY_PRINCIPAL

`Context.SECURITY_PRINCIPAL` holds the database username.

SECURITY_CREDENTIALS

`Context.SECURITY_CREDENTIAL` holds the clear-text password. This is the Oracle database password for the `SECURITY_PRINCIPAL` (the database user). In all of the three authentication methods mentioned in `SECURITY_AUTHENTICATION` below, the password is encrypted when it is transmitted to the server.

SECURITY_ROLE

`Context.SECURITY_ROLE` holds the Oracle8i database role with which the user is connecting. For example, "CLERK" or "MANAGER".

SECURITY_AUTHENTICATION

`Context.SECURITY_AUTHENTICATION` holds the name of the environment property that specifies the type of authentication to use. Values for this property provide for the authentication types supported by Oracle8i. There are four possible values, which are defined in the `ServiceCtx` class:

- `ServiceCtx.NON_SSL_LOGIN`: The client authenticates itself to the server with a username and password using the Login protocol over a standard TCP/IP connection (not a secure socket layer connection). The Login protocol provides for encryption of the password as it is transmitted from the client to the server. The server does not authenticate itself to the client. See "[Providing Username and Password for Client-Side Authentication](#)" on page 6-9 for more information about this protocol.
- `ServiceCtx.SSL_CREDENTIAL`: The client authenticates itself to the server providing a username and password that are encrypted over a secure socket layer (SSL) connection. The server authenticates itself to the client by providing credentials.
- `SSL_LOGIN`: The client authenticates itself to the server with a username and password within the Login protocol, over an SSL connection. The server does not authenticate itself to the client.

- `SSL_CLIENT_AUTH`: Both the client and the server authenticate themselves to each other by providing certificates to each other over an SSL connection.

Note: To use an SSL connection, you must be able to access a listener that has an SSL port configured, and the listener must be able to redirect requests to an SSL-enabled database IIOP port. You must also include the following JAR files when you compile and build your application:

- If your client uses JDK 1.1, import `jssl-1_1.jar` and `javax-ssl-1_1.jar`.
 - If your client uses Java 2, import `jssl-1_2.jar` and `javax-ssl-1_2.jar`.
-

USE_SERVICE_NAME

If you are using a service name instead of an SID in the URL, you set this property to true. Otherwise, the last string in the URL must contain the SID. Given a `Hashtable` within the variable, `env`, the following designates that the service name is used instead of the SID within the URL:

```
Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(Context.SECURITY_PRINCIPAL, "scott");
env.put(Context.SECURITY_CREDENTIALS, "tiger");
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
env.put("USE_SERVICE_NAME", "true");
Context ic = new InitialContext(env);
```

The default is false.

The URL given within the lookup should contain a service name, instead of an SID. The following URL contains the service name, `orasun12`:

```
myHello hello =
    (myHello) ic.lookup("sess_iiop://localhost:2481:orasun12/test/myHello");
```

The JNDI InitialContext Methods

`InitialContext` is a class in the `javax.naming` package that implements the `Context` interface. All naming operations are relative to a context. The initial context implements the `Context` interface and provides the starting point for resolution of names.

Constructor

You construct a new initial context using the constructor:

```
public InitialContext(Hashtable environment)
```

It requires a `Hashtable` for the input parameter that contains the environment information described in ["Retrieving the JNDI InitialContext"](#) above. The following code fragment sets up an environment for a typical client, and creates a new initial context:

```
Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(Context.SECURITY_PRINCIPAL, "scott");
env.put(Context.SECURITY_CREDENTIALS, "tiger");
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
Context ic = new InitialContext(env);
```

lookup

This is the most common initial context class method that the CORBA or EJB application developer will use:

```
public Object lookup(String URL)
```

You use `lookup()` to retrieve an object instance or to create a new service context.

- To retrieve an object instance, specify a URL for the service name and append the JNDI bound name (the session name). The returned result must be cast to the expected object type. For example, to retrieve the `Hello` interface, you would do the following:

```
myHello hello =
    (myHello) ic.lookup("sess_iiop://localhost:2481:ORCL/test/myHello");
```

The service name is `"sess_iiop://localhost:2481:ORCL"`; the JNDI bound name for `Hello`'s home interface is `"/test/myHello"`.

- To retrieve a handle to a specific namespace, specify the desired service context. The return result must be cast to `ServiceCtx` when a new service context is being created. For example, if `initContext` is a JNDI initial context, the following statement creates a new service context:

```
ServiceCtx service =
    (ServiceCtx) initContext.lookup("sess_iiop://localhost:2481:ORCL");
```


See "[Session Management Scenarios](#)" on page 4-18 for examples on how to use the `JNDI.lookup` method within an EJB or CORBA application.

Session IIOp Service

All client/server network communications route requests over an accepted protocol between both entities. Most network communications to the Oracle8i database routed over the two-task common (TTC) layer. This is the service that processes incoming Net8 requests for database SQL services. However, with the addition of Java into the database, JServer required that clients communicate with the server over an IIOp transport that recognized database sessions. This is accomplished through the session IIOp service.

The session IIOp service is used for facilitating requests for IIOp applications, which includes CORBA and EJB applications. The following sections describe how to manage your applications within one or more database sessions:

- [Session IIOp Service Overview](#)
- [Session Management](#)
- [Service Context Class](#)
- [Session Context Class](#)
- [Session Management Scenarios](#)
- [Setting Session Timeout](#)

Session IIOp Service Overview

As discussed in the *Oracle8i Java Developer's Guide*, since the EJB is loaded into the database, your client application must start up the EJB within the context of a database session. Because beans are activated within a session, each client cannot see bean instances active in another session, unless given a handle to that session. Also, you can activate objects within the existing session or another session.

The session IIOp service session component tag—`TAG_SESSION_IIOp`—exists inside the IIOp profile—`SessionIIOp`. The value for this Oracle session IIOp component tag is `0x4f524100` and contains information that uniquely identifies the session in which the object was activated. The client ORB runtime uses this information to send requests to objects in a particular session.

Although the Oracle8i session IIOB service provides an enhancement of the standard IIOB protocol—it includes session ID information—it does not differ from standard IIOB in its on-the-wire data transfer protocol.

Client Requirements

Clients must have an ORB implementation that supports session IIOB to be able to access objects in different sessions simultaneously, from within the same program, and to be able to disconnect from and reconnect to the same session. The version of the Visigenic ORB that ships with Oracle8i has been extended to support session IIOB.

Session Routing

When a client makes an IIOB connection to the database, Oracle8i decides if a new session should be started to handle the request, or if the request should be routed to an existing session. If the client initializes a new request for a connection (using the `InitialContext.lookup()` method) and no session is active for that connection, a new session is automatically started. If a session has already been activated for the client, the session identifier is encoded into the object key of the object. This information enables the session IIOB service to route the request to the correct session. In addition, you can also use this session identifier to allow a single client to access multiple sessions. See "[Session Management Scenarios](#)" on page 4-18 for more information.

JServer Tools

When using the Oracle8i JServer tools, especially when developing EJB and CORBA applications, it is very important to distinguish the two network service protocol types: TTC and IIOB.

Figure 4–2 *TTC and IIOB Services*

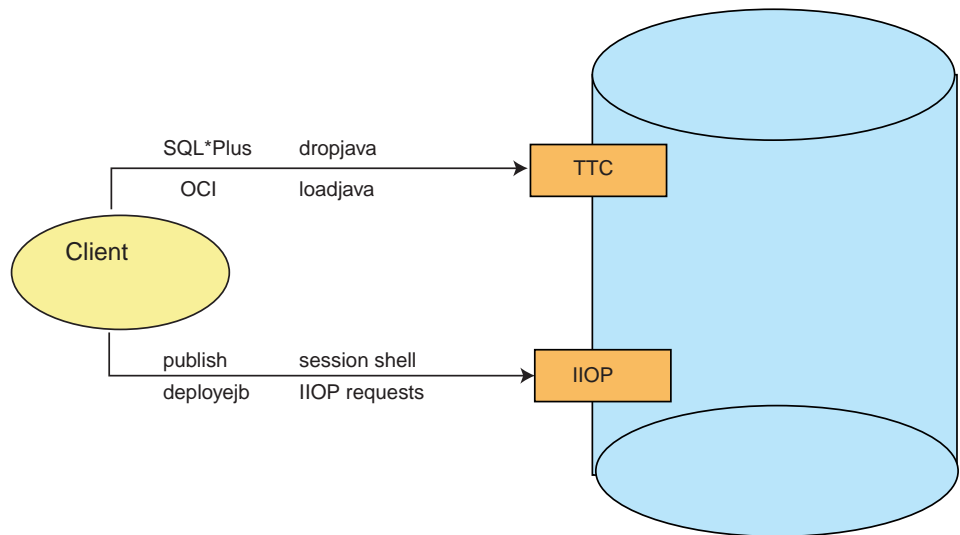


Figure 4–2 shows which tools and requests use TTC and which use IIOB database ports. 1521 is the default port number for TTC, and 2481 is the default for IIOB.

- Tools such as `publish`, `deployejb`, and the session shell access IIOB objects, and so must connect using an IIOB port. In addition, EJB and CORBA clients must use an IIOB port when sending requests to Oracle.
- Tools such as `loadjava` and `dropjava` connect using a TTC port.

Session Management

In the simple cases, a client (or a server object acting as a client) starts a new server session implicitly when it performs the lookup for a server object. Oracle8i also gives you the ability to control session start-up explicitly. Two Oracle-specific classes are provided that gives you control over the session IIOB service connection and over the sessions within the database.

- **Service Context Class**—Controls the session IIOB service connection to the database. Given a URL to that database, you can create a service context. Off of this service context, you can open one or more named sessions within the database.

- **Session Context Class**—Controls named database sessions created off of a service context. Once created, you can activate CORBA or EJB objects within the session using the named session context object.

Service Context Class

Controls the session IIOp service connection to the database. Given a URL to that database, you can create a service context. Off of this service context, you can open one or more named sessions within the database. This Oracle-specific class extends the JNDI `Context` class.

Variables

The `ServiceCtx` class defines a number of final public static variables that you can use to define environment properties and other variables. [Table 4-1](#) shows these.

Table 4-1 *ServiceCtx Public Variables*

String Name	Value
NON_SSL_CREDENTIAL	"Credential"
NON_SSL_LOGIN	"Login"
SSL_CREDENTIAL	"SecureCredential"
SSL_LOGIN	"SecureLogin"
SSL_CLIENT_AUTH	"SslClientAuth"
SSL_30	"30"
SSL_20	"20"
SSL_30_WITH_20_HELLO	"30_WITH_20_HELLO"
Integer Name	Integer Constructor
SESS_IIOp	new Integer(2)
IIOp	new Integer(1)

Methods

The public methods in this class that CORBA and EJB application developers can use are as follows:

```
public Context createSubcontext(String name)
```

This method takes a Java String as the parameter and returns a JNDI Context object representing a session in the database. The method creates a new named session. The parameter is the name of the session to be created, which must start with a colon (:).

The return result should be cast to a SessionCtx object.

Throws `javax.naming.NamingException`.

```
public Context createSubcontext(Name name)
```

(Each of the methods that takes a `String` parameter has a corresponding method that takes a `Name` parameter. The functionality is the same.)

```
public static org.omg.CORBA.ORB init(String username,
                                   String password,
                                   String role,
                                   boolean ssl,
                                   java.util.Properties props)
```

Gets access to the ORB created when you perform a look up. Set the `ssl` parameter to `true` for SSL authentication. Clients that do not use JNDI to access server objects should use this method.

See "sharedsession" in the demos installed in the `demo/examples/corba/basic` directory.

```
public Object lookup(String string)
```

`lookup()` looks up a published object in the database instance associated with the service context, and either returns an activated instance of the object, or throws `javax.naming.NamingException`.

Session Context Class

Controls named database sessions created off of a service context. Once created, you can activate CORBA or EJB objects within the session using the named session context object. Session contexts represent sessions and contain methods that enable you to perform session operations such as authenticating the client to the session or activating objects. This class extends the JNDI Context class.

Note: Creating a subcontext within the session context affects the object type returned on the final JNDI lookup. See "[Lookup of Objects Off of JNDI Context](#)" on page 4-25 for more information.

Methods

The session context methods that a client uses are the following:

```
public synchronized boolean login()
```

`login()` authenticates the client using the initial context environment properties passed in the `InitialContext` constructor: `username`, `password`, and `role`.

```
public synchronized boolean login(String username,  
                                  String password,  
                                  String role)
```

`login()` authenticates the client using the `username`, `password`, and optional database `role` supplied as parameters.

```
public Object activate(String name)
```

Looks up and activates a published object having the name.

Session Management Scenarios

The following sections describe the different scenarios for managing database sessions:

- [Client Accessing a Single Session](#)—client activates and accesses an object in the `:default` session.
- [Ending a Session](#)—discusses methods that explicitly terminate a session.
- [Client Starting a Named Session](#)—client activates and accesses one or more objects in a session other than the `:default` session. This session is identified by a name within a `SessionCtx`.
- [Two Clients Accessing the Same Session](#)—two or more clients can access an activated object within a session by providing `x` and `y` to both clients.
- [In-Session Activation](#)—a server object, acting as a client, activates another object within the same session.

- **Lookup of Objects Off of JNDI Context**—lookup of a partial JNDI name requires that you activate the bound object.

Client Accessing a Single Session In general, when you look up a published object from a client with a URL, hostname, and port, the object is activated in a new session. For example, a client would perform the following:

```
Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(Context.SECURITY_PRINCIPAL, "scott");
env.put(Context.SECURITY_CREDENTIALS, "tiger");
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
Context ic = new InitialContext(env);
SomeObject myObj =
    (SomeObject) ic.lookup("sess_iiop://localhost:5521:ORCL/test/myObj");
```

Activating an object in a new session from a server object is identical to starting a session from an application client. If the `lookup` method is invoked within the server object, the second object instance is activated in a separate session from the originating session.

Ending a Session Normally, a session terminates when the client terminates. However, if you want to explicitly terminate a session, you can do one of the following:

Terminate A Session From The Server-Side Using The Endsession Method

The server can control session termination by executing the following method:

```
oracle.aurora.mts.session.Session.THIS_SESSION().endSession();
```

Terminate A Session From The Client-side Using The Logout Object

If the client wishes to exit the session, it can execute the `logout` method of the `LogoutServer` object, which is pre-published as `/etc/logout`. Only the session owner is allowed to logout. Any other owner receives a `NO_PERMISSION` exception.

The `LogoutServer` object is analogous to the `LoginServer` object, which is pre-published as `/etc/login`. You can use the `LoginServer` object to retrieve the `Login` object, which is used to authenticate to the server. This is an alternative method to using the `Login` object within the JNDI lookup.

The following example shows how a client can authenticate using the `LoginServer` object and can exit the session through the `LogoutServer` object.

```
import oracle.aurora.AuroraServices.LoginServer;
```

```
import oracle.aurora.AuroraServices.LoginServer;
...
// To log in using the LoginServer object
LoginServer loginServer = (LoginServer)ic.lookup(serviceURL + "/etc/login");
Login login = new Login(loginServer);
System.out.println("Logging in ..");
login.authenticate(user, password, null);
...
//To logout using the LogoutServer
LogoutServer logout = (LogoutServer)ic.lookup(serviceURL + "/etc/logout");
logout.logout();
```

Client Starting a Named Session You can explicitly create multiple session on the database instance through the JNDI methods provided in the `ServiceCtx` and `SessionCtx` classes.

The following `lookup` method contains a URL that defines the IOP service URL of `"sess_iiop://localhost:5521:ORCL"` and a default session context.

```
SomeObject myObj =
    (SomeObject) ic.lookup("sess_iiop://localhost:5521:ORCL/test/myHello");
```

In this simple case, the JNDI initial context `lookup` method implicitly starts a session and authenticates the client. This session becomes the default session, which is identified by the name `":default"`. All sessions are named. However, in the default case, the client does not need to know the name of the session, because all requests go to this single session. Unless specified, all additional objects activated will be activated in the default session. Even if you create a new JNDI initial context and look up the same or a new object, the object is instantiated in the same session as the first object.

The only way to activate objects within another session is to create a named session. You can create other sessions in place of or in addition to the default session by creating session contexts off of the service context. Each session would be a named session, so that you can activate objects in different sessions within the database.

1. Instantiate a new hashtable for the environment properties to be passed to the server.

```
Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
```

Note: Only the `URL_PKG_PREFIXES` Context variable is filled in—the other information will be provided in the `login.authenticate()` method parameters.

2. Create a new JNDI Context.

```
Context ic = new InitialContext(env);
```

3. Use the JNDI `lookup` method on the initial context, passing in the service URL, to establish a service context. This example uses a service URL with the service prefix of hostname, listener port, and SID.

Note: Provide only the service URL of hostname, listener port, and database SID. If you provide the JNDI name of the desired object, a default session will be created for you.

```
ServiceCtx service =
    (ServiceCtx) ic.lookup("sess_iiop://localhost:2481:ORCL");
```

4. Create a session by invoking the `createSubcontext` method on the service context object. Provide the name for the session as a parameter to the `createSubcontext` method. A new session is created within the database.

```
SessionCtx session = (SessionCtx) service.createSubcontext(":session1");
```

Note: You must name a new session when you create it. The session name must start with a colon (:), and cannot contain a slash (/), but is not otherwise restricted.

5. Authenticate the client program to the database by invoking the `login` method on the session context object.

```
session.login("scott", "tiger", null); // role is null
```

6. Activate the object, identified by its bound JNDI name, in the named session.

```
Hello hello = (Hello)session.activate (objectName);
```

```
System.out.println (hello.helloWorld ());
```

Example 4-1 Activating Objects in Named Sessions

The following example creates two named sessions of the name `:session1` and `:session2`. Each one retrieves the `Hello` object separately. The client invokes both `Hello` objects in each named session.

```
Hashtable env = new Hashtable ();
env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
Context ic = new InitialContext (env);

// Get a SessionCtx that represents a database instance
ServiceCtx service = (ServiceCtx) ic.lookup ("sess_iiop://localhost:2481:ORCL");

// Create and authenticate a first session in the instance.
SessionCtx session1 = (SessionCtx) service.createSubcontext (":session1");

// Authenticate
session1.login("scott", "tiger", null);

// Create and authenticate a second session in the instance.
SessionCtx session2 = (SessionCtx) service.createSubcontext (":session2");

// Authenticate using a login object (not required, just shown for example).
LoginServer login_server2 = (LoginServer)session2.activate ("/etc/login");
Login login2 = new Login (login_server2);
login2.authenticate ("scott", "tiger", null);

// Activate one Hello object in each session
Hello hello1 = (Hello)session1.activate (objectName);
Hello hello2 = (Hello)session2.activate (objectName);

// Verify that the objects are indeed different
System.out.println (hello1.helloWorld ());
System.out.println (hello2.helloWorld ());
```

Two Clients Accessing the Same Session When the client invokes the JNDI lookup method, JServer creates a session. If you want a second client to access the instantiated object in this session, you must do the following:

1. The first client saves both the object instance handle and a `Login` object reference.
2. The second client retrieves the handle and `Login` object reference and uses them to access the object instance.

Example 4–2 Two Clients Accessing a Single Instance

1. The first client authenticates itself to the database by providing a username and password through the `authenticate` method on a `Login` object.
2. The session is created and the object is instantiated through the `lookup` method given the URL.
3. Both the `LoginServer` object and the server object instance handle are saved to a file for the second client to retrieve.

```
// Login to the 8i server
LoginServer lserver = (LoginServer)ic.lookup (serviceURL + "/etc/login");
new Login (lserver).authenticate (username, password, null);

// Activate a Hello in the 8i server
// This creates a first session in the server
Hello hello = (Hello)ic.lookup (serviceURL + objectName);
hello.setMessage ("As created by Client1");
System.out.println ("Client1: " + hello.helloWorld ());

// save Login object into a file, loginFile, for Client2 to read
com.visigenic.vbroker.orb.ORB orb = oracle.aurora.jndi.orb_dep.Orb.init();
String log = orb.object_to_string (lserver);
OutputStream os = new FileOutputStream (loginFile);
os.write (log.getBytes ());
os.close ();

// save object instance handle into a file, helloFile,
// for Client2 to read
String obj_hndl = orb.object_to_string (hello);
OutputStream os = new FileOutputStream (helloFile);
os.write (obj_hndl.getBytes ());
os.close ();
```

The second client would access the `Hello` object instance in the active session by doing the following:

1. Retrieve the object handle and the `Login` object. This example uses implementation-defined methods of `readHandle` and `readLogin` to retrieve these objects from storage.
2. Authenticate to the database session with the same `Login` object as the first client through the `authenticate` method. You can recreate the `Login` object from the `LoginServer` object through the `Login` constructor.

```
FileInputStream finstream = new FileInputStream (hellofile);
```

```
ObjectInputStream istream = new ObjectInputStream (finstream);
Hello hello = (Hello) orb.string_to_object(istream.readObject());
finstream.close ();

// Authenticate with the login Object
LoginServer lserver = (LoginServer) readLogin(loginFile);

//Set the VisiBroker bind options to specify that the
//login is to not try recursively, which means that if it
//fails on the first try, return with the error immediately.
//See VisiBroker manuals for more information.
lserver._bind_options (new BindOptions (false, false));

Login login = new Login (lserver);
login.authenticate (username, password, null);
```

In-Session Activation If the server object wants to look up and activate a new published object in the same session in which it is running, the server object can execute the following:

```
Context ic = new InitialContext( );
SomeObject myObj = (SomeObject) ic.lookup("/test/Hello");
```

Notice that there are no environment settings for authentication information in the environment or a session URL in the lookup. The authentication already succeeded in order to log into the session. Plus, the object exists on the local machine. So, any other object activation within the session can proceed without specifying authentication information or a target `sess_iiop` URL address.

Note: In-session activation as demonstrated in this section is valid for both IIOP and non-IIOP clients.

In-Session Activation in Pre-8.1.7 Releases In releases previous to Release 8.1.7, in-session activation was performed with the `thisServer/:thisSession` notation in place of the `hostname:port:SID` in the URL. This notation is still valid, but only for IIOP clients.

For example, to look up and activate an object in the same session, do the following:

```
Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
Context ic = new InitialContext(env);
SomeObject myObj =
    (SomeObject) ic.lookup("sess_iiop://thisServer/:thisSession/test/Hello");
```

In this case, `myObj` is activated in the same session in which the invoking object is running. Note that there is no need to supply login authentication information, as the client (a server object in this case) is already authenticated to Oracle8i.

Realize that objects are not authenticated, instead, clients must be authenticated to a session. However, when a separate session is to be started, then some form of authentication must be done—either login or SSL credential authentication.

Note: You can only use the `thisServer` notation on the server side, that is, from server objects. You cannot use it in a client program.

Lookup of Objects Off of JNDI Context In the Sun Microsystems JNDI, if you bind a name of `"/test/myObject"`, you can retrieve an object from a `Context` when executing the following:

```
Context ctx = ic.lookup("/test");
MyObject myobj = ctx.lookup("myObject");
```

The returned object is activated and ready for you to perform method invocations off of it.

In Oracle8i, trying to retrieve an object from a `Context` results in an inactive object being returned. Instead, you must do the following:

1. Retrieve a `SessionCtx`, instead of a `Context`. You can retrieve the `SessionCtx` from a `ServiceCtx`, in one of the two following ways:

- Retrieve the `ServiceCtx` first and the `SessionCtx` from the `ServiceCtx`, as follows:

```
ServiceCtx service =
    (ServiceCtx) ic.lookup("sess_iiop://localhost:2481:ORCL");
//Retrieve the ServiceCtx subcontext
SessionCtx sess = (SessionCtx) service.lookup("/test");
```

- Retrieve the `ServiceCtx` and `SessionCtx` in the same lookup, as follows:

```
SessionCtx sess =
    (SessionCtx) ic.lookup("sess_iiop://localhost:2481:ORCL/test");
```

2. Execute the Oracle-specific `SessionCtx.activate` method for each object in the session that you want to retrieve. This method activates the object in the session and returns the object reference. You cannot just perform a lookup of the object, as it will return an inactive object. Instead, execute the `activate` method, as follows:

```
MyObject myObj = (MyObject) sessCtx.activate("myObject");  
// Verify that the objects are indeed different  
System.out.println (myObj.printMe ());
```

The JServer JNDI implementation provides two implementations of the `Context` object:

- `ServiceCtx`—identifies the database instance through a `sess_iiop` URL
- `SessionCtx`—represents database session within the database

In performing a lookup, you must lookup both the `ServiceCtx` for identifying the database and the `SessionCtx` for retrieving the actual JNDI bound object. Normally, you supply the URLs for both objects within the JNDI URL given to the `lookup` method. However, you can also retrieve each individually as demonstrated above.

Setting Session Timeout

A session—with its state—normally exits when the last connection terminates. However, there are situations where you may want a session and its state to idle for a specified amount of time after the last connection terminates, such as the following:

- A middle-tier layer does not want to keep connections open to the session because connections are expensive; but, the middle-tier may want to keep the session open in case of another incoming client request.
- If you experience a network problem that abnormally terminates the connection, the session will stay around for the specified amount of time to allow the connection to be re-established.
- If your application passes a handle to an existing object within the session to another client before its connection terminates, the second client has time to access the session.

The timeout clock starts when the last connection to the session terminates. If another connection to the session starts within the timed window, the timeout clock is reset. If not, the session exits.

You can set the session idle timeout either from the client or from within a server object:

- [Set the Session Timeout from the Client](#)
- [Set the Session Timeout from a Server Object](#)

Set the Session Timeout from the Client

You can set the idle timeout on the client through the pre-published utility object—`oracle.aurora.AuroraServices.Timeout`. This object is pre-published under `/etc/timeout`. Use the `setTimeout` method from this object.

1. Retrieve the `Timeout` object through a JNDI lookup of `/etc/timeout`
2. Set the timeout with the `setTimeout` method giving the number of seconds for session idle.

```
Timeout timeout = (Timeout)ic.lookup(serviceURL + "/etc/timeout");
System.out.println("Setting a timeout of 20 seconds ");
timeout.setTimeout(20);
```

Set the Session Timeout from a Server Object

A server object can control the session timeout by using the `oracle.aurora.net.Presentation` object, which contains the `sessionTimeout()` method. This method takes one parameter; the session timeout value in seconds. For example:

```
int timeoutValue = 30;
...
// set the timeout to 30 seconds
oracle.aurora.net.Presentation.sessionTimeout(timeoutValue);
...
// set the timeout to a very long time
oracle.aurora.net.Presentation.sessionTimeout(Integer.MAX_INT);
```

Note: When you use the `sessionTimeout()` method, you must add `$(ORACLE_HOME)/javavm/lib/aurora.zip` to your CLASSPATH.

Retrieving JServer Version Number

You can retrieve the version of JServer that is installed in the database through the pre-published `oracle.aurora.AuroraServices.Version` object, which is published as `/etc/version` in the JNDI namespace. The `Version` object contains the `getVersion` method, which returns a string that contains the version, such as "8.1.7". You can retrieve the `Version` object by providing `/etc/version` within the JNDI lookup. The following example retrieves the version number:

```
Version version = (Version)ic.lookup(serviceURL + "/etc/version");
System.out.println("The server version is : " + version.getVersion());
```

Activating In-Session CORBA Objects From Non-IIOP Presentations

Non-IIOP server requests, such as HTTP or DCOM, can activate a CORBA object within the same session.

- HTTP An HTTP client interacts with the JServer webserver and executes a JSP or servlet, which can activate the CORBA object within the same session that it is running in.
- DCOM A DCOM client uses a DCOM bridge to access JServer. While within the JServer session, the DCOM bridge session can activate the CORBA object within the same session that it is running in.

If the non-IIOP server object wants to look up and activate a new published object in the *same session* in which it is running, the server object can execute the following:

```
Context ic = new InitialContext( );
SomeObject myObj = (SomeObject) ic.lookup("/test/Hello");
```

Note: Once you retrieve the IIOP object reference through this method, you cannot pass this object to a remote client or server.

Notice that there are no environment settings for authentication information in the environment or a URL specified in the lookup. The authentication already succeeded in order to log into the session. Plus, the object exists on the local

machine. So, any other object activation within the session can proceed without specifying authentication information or a target URL address.

Accessing CORBA Objects Without JNDI

It is possible for clients to access server objects without using the JNDI classes shown in the other sections of this chapter. These clients can connect to an Oracle server by using `CosNaming` methods.

Retrieving the NameService Initial Reference

In order to use the CORBA ORB methods, you must first retrieve the naming service object. Oracle8i prepublishes a `NameService` object that you can retrieve through the ORB `resolve_initial_references` method.

In CORBA, there are two methods to retrieve the `NameService` initial reference: using `ORBInitRef` or `ORBDefaultInitRef`. At this time, we have provided only the `ORBDefaultInitRef` methodology.

You must provide a service URL to the `ORBDefaultInitRef` of the form of host, port, and SID. Or you can provide the service URL with host, port, service name. In addition, you can specify some optional properties, such as:

- The connection should use SSL, set the `ORBUseSSL` property to true:

```
System.setProperty("ORBUseSSL", "true");
```

- The transport type, which can be `sess_iiop` or `iiop`. Set the `TRANSPORT_TYPE` property, as follows:

```
System.setProperty("TRANSPORT_TYPE", "sess_iiop");
```

- If retrieving the `NameService` without first accessing the `BootService`, set the backward compatible property (`ORBNameServiceBackCompat`) to false, as follows:

```
System.setProperty("ORBNameServiceBackCompat", "false");
```

- Use the service name instead of the SID in the service URL. Set the `USE_SERVICE_NAME` property to true, as follows:

```
System.setProperty("USE_SERVICE_NAME", "true");
```

Note: You initialize the server URL either through the `ORBDefaultInitRef` or through the individual properties: `ORBBootHost`, `ORBBootPort`, and `ORACLE_SID`.

Example 4–3 Retrieving a Server Object Using CosNaming

The following example demonstrates how to retrieve the `NameService` object. From this object, the login is executed and the server object is retrieved.

```
import java.lang.Exception;

import org.omg.CORBA.Object;
import org.omg.CORBA.SystemException;
import org.omg.CosNaming.NameComponent;

import oracle.aurora.client.Login;
import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.AuroraServices.LoginServerHelper;
import oracle.aurora.AuroraServices.PublishedObject;
import oracle.aurora.AuroraServices.PublishingContext;
import oracle.aurora.AuroraServices.PublishedObjectHelper;
import oracle.aurora.AuroraServices.PublishingContextHelper;

import Bank.Account;
import Bank.AccountManager;
import Bank.AccountManagerHelper;

public class Client {
    public static void main(String args[]) throws Exception {
        // Parse the args
        if (args.length < 4 || args.length > 5 ) {
            System.out.println ("usage: Client host port username password <sid>");
            System.exit(1);
        }
        String host = args[0];
        String port = args[1];
        String username = args[2];
        String password = args[3];
        String sid = null;
        if(args.length == 5)
            sid = args[4];

        // Declarations for an account and manager
        Account account = null;
```

```

AccountManager manager = null;
PublishingContext rootCtx = null;

// access the Aurora Names Service
try {
    // Initialize the ORB
    // The service URL for the server is provided in a string
    // that is prefixed with 'iioploc://' and includes either
    // host, port, sid or, if the USE_SERVICE_NAME is set to true,
    // host, port, service_name. This example uses host, port, sid
    // and sets it in the ORBDefaultInitRef.
    String initref;
    initref = (sid == null) ? "iioploc://" + host + ":" + port :
        "iioploc://" + host + ":" + port + ":" + sid;
    System.setProperty("ORBDefaultInitRef", initref);

    /*
     * Alternatively, you can set the host, port, sid or service in the
     * following individual properties. If set, these properties
     * take precedence over the URL set within the ORBDefaultInitRef property
    System.setProperty("ORBBootHost", host);
    System.setProperty("ORBBootPort", port);
    if (sid != null)
        //set the SID. alternatively, if the USE_SERVICE_NAME property is
        //true, this should contain the service name instead of the sid.
        System.setProperty("ORACLE_SID", sid);
     */

    /*
     * Some of the other properties that you can set
     * include the backwards compatibility flag, the service name
     * indicator, the SSL protocol definition, and the transport type.
    System.setProperty("ORBNameServiceBackCompat", "false");
    System.setProperty("USE_SERVICE_NAME", "true");
    System.setProperty("ORBUseSSL", "true");
    //transport type can be either sess_iiop or iiop
    System.setProperty("TRANSPORT_TYPE", "sess_iiop");
     */

    //initialize the ORB
    com.visigenic.vbroker.orb.ORB orb =
        oracle.aurora.jndi.orb_dep.Orb.init();

    // Get the Name service Object reference with the
    // resolve_initial_references method

```

```

rootCtx = PublishingContextHelper.narrow(orb.resolve_initial_references(
    "NameService"));

//After retrieving the NameService initial reference, you must perform
// the login, as follows:
// Get the pre-published login object reference
PublishedObject loginPubObj = null;
LoginServer serv = null;
NameComponent[] nameComponent = new NameComponent[2];
nameComponent[0] = new NameComponent ("etc", "");
nameComponent[1] = new NameComponent ("login", "");

// Lookup this object in the Name service
Object loginCorbaObj = rootCtx.resolve (nameComponent);

// Make sure it is a published object
loginPubObj = PublishedObjectHelper.narrow (loginCorbaObj);

// create and activate this object (non-standard call)
loginCorbaObj = loginPubObj.activate_no_helper ();
serv = LoginServerHelper.narrow (loginCorbaObj);

// Create a client login proxy object and authenticate to the DB
Login login = new Login (serv);
login.authenticate (username, password, null);

// Now create and get the bank object reference
PublishedObject bankPubObj = null;
nameComponent[0] = new NameComponent ("test", "");
nameComponent[1] = new NameComponent ("bank", "");

// Lookup this object in the name service
Object bankCorbaObj = rootCtx.resolve (nameComponent);

// Make sure it is a published object
bankPubObj = PublishedObjectHelper.narrow (bankCorbaObj);

// create and activate this object (non-standard call)
bankCorbaObj = bankPubObj.activate_no_helper ();
manager = AccountManagerHelper.narrow (bankCorbaObj);

account = manager.open ("Jack.B.Quick");

float balance = account.balance ();
System.out.println ("The balance in Jack.B.Quick's account is $"

```

```

        + balance);
    } catch (SystemException e) {
        System.out.println ("Caught System Exception: " + e);
        e.printStackTrace ();
    } catch (Exception e) {
        System.out.println ("Caught Unknown Exception: " + e);
        e.printStackTrace ();
    }
}
}
}

```

See ["Ending a Session"](#) on page 4-19 for more information on the LoginServer, Login, and LogoutServer objects.

Retrieving Initial References from ORBDefaultInitRef

CORBA 2.3 Interoperable Name Service supports both the ORBInitRef and ORBDefaultInitRef methodologies for creating and retrieving initial references. At this time, Oracle8i only supports an IIOP URL scheme within the ORBDefaultInitRef, as shown in ["Retrieving the NameService Initial Reference"](#) on page 4-29. You can only provide either a host, port, SID or host, port, service name combination—prefixed by "iioploc://"—to the ORBDefaultInitRef for locating the initial reference. Within this location, the service must have been activated. Any service activated within the specified location can be retrieved using the `resolve_initial_references` method with its object key, which is defined at the time of activation.

For example, if you set the ORBDefaultInitRef to the following server URL:

```
System.setProperty("ORBDefaultInitRef", "iioploc://myHost:myPort:mySID);
```

Then, initialize the ORB and retrieve your service, as follows:

```
//initialize the ORB
com.visigenic.vbroker.orb.ORB orb = oracle.aurora.jndi.orb_dep.Orb.init();

// Get the myService service Object reference with resolve_initial_references
rootCtx = PublishingContextHelper.narrow(orb.resolve_initial_references(
    "myService"));

```

The object key that is used to retrieve the service is "myService". The object with this key is returned with the `resolve_initial_references` method.

The following are the Oracle8i services that are activated during startup:

NameService, BootService, AuroraSSLCurrent, and AuroraSSLCertificateManager.

If you want Oracle8i to initiate any services for you during startup, supply a string with a comma-separated list of services to be installed when the ORB is initialized in the `UserORBServices` property. Each service must be a fully-qualified package name and name of the class that extends the `ORBServiceInit` class. You must extend this class in order for your service to be installed by Oracle8i.

Advanced CORBA Programming

This chapter discusses advanced CORBA programming techniques, such as calling back to the client from the server. Advanced programming for security and transactions are covered in their own chapters. This chapter covers the following topics:

- [Using SQLJ](#)
- [Implementing CORBA Callbacks](#)
- [Retrieving Interfaces With The IFR](#)
- [Using the CORBA Tie Mechanism](#)
- [Migrating from JDK 1.1 to Java 2](#)
- [Invoking CORBA Objects From Applets](#)
- [Interoperability with non-Oracle ORBs](#)

Using SQLJ

You can often simplify the implementation of a CORBA server object by using Oracle8i SQLJ to perform static SQL operations. Using SQLJ statements results in less code than the equivalent JDBC calls and makes the implementation easier to understand and debug. This section describes a version of the example first shown in ["A First CORBA Application"](#) on page 2-2, but uses SQLJ rather than JDBC for the database access. Refer to the *Oracle8i SQLJ Developer's Guide and Reference* for complete information about SQLJ.

The only code that changes for this SQLJ implementation is in the `EmployeeImpl.java` file, which implements the `Employee` object. The SQLJ implementation, which can be called `EmployeeImpl.sqlj`, is listed below. You can contrast that with the JDBC implementation of the same object in ["Write the Server Object Implementation"](#) on page 2-6.

```
package employeeServer;

import employee.*;
import java.sql.*;

public class EmployeeImpl extends _EmployeeImplBase {
    public EmployeeInfo getEmployee (int ID) throws SQLException {
        try {
            String name = null;
            double salary = 0.0;
            #sql { select ename, sal into :name, :salary from emp
                    where empno = :ID };
            return new EmployeeInfo (name, empno, (float)salary);
        } catch (SQLException e) {
            throw new SQLException (e.getMessage ());
        }
    }
}
```

The SQLJ version of this implementation is considerably shorter than the JDBC version. In general, Oracle recommends that you use SQLJ where you have static SQL commands to process, and use JDBC, or a combination of JDBC and SQLJ, in applications where dynamic SQL statements are required.

Running the SQLJ Translator

To compile the `EmployeeImpl.sqlj` file, you issue the following SQLJ command:

```
% sqlj -J-classpath
```



```

.:$(ORACLE_HOME)/lib/aurora_client.jar:$(ORACLE_HOME)/jdbc/lib/classes111.zip:
$(ORACLE_HOME)/sqlj/lib/translator.zip:$(ORACLE_HOME)/lib/vbjobr.jar:
$(ORACLE_HOME)/lib/vbjapp.jar:$(JDK_HOME)/lib/classes.zip -ser2class
employeeServer/EmployeeImpl.sqlj

```

This command does the following:

- translates the SQLJ code into a pure Java file
- compiles the resulting .java source to get a .class file
- the `-ser2class` option translates SER files to .class files

The SQLJ translation generates two additional class files:

```

employeeServer/EmployeeImpl_SJProfile0
employeeServer/EmployeeImpl_SJProfileKeys

```

which you must also load into the database when you execute the `loadjava` command.

A Complete SQLJ Example

This example is available in complete form in the `examples/corba/basic` example directory, complete with a Makefile or Windows NT batch file so you can see how the example is compiled and loaded.

Implementing CORBA Callbacks

This section describes how a CORBA server object can call back to a client. The basic technique that is shown in this example is the following:

- Write a client object that runs on the client side and contains the methods the called-back-to object performs.
- Implement a server object that has a method that takes a reference to the client callback object as a parameter.
- In the client code:
 - Instantiate the client callback object.
 - Register it with the BOA.
 - Pass its reference to the server object that calls it.
- In the server object implementation, perform the callback to the client.

Note: See "[Callbacks using Security](#)" on page 6-21 for examples of using callbacks within an SSL environment.

IDL

The IDL for this example is shown below. There are two separate IDL files: `client.idl` and `server.idl`:

```
/* client.idl */
module client {
    interface Client {
        wstring helloBack ();
    };
};

/* server.idl */
#include <client.idl>

module server {
    interface Server {
        wstring hello (in client::Client object);
    };
};
```

Note that the server interface includes the interface defined in `client.idl`.

Client Code

The client code for this example must instantiate the client-side callback object and register it with the BOA so that it can be accessed by the server. The code performs the following steps to do this:

- Invokes the `init()` method, with no parameters, on the ORB pseudo-object. This returns a reference to the existing client-side ORB.
- Uses the ORB reference to initialize the BOA.
- Instantiates a new client object.
- Registers the client object with the client-side BOA.

The code to perform these steps is as follows:

```
com.visigenic.vbroker.orb.ORB orb = oracle.aurora.jndi.orb_dep.Orb.init();
org.omg.CORBA.BOA boa = orb.BOA_init ();
```

```
ClientImpl client = new ClientImpl ();
boa.obj_is_ready (client);
```

Finally, the client code calls the server object, passes it a reference to the registered client-side callback object, and prints its return value, as follows:

```
System.out.println (server.hello (client));
```

Callback Server Implementation

The implementation of the server-side object is very simple. It receives the client-side callback object and invokes a method from this object. In this example, the server invokes the client-side `helloBack` method.

```
package serverServer;

import server.*;
import client.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class ServerImpl extends _ServerImplBase implements ActivatableObject
{
    public String hello (Client client) {
        return "I Called back and got: " + client.helloBack ();
    }

    public org.omg.CORBA.Object _initializeAuroraObject () {
        return this;
    }
}
```

The server simply returns a string that includes the string return value from the callback.

Callback Client-Server Implementation

The client-side callback server implements the desired callback method. The following example implements the `helloBack` method:

```
package clientServer;

import client.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class ClientImpl extends _ClientImplBase implements ActivatableObject
```

```
{
    public String helloBack () {
        return "Hello Client World!";
    }

    public org.omg.CORBA.Object _initializeAuroraObject () {
        return this;
    }
}
```

The client-side object is just like any other server object. But in this callback example it is running in the client ORB, which can be running on a client system, not necessarily running inside an Oracle8i database server.

Retrieving Interfaces With The IFR

The Interface Repository (IFR) specified by OMG defines how to store and retrieve interface definitions. The information contained within the interface can be used internally by the ORB to retrieve information about an object reference, for type-checking the request signatures, or used externally by DII/DSI applications for instantiating objects dynamically through DII/DSI.

You store the IDL interface definition within the IFR through the Oracle8i JServer `publish` command. The `publish` command stores the interface within a flat file, `AuroraIFR.idl`.

Note: Normally, this file is automatically written to `$ORACLE_HOME/javavm/admin`. However, if this directory is not write-enabled, you can specify another fully-qualified filename within the `"aurora.ifr.file"` system property through the `modifyprops` tool, as follows:

```
modifyprops -user scott/tiger@dbhost:5521:orcl
            "aurora.ifr.file" "/private/ifr/myIFRfile"
```

Once stored, you can retrieve the interface definition either implicitly through the `_get_interface_def` method or explicitly looking up the IFR Repository object and invoking the standard methods to traverse through the repository.

The following sections detail how to publish and retrieve IDL interface information:

- [Publishing the IDL Interface](#)

- [Retrieving Interfaces Implicitly](#)
- [Retrieving Interfaces Explicitly](#)

Publishing the IDL Interface

You store the IDL interface definition within the IFR through the Oracle8i JServer `publish` command. This command contains the following two options for storing the IDL interface definition within the IFR:

<code>-idl</code>	Load the IDL interface definition into the IFR. You can only load IDL interface definitions when using a "sess_iiop" URL.
<code>-replaceIDL</code>	If an IDL interface definition currently exists within the IFR, replace it with this version. If not specified, the <code>publish</code> command will not replace the existing interface within the IFR.

Note: The `-replaceIDL` flag will replace any interface with the same name in the IFR, even if it was originally stored by another user. Thus, different users can overwrite another user's interface unknowingly.

The following `publish` command loads the `Bank.idl` interfaces into the IFR. This is executed under the `SCOTT` schema security permissions. If it already exists, the `-replaceIDL` option specifies that the interfaces should be replaced with this version of `Bank.idl`.

```
publish -republish -user SCOTT -password TIGER -schema SCOTT \
  -service sess_iiop://dlsun164:2481:orcl \
  /test/myBank bankServer.AccountManagerImpl \
  Bank.AccountManagerHelper -idl Bank.idl -replaceIDL
```

To remove the IDL interface from the IFR, use the `-idl` option for the `sess_sh` `remove` command.

Retrieving Interfaces Implicitly

You can retrieve the interface definition implicitly through the `org.omg.CORBA.Object._get_interface_def` method. The object returned should be cast to `InterfaceDef`. The following code retrieves the `InterfaceDef` object for the `Bank.Account`:

```
AccountManager manager =
    (AccountManager)ic.lookup (serviceURL + objectName);

Bank.Account account = manager.open(name);

org.omg.CORBA.InterfaceDef intf = (org.omg.CORBA.InterfaceDef)
    account._get_interface_def();
```

Once retrieved, you can execute any of the `InterfaceDef` methods for retrieving information about the interface.

Retrieving Interfaces Explicitly

All defined interfaces stored in the IFR are stored in a hierarchy. The top level of the hierarchy is a `Repository` object, which is also a `Container` object. All objects under the `Repository` object are `Contained` objects. You can parse down through the `Container` objects, reviewing the `Contained` objects, until you find the particular interface definition you want.

Note: The user can only see the objects to which the user has read privileges.

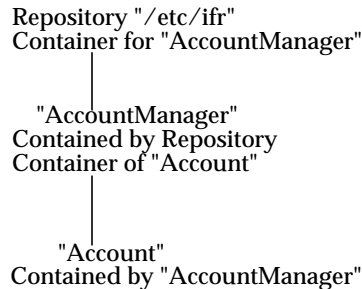
The `Repository` object is pre-published under the name `"/etc/ifr"`. You can retrieve the `Repository` object by executing the following:

You retrieve a prepublished IFR `Repository` object by looking up the `"/etc/ifr"` object as shown below:

```
Repository rep = (Repository)ic.lookup(serviceURL + "/etc/ifr");
```

Once the `Repository` object is retrieved, you can traverse through the hierarchy until you reach the object you are interested in. The methods for each object type, `InterfaceDef` and others, are documented fully in the OMG CORBA specification.

As shown in [Figure 5-1](#), the `Account` interface is contained within `AccountManager`, which is container within the `Repository` object.

Figure 5–1 IFR Hierarchy for Account Interface**Example 5–1 Traversing IFR Repository Within the print Method**

Once you retrieve the IFR object, you can traverse through all stored definitions within the IFR. The `print` method in [Example 5–1](#) prints out all stored definitions located within the IFR.

```

public void print( ) throws org.omg.CORBA.UserException {

    //retrieve the repository as a container... as the top level container
    org.omg.CORBA.Container container =
        (Container)ic.lookup(serviceURL + "/etc/ifr");

    //All objects in the IFR are Contained, except for the Repository.
    //Retrieve the contents of the Repository, which would be all objects that
    //it contains.
    org.omg.CORBA.Contained[] contained =
        container.contents(org.omg.CORBA.DefinitionKind.dk_all, true);

    //The length is equal to the number of objects contained within the IFR
    for(int i = 0; i < contained.length; i++) {
        {
            //Each Contained object has a description.
            org.omg.CORBA.ContainedPackage.Description description =
                contained[i].describe();

            //Each object is of a certain type, which is retrieved by the value method.
            switch(contained[i].def_kind().value()) {
                case org.omg.CORBA.DefinitionKind._dk_Attribute:
                    printAttribute(org.omg.CORBA.AttributeDefHelper.narrow(contained[i]));
                    break;
                case org.omg.CORBA.DefinitionKind._dk_Constant:
                    printConstant(org.omg.CORBA.ConstantDefHelper.narrow(contained[i]));

```

```
        break;
    case org.omg.CORBA.DefinitionKind._dk_Exception:
        printException(org.omg.CORBA.ExceptionDefHelper.narrow(contained[i]));
        break;
    case org.omg.CORBA.DefinitionKind._dk_Interface:
        printInterface(org.omg.CORBA.InterfaceDefHelper.narrow(contained[i]));
        break;
    case org.omg.CORBA.DefinitionKind._dk_Module:
        printModule(org.omg.CORBA.ModuleDefHelper.narrow(contained[i]));
        break;
    case org.omg.CORBA.DefinitionKind._dk_Operation:
        printOperation(org.omg.CORBA.OperationDefHelper.narrow(contained[i]));
        break;
    case org.omg.CORBA.DefinitionKind._dk_Alias:
        printAlias(org.omg.CORBA.AliasDefHelper.narrow(contained[i]));
        break;
    case org.omg.CORBA.DefinitionKind._dk_Struct:
        printStruct(org.omg.CORBA.StructDefHelper.narrow(contained[i]));
        break;
    case org.omg.CORBA.DefinitionKind._dk_Union:
        printUnion(org.omg.CORBA.UnionDefHelper.narrow(contained[i]));
        break;
    case org.omg.CORBA.DefinitionKind._dk_Enum:
        printEnum(org.omg.CORBA.EnumDefHelper.narrow(contained[i]));
        break;
    case org.omg.CORBA.DefinitionKind._dk_none:
    case org.omg.CORBA.DefinitionKind._dk_all:
    case org.omg.CORBA.DefinitionKind._dk_Typedef:
    case org.omg.CORBA.DefinitionKind._dk_Primitive:
    case org.omg.CORBA.DefinitionKind._dk_String:
    case org.omg.CORBA.DefinitionKind._dk_Sequence:
    case org.omg.CORBA.DefinitionKind._dk_Array:
    default:
        break;
    }
}
}
```

Using the CORBA Tie Mechanism

There is only one special consideration when you use the CORBA Tie, or delegation, mechanism rather than the inheritance mechanism for server object implementations. In the Tie case, you must implement the

`oracle.aurora.AuroraServices.ActivatableObject` interface. This interface has a single method: `_initializeAuroraObject()`.

(Note that earlier releases of the Oracle8i ORB required you to implement this method for all server objects. For this release, its implementation is required only for Tie objects.)

The implementation of `_initializeAuroraObject()` for a tie class is typically:

```
import oracle.aurora.AuroraServices.ActivatableObject;
...
public org.omg.CORBA.Object _initializeAuroraObject () {
    return new _tie_Hello (this);
}
...
```

where `_tie_<interface_name>` is the tie class generated by the IDL compiler.

You must also always include a public, parameterless constructor for the implementation object.

See the `tieimpl` example in the CORBA examples set for a complete example that shows how to use the Tie mechanism. See also "[TIE Example](#)" on page A-22 for the code.

Migrating from JDK 1.1 to Java 2

Oracle8i JServer updated its ORB implementation to Visibroker 3.4, which is compatible with both JDK 1.1 and Java 2.

Note: All existing CORBA applications must regenerate their stubs and skeletons to work with 8.1.6. You must use the 8.1.6 tools when regenerating code from an IDL file.

Sun Microsystems's Java 2 contains an OMG CORBA implementation; JDK 1.1 did not contain an OMG CORBA implementation. Thus, when you imported the Inprise libraries and invoked the CORBA methods, it always invoked the Visibroker implementation. With the implementation being contained in Java 2, if you invoke the CORBA methods without any modifications—as discussed below—you will invoke the Sun Microsystems CORBA implementation, which can cause unexpected results.

The following lists the three methods for initializing the ORB on the client-side and recommendations for bypassing the Sun Microsystems CORBA implementation:

- **JNDI Lookup**—The setup for the lookup method is the same for both JDK 1.1 and Java 2. However, you must regenerate the stubs and skeletons.
- **Aurora ORB Interface**—The Aurora ORB provides an interface for initializing the ORB. If you do not use JNDI, your client initializes an ORB on its node to communicate with the ORB in the database. You can use an Aurora ORB on your client through this class.
- **CORBA ORB Interface**—If you want to use OMG's CORBA ORB interface, you must set a few properties to ensure you are accessing the correct implementation. If you do not wish to use the Aurora ORB on your client, you can use the pure CORBA interfaces. However, you must set up your environment to direct your calls to the correct implementation.

JNDI Lookup

If you are using JNDI on the client to access CORBA objects that reside in the server, no code changes are necessary. However, you must regenerate your CORBA stubs and skeletons.

Aurora ORB Interface

If your client environment uses JDK 1.1, you do not need to change your existing code. However, you must regenerate your stubs and skeletons.

If your client environment has been upgraded to Java 2, you can initialize the ORB through the `oracle.aurora.jndi.orb_dep.Orb.init` method. This method guarantees that when you initialize the ORB, it will initialize only a single ORB instance. That is, if you use the Java 2 ORB interface, it returns a new ORB instance each time you invoke the `init` method. Aurora's `init` method initializes a singleton ORB instance. Each successive call to `init` returns an object reference to the existing ORB instance.

In addition, the Aurora ORB interface manages the session-based IIOP connection.

oracle.aurora.jndi.orb_dep.Orb Class There are several `init` methods, each with a different parameter list. The following describes the syntax and parameters for each `init` method.

Note: The returned class for each `init` method is different. You can safely cast the `org.omg.CORBA.ORB` class to `com.visigenic.vbroker.orb.ORB`.

No Parameters

If you execute the `ORB.init` method that takes no parameters, it does the following:

- If no ORB instance exists, it creates an ORB instance and returns its reference to you.
- If an ORB instance exists, it returns the ORB reference to you.

Syntax

```
public com.visigenic.vbroker.orb.ORB init();
```

Providing ORB Properties

If you execute the `ORB.init` method that takes the ORB properties as the only parameter, it does the following:

- If no ORB instance exists, it creates an ORB instance, taking into account the properties argument, and returns its reference to you.
- If an ORB instance exists, it returns the ORB reference to you.

Syntax

```
public org.omg.CORBA.ORB init(Properties props);
```

Providing Input Arguments and ORB Properties

If you execute the `ORB.init` method that takes the ORB properties and ORB command-line arguments, it always creates an ORB instance and returns the reference to you.

Syntax

```
public org.omg.CORBA.ORB init(String[] args, Properties props);
```

Parameter	Description
Properties props	ORB system properties.
String[] args	Arguments that are passed to the ORB instance.

Example 5–2 Using Aurora ORB init method

The following example shows a client instantiating an ORB using the Aurora Orb class.

```
// Create the client object and publish it to the orb in the client
// Substitute Aurora's Orb.init for OMG ORB.init call
// old way: org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init ();
com.visigenic.vbroker.orb.ORB orb = oracle.aurora.jndi.orb_dep.Orb.init();
```

Providing ORB Properties with Username, Password, and Role

If you execute the `ORB.init` method that provides the ORB properties, username, password, and role as parameters, it does the following:

- If no ORB instance exists, it creates an ORB instance and returns its reference to you.
- If an ORB instance exists, it returns the ORB reference to you.

You would use this method when your client chooses to not use JNDI for ORB initialization and it receives a reference to an existing object from another client. To access an active object within a session, the new client must authenticate itself to the database in one of two ways:

- If `SSL_CREDENTIALS` is requested, provide the username, password, and role in the `init` method parameters. Then, when you invoke a method on the supplied object reference, the username, password, and role are passed implicitly on the first message to authenticate the client to the database.
- If the login protocol is requested, through either `SSL_LOGIN` or `NON_SSL_LOGIN`, the first client must pass object references to both the login object and the destination object. The second client authenticates itself by providing the username, password, and role on the `authenticate` method of the login object. Then, it executes any method on the object.

This method is how a second client invokes an active object in an established session.

Syntax

```
public org.omg.CORBA.ORB init(String un, String pw, String role,
                             boolean ssl, java.util.Properties props);
```

Parameter	Description
String un	The username for client-side authentication.
String pw	The password for client-side authentication.
String role	The role to use after logging on.
Boolean ssl	If true, SSL is enabled for the connection. If false, a NON-SSL connection is used.
Properties props	Properties that are used by the ORB.

CORBA ORB Interface

If you have implemented a pure CORBA client—that is, you do not use JNDI—you must set the following properties before the ORB initialization call. These properties direct the call to the Aurora implementation, rather than the Java 2 implementation. This ensures the behavior that you expect. The behavior expected from Visibroker is as follows:

- Even if you invoke `ORB.init` more than once, JServer creates only a single ORB instance. If you do not set these properties, be aware that each invocation of `ORB.init` will create a new ORB instance.
- The session IIOP connection is managed correctly.
- Callbacks from the server are managed correctly.

Property	Assign Value
<code>org.omg.corba.ORBClass</code>	<code>com.visigenic.vbroker.orb</code>
<code>org.omg.corba.ORBSingletonClass</code>	<code>com.visigenic.vbroker.orb</code>

Example 5-3 Assigning Visibroker values to OMG properties

The following example shows how to set up the OMG properties for directing the OMG CORBA `init` method to the Visibroker implementation.

```
System.getProperties().put("org.omg.CORBA.ORBClass",
                          "com.visigenic.vbroker.orb.ORB");
```

```
System.getProperties().put("org.omg.CORBA.ORBSingletonClass",  
                           "com.visigenic.vbroker.orb.ORB");
```

Or you can set the properties on the command line, as follows:

```
java -Dorg.omg.CORBA.ORBClass=com.visigenic.vbroker.orb.ORB  
     -Dorg.omg.CORBA.ORBSingletonClass=com.visigenic.vbroker.orb.ORB
```

Backwards Compatibility with 8.1.5

The tools provided with Oracle8i, such as `publish`, have been modified to work with either a JDK 1.1 or Java 2 environment. However, any code that has been generated or loaded with the 8.1.5 version of any tool, will not succeed. Make sure that you always use the 8.1.6 version of all tools. This rule applies to your CORBA stubs and skeletons. You must regenerate all stubs and skeletons with the 8.1.6 IDL compiler.

Invoking CORBA Objects From Applets

You invoke a server object from an applet in the same manner as from a client. The only differences are the following:

- You must conform to the applet standards.
- You must conform to the Java plug-in standards. The Java plug-ins that are supported are JDK 1.1, Java 2, and Oracle's JInitiator.
- You set the following properties within the initial context environment before the object lookup: `ORBdisableLocator`, `ORBClass`, and `ORBSingletonClass`.

Using Signed JAR Files to Conform to Sandbox Security

The security sandbox constricts your applet from accessing anything on the local disk or from connecting to a remote host other than the host that the applet was downloaded from. If you create a signed JAR file as a trusted party, you can bypass the sandbox security. See <http://java.sun.com> for more information on applet sandbox security and signed JAR files.

Performing Object Lookup in Applets

You perform the JNDI lookup within the applet the same as within any Oracle Java client, except that you set the following property within the initial context:

```
env.put(ServiceCtx.APPLET_CLASS, this);
```

By default, you do not need to install any JAR files on the client to run the applet. However, if you want to place the Oracle JAR files on the client machine, set the `ClassLoader` property in the `InitialContext` environment, as follows:

```
env.put('ClassLoader', this.getClass().getClassLoader());
```

The following shows the `init` method within an applet that invokes the `Bank` example. The applet sets up the initial context—including setting the `APPLET_CLASS` property—and performs the JNDI lookup giving the URL.

```
public void init() {
    // This GUI uses a 2 by 2 grid of widgets.
    setLayout(new GridLayout(2, 2, 5, 5));
    // Add the four widgets.
    add(new Label("Account Name"));
    add(_nameField = new TextField());
    add(_checkBalance = new Button("Check Balance"));
    add(_balanceField = new TextField());
    // make the balance text field non-editable.
    _balanceField.setEditable(false);
    try {
        // Initialize the ORB (using the Applet).
        Hashtable env = new Hashtable();
        env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        env.put(Context.SECURITY_PRINCIPAL, "scott");
        env.put(Context.SECURITY_CREDENTIALS, "tiger");
        env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
        env.put(ServiceCtx.APPLET_CLASS, this);

        Context ic = new InitialContext(env);
        _manager = (AccountManager)ic.lookup
            ("sess_iiop://hostfunkt:2222/test/myBank");
    } catch (Exception e) {
        System.out.println(e.getMessage());
        e.printStackTrace();
        throw new RuntimeException();
    }
}
```

Within the `action` method, the applet invokes methods off of the retrieved object. In this example, the `open` method of the retrieved `AccountManager` object is invoked.

```
public boolean action(Event ev, Object arg) {
    if(ev.target == _checkBalance) {
        // Request the account manager to open a named account.
        // Get the account name from the name text widget.
        Bank.Account account = _manager.open(_nameField.getText());
        // Set the balance text widget to the account's balance.
        _balanceField.setText(Float.toString(account.balance()));
        return true;
    }
    return false;
}
```

Modifying HTML for Applets that Access CORBA Objects

Oracle8i supports only the following Java plug-ins for the HTML page that loads in the applet: JDK 1.1, Java 2, and Oracle JInitiator. Each plug-in contains different syntax for the applet information. However, each HTML page may contain definitions for the following two properties:

- `ORBdisableLocator` set to `TRUE`—Required for all applets.
- `ORBClass` and `ORBSingletonClass` definitions—Required for the applets that use the Java 2 or JInitiator plug-in.

Note: Because of the sandbox security rules, you cannot set or read any system properties. Therefore, any values that you want to pass on to the ORB runtime, you may set within the applet parameters. This is the method used to set the `ORBdisableLocator`, `ORBClass` and `ORBSingletonClass` properties.

The examples in the following sections show how to create the correct HTML definition for each plug-in type. Each HTML definition defines the applet bank example.

- [Example 5-4, "HTML Definition for JDK 1.1 Plug-in"](#)
- [Example 5-5, "HTML Definition for Java 2 Plug-in"](#)
- [Example 5-6, "HTML Definition for JInitiator Plug-in"](#)

Example 5-4 HTML Definition for JDK 1.1 Plug-in

```
<pre>
```



```

<html>
<title>Applet talking to 8i</title>
<h1>applet talking to 8i using java plug in 1.1 </h1>
<hr>
The bank example
Specify the plugin in codebase, the class within the CODE parameter, the JAR
files in the ARCHIVE parameter, the plugin version in the type parameter, and
set ORBdisableLocator to true.
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
        WIDTH = 500 HEIGHT = 50
        codebase="http://java.sun.com/products/plugin/1.1/
        jinstall-11-win32.cab#Version=1,1,0,0">
  <PARAM NAME = CODE VALUE = OracleClientApplet.class >
  <PARAM NAME = ARCHIVE VALUE = "oracleClient.jar,
        aurora_client.jar,vbjorb.jar,vbjapp.jar" >
  <PARAM NAME="type" VALUE="application/x-java-applet;version=1.1">
  <PARAM NAME="ORBdisableLocator" VALUE="true">
<COMMENT>
Set the plugin version in the type, set ORBdisableLocator to true, the applet
class within the java_CODE tag, the JAR files in the java_ARCHIVE tag, and the
plug-in source site within the pluginspage tag.
<EMBED type="application/x-java-applet;version=1.1"
        ORBdisableLocator="true"
        java_CODE = OracleClientApplet.class
        java_ARCHIVE = "oracleClient.jar,
        aurora_client.jar,vbjorb.jar,vbjapp.jar"
        WIDTH = 500 HEIGHT = 50
        pluginspage="http://java.sun.com/products/plugin/1.1/plugin-install.html">
<NOEMBED></COMMENT>
</NOEMBED></EMBED>
</OBJECT>

</center>
<hr>
</pre>

```

Example 5-5 HTML Definition for Java 2 Plug-in

```

<pre>
<html>
<title>applet talking to 8i</title>
<h1>applet talking to 8i using Java plug in 1.2 </h1>
<hr>
The bank example
Specify the plugin in codebase, the class within the CODE parameter, the JAR

```

files in the ARCHIVE parameter, the plugin version in the type parameter, and set ORBdisableLocator to true.

```

<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
  WIDTH = 500 HEIGHT = 50
  codebase="http://java.sun.com/products/plugin/1.2/jinstall-11-win32.cab#
    Version=1,1,0,0">
  <PARAM NAME = CODE VALUE = OracleClientApplet.class >
  <PARAM NAME = ARCHIVE VALUE = "oracleClient.jar,
    aurora_client.jar,vbjorb.jar,vbjapp.jar" >
  <PARAM NAME="type" VALUE="application/x-java-applet;version=1.1.2">
  <PARAM NAME="ORBdisableLocator" VALUE="true">
  <PARAM NAME="org.omg.CORBA.ORBClass" VALUE="com.visigenic.vbroker.orb.ORB">
  <PARAM NAME="org.omg.CORBA.ORBSingletonClass"
    VALUE="com.visigenic.vbroker.orb.ORB">
  <COMMENT>
  Set the plugin version in the type, set ORBdisableLocator to true, the ORBClass
  and ORBSingletonClass to the correct ORB class, the applet
  class within the java_CODE tag, the JAR files in the java_ARCHIVE tag, and the
  plug-in source site within the pluginspage tag.
  <EMBED type="application/x-java-applet;version=1.1.2"
    ORBdisableLocator="true"
    org.omg.CORBA.ORBClass="com.visigenic.vbroker.orb.ORB"
    org.omg.CORBA.ORBSingletonClass="com.visigenic.vbroker.orb.ORB"
    java_CODE = OracleClientApplet.class
    java_ARCHIVE = "oracleClient.jar,
      aurora_client.jar,vbjorb.jar,vbjapp.jar"
    WIDTH = 500 HEIGHT = 50
  pluginspage="http://java.sun.com/products/plugin/1.2/plugin-install.html">
  <NOEMBED></COMMENT>
</NOEMBED></EMBED>
</OBJECT>

</center>
<hr>
</pre>

```

Example 5-6 HTML Definition for JInitiator Plug-in

```

<h1> applet talking to 8i using JInitiator 1.1.7.18</h1>
  <COMMENT>
  Set the plugin version in the type, set ORBdisableLocator to true, the
  ORBClass and ORBSingletonClass to the correct ORB class, the applet
  class within the java_CODE tag, the source of the applet in the java_CODEBASE
  and the JAR files in the java_ARCHIVE tag.
  <EMBED type="application/x-jinit-applet;version=1.1.7.18"

```

```

java_CODE="OracleClientApplet"
java_CODEBASE="http://hostfunk:8080/applets/bank"
java_ARCHIVE="oracleClient.jar,aurora_client.jar,vbJORB.jar,vbjapp.jar"
WIDTH=400
HEIGHT=100
ORBdisableLocator="true"
org.omg.CORBA.ORBClass="com.visigenic.vbroker.orb.ORB"
org.omg.CORBA.ORBSingletonClass="com.visigenic.vbroker.orb.ORB"
serverHost="orasundb"
serverPort=8080
<NOEMBED>
</COMMENT>
</NOEMBED>
</EMBED>

```

Interoperability with non-Oracle ORBs

You can interoperate with Oracle8i from a client that uses another vendor's ORB. To do so, the vendor must provide the functionality that Oracle8i uses by being part of the database: functions such as session-based connections, extended CosNaming functions, and the login protocol. To provide this functionality, your ORB vendor must work with Oracle's Product Management to provide libraries for you.

All client-side functionality has been packaged into `aurora_client.jar`. This JAR file has been broken into two JAR files for interoperating with your ORB vendor:

- `aurora_orbindep.jar`—includes ORB independent features, such as JNDI
- `aurora_orbdep.jar`—includes Oracle ORB dependent functionality, such as session-based communication, the login protocol, and security context

Your ORB vendor needs to provide you the `aurora_orbdep.jar` file. Thus, you include their `aurora_orbdep.jar` file and the Oracle-provided `aurora_orbindep.jar` file to replace `aurora_client.jar`.

Note: If you do not remove `aurora_client.jar` file from your CLASSPATH, you will be using Oracle's classes, instead of your ORB vendor's classes.

The `aurora_orbdep.jar` includes the following functionality:

Function	Description
login	The login protocol performs the challenge/response protocol for authenticating the client to the database. See "IOP Security" on page 6-1 for more information.
bootstrap	The boot service obtains key services, such as CosNaming.
extended CosNaming	The Aurora ORB extended CosNaming to automatically instantiate an object upon first lookup.
Session IOP	Session IOP is implemented to allow one client connect to more than a single IOP session at the same time. See Chapter 3, "Configuring IOP Applications" for more information.
Credentials	The security context interceptor for the credential type of authentication.

Java Client using Oracle ORB

You perform the following if you choose to use the Oracle-provided ORB on your client:

1. Put `aurora_client.jar` in a directory that exists in the CLASSPATH.
2. Compile and run your CORBA application.

Java Client using non-Oracle ORB

You perform the following if you choose to use another vendor's ORB on your client:

1. Put `aurora_orbindep.jar` in a directory that exists in the CLASSPATH.
2. Contact your ORB vendor to receive their `aurora_orbdep.jar`.
3. Put their `aurora_orbdep.jar` in a directory that exists in the CLASSPATH.
4. Compile and run your CORBA application.

Note: If you do not remove `aurora_client.jar` file from your CLASSPATH, you will be using Oracle's classes, instead of your ORB vendor's classes.

C++ Client Interoperability

With C++ clients, the ORB vendor must provide the `aurora_client.jar` file functionality in shared libraries. The vendor will make use of Oracle-provided C++ login protocol for authentication. All clients are required to authenticate themselves to the database. One of the methods for authenticating is through the login protocol.

The login protocol is an Oracle-specific design, used for logging in to a database by providing a username and password to authenticate the client. The following example shows how to write a sample C++ CORBA client to Oracle8i. This example uses the Visigenics C++ ORB for its client-side ORB.

Example 5-7 C++ client using login protocol to authenticate

The following C++ client uses the Visigenics C++ ORB for the client-side ORB. Your implementation can be different, depending on the type of ORB you use.

```
#include <Login.h>
#include <oracle_orbdep.h>

// set up host, port, and SID
char *sid = NULL;
char *host = argv[1];
int port = atoi(argv[2]);
if(argc == 4) sid = argv[3];

// set up username, password, and role
wchar_t *username = new wchar_t[6];
username[0] = 's';
username[1] = 'c';
username[2] = 'o';
username[3] = 't';
username[4] = 't';
username[5] = '\\0';

wchar_t *password = new wchar_t[6];
password[0] = 't';
password[1] = 'i';
password[2] = 'g';
```

```
password[3] = 'e';
password[4] = 'r';
password[5] = '\\0';

wchar_t *role = new wchar_t[1];
role[0] = '\\0';

// Get the Name service Object reference
AuroraServices::PublishingContext_ptr rootCtx = NULL;

// Contact Visibroker's boot service for initializing
rootCtx = VisiCppBootstrap::getNameService (host, port, sid);

// Get the pre-published login object reference
AuroraServices::PublishedObject_ptr loginPubObj = NULL;
AuroraServices::LoginServer_ptr serv = NULL;
CosNaming::NameComponent *nameComponent = new CosNaming::NameComponent[2];

nameComponent[0].id = (const char *)"etc";
nameComponent[0].kind = (const char *)"";
nameComponent[1].id = (const char *)"login";
nameComponent[1].kind = (const char *)"";

CosNaming::Name *name1 = new CosNaming::Name(2, 2, nameComponent, 0);

// Lookup this object in the Name service
CORBA::Object_ptr loginCorbaObj = rootCtx->resolve (*name1);

// Make sure it is a published object
loginPubObj = AuroraServices::PublishedObject::_narrow (loginCorbaObj);

// create and activate this object (non-standard call)
loginCorbaObj = loginPubObj->activate_no_helper ();
serv = AuroraServices::LoginServer::_narrow (loginCorbaObj);

// Create a client login proxy object and authenticate to the DB
oracle_orbdep *_visi = new oracle_orbdep(serv);
Login login(_visi);
boolean res = login.authenticate(username, password, role);
```

IIOP Transport Protocol

If, when using another vendor's ORB, the ORB vendor does not support session-based IIOP, you can use a regular IIOP port. Any client that uses a regular IIOP transport can not access multiple sessions.

To configure a non-session-based IIOP listener, you must do the following:

1. Configure the `MTS_DISPATCHERS` parameter to `oracle.aurora.server.GiopServer` instead of `oracle.aurora.server.SGiopServer`.

```
mts_dispatchers="(protocol=tcp | tcps)
                 (presentation=oracle.aurora.server.GiopServer)"
```

2. Set the `TRANSPORT_TYPE` property to `ServiceCtx.IIOP`, as shown below:

```
Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(Context.SECURITY_PRINCIPAL, user);
env.put(Context.SECURITY_CREDENTIALS, password);
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
env.put("TRANSPORT_TYPE", ServiceCtx.IIOP);
Context ic = new InitialContext(env);
```

Note: Instead of setting the `TRANSPORT_TYPE` property, you can use the `-iiop` option on any of the command-line tools. If your client is directing the request to a dispatcher, you must also provide the regular IIOP port within the service name on the command-line.

IIOp Security

Security involves data integrity, authentication, and authorization.

- For data integrity, Oracle8i enables your application to use IIOp over a secure socket layer (SSL).
- For authentication, your application can choose between providing a username/password combination or a certificate.
- For authorization, you can choose the level of trust points that any incoming clients will be required to give.

The following sections explain these subjects in detail:

- [Overview](#)
- [Data Integrity](#)
- [Authentication](#)
- [Client-side Authentication](#)
- [Server-Side Authentication](#)
- [Authorization](#)

Overview

As discussed in the *Oracle8i Java Developer's Guide*, there are several security issues you must think about for your application. The *Oracle8i Java Developer's Guide* divides security into network connection, database contents, and JVM security issues. All these issues are pertinent to IIOP. However, IIOP has specific implementation issues for both the networking and the JVM security, as listed below:

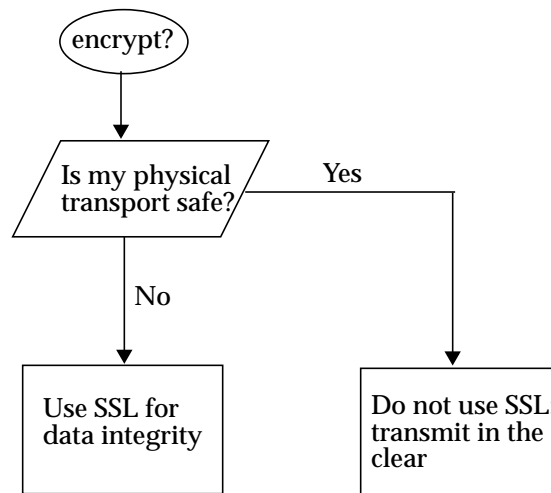
- JVM security includes both utilizing Java2 permissions and granting execution rights. For IIOP, you can grant execution privileges in one of two ways:
 - * CORBA—The owner grants execution rights to CORBA objects with an option on the `loadjava` tool. See the `loadjava` discussion in the *Oracle8i Java Developer's Guide* for information on granting execution rights when loading the CORBA classes.
 - * EJB—The owner grants execution rights to EJB objects and, potentially, methods within the deployment descriptor. See the section on "Access Control" in the *Oracle8i Enterprise JavaBeans Developer's Guide* for more information on defining execution rights within your deployment descriptor.
- Network connection security includes the following issues:
 - * **Data Integrity**—To prevent a sniffer from reading the transmission directly off the wire, all transmissions are encoded. Oracle supports Secure Socket Layer (SSL) for encryption.
 - * **Authentication**—To prevent an invalid user from impersonating a valid user, the client or server provides authentication information. This information can take the form of a username/password combination or certificates.
 - * **Authorization**—To prove that the user is allowed access to the object, two types of authorization are performed:
 - Session authorization—The session is authorized to the user. In this case, the client is authorized to access the server through validating either the username or certificate provided.
 - User authorization—The client or server can perform authorization on a provided certificate. This type of authorization can be performed only when the client or server authenticates itself by providing a certificate.

This section describes fully the network connection security issues that IIO applications must consider.

Data Integrity

Do you want your transport line to be encrypted? Do you want data integrity and confidentiality? If you believe that the physical connection can be tampered with, you can consider encrypting all transmissions by using the secure socket layer (SSL) encryption technology. However, because adding encryption to your transmission affects your connection performance, if you do not have any transport security issues, you should transmit unencrypted.

Figure 6–1 Data Integrity Decision Tree



Using the Secure Socket Layer

JServer's CORBA and EJB implementations rely on the Secure Socket Layer (SSL) for data integrity and authentication. SSL is a secure networking protocol, originally defined by Netscape Communications, Inc. Oracle8i JServer supports SSL over the IIO protocol used for the ORB.

When a connection is requested between a client and the server, the SSL layer within both parties negotiate during the connection handshake to verify if the connection is allowed. The connection is verified at several levels:

1. The SSL version on both the client and the server must agree for the transport to be guaranteed for data integrity.
2. If server-side authentication with certificates is requested, the certificates provided by the server are verified by the client at the SSL layer. This means that the server is guaranteed to be itself. That is, it is not a third party pretending to be the server.
3. If client-side authentication with certificates is requested, the certificates provided by the client are verified at the SSL layer. The server receives the client's certificates for authentication or authorization of the client.

Note: Normally, client-side authentication means only that the server verifies that the client is not an impersonator and is trusted. However, when you specify `SSL_CLIENT_AUTH` in JServer, you are requesting both server-side and client-side authentication.

The SSL layer performs authentication between the peers. After the handshake, you can be assured that the peers are authenticated to be who they say they are. You can perform additional tests on their certificate chain to authorize that this user can access your application. See "[Authorization](#)" on page 6-26 on how to go beyond authentication.

Note: If you decide to use SSL, your client must import the following JAR files:

- If your client uses JDK 1.1, import `jssl-1_1.jar` and `javax-ssl-1_1.jar`.
 - If your client uses Java 2, import `jssl-1_2.jar` and `javax-ssl-1_2.jar`.
-
-

SSL Version Negotiation

SSL makes sure that both the client and server side agree on an SSL protocol version number. The values that you can specify are as follows:

- **Undetermined:** `SSL_UNDETERMINED`. This is the default setting.
- **3.0 with 2.0 Hello:** This setting is not supported.
- **3.0:** `SSL_30`.

- 2.0: This setting is not supported.

On the database, the default is "Undetermined". The database does not support 2.0 or 3.0 with 2.0 Hello. Thus, you can use only the Undetermined or 3.0 setting for the client.

- The server's version is set within the database SQLNET.ORA file, using the SSL_VERSION parameter. For example, SSL_VERSION = 3.0.
- For the client, you set the SSL client version number in the client's JNDI environment, as follows:

```
environment.put("CLIENT_SSL_VERSION", ServiceCtx.SSL_30);
```

Table 6-1 shows which handshakes resolve to depending on SSL version settings on both the client and the server. The star sign "*" indicates cases where the handshake fails.

Table 6-1 SSL Version Numbers

Client Setting	Server Setting			
	Undetermined	3.0 W/2.0 Hello (not supported)	3.0	2.0 (not supported)
Undetermined	3.0	*	*	*
3.0 W/2.0 Hello (not supported)	*	*	*	*
3.0	3.0	*	3.0	*
2.0 (not supported)	*	*	*	*

Authentication

Authentication is the process where one party supplies to a requesting party information that identifies itself. This information guarantees that the originator is not an imposter. In the client/server distributed environment, authentication can be required from the client or the server:

- Server-side authentication—The server sends identifying information to authenticate itself. The client uses this information to verify that the server is itself and not an imposter. If you request SSL, the server will always send certificate-based authentication information.

- Client-side authentication—For the same reasons, the client sends identifying information to the server, which includes either a username/password combination or certificates. Since the client is logging on to a database, the client must always authenticate itself to the database.
- Callout authentication—The server initiates a call to another object. This causes the server to act as a client; as such, the server cannot use the database authentication information, but must provide information and authenticate itself as an independent party.
- Callback authentication—The server is given either a CORBA IOR or an EJB handle for calling back to an object that exists on the client. In this scenario, the server is acting as a client; as such, the server cannot use the database authentication information, but must provide information and authenticate itself as an independent party.

Client-side Authentication

The Oracle data server is a secure server; a client application cannot access data stored in the database without first being authenticated by the database server. Oracle8i CORBA server objects and Enterprise JavaBeans execute in the database server. For a client to activate such an object and invoke methods on it, the client must authenticate itself to the server. The client authenticates itself when a CORBA or EJB object starts a new session. The following are examples of how each IIOP client must authenticate itself to the database:

- When a client initially starts a new session, it must authenticate itself to the database.
- When a client passes an object reference (a CORBA IOR or an EJB bean handle) to a second client, the second client connects to the session specified in the object reference. The second client authenticates itself to the server.

The client authenticates itself by providing one of the following types:

Authentication type	Definition
Certificates	You can provide the user certificate, the Certificate Authority certificate (or a chain that contains both, including other identifying certificates), and a private key.
Username and password combination	You can provide the username and password through either credentials or the login protocol. In addition, you can pass a database role to the server, along with the username and password.

The type of client-side authentication can be determined by the server's configuration. If, within the SQLNET.ORA file, the `SSL_CLIENT_AUTHENTICATION` parameter is `TRUE`, the client must provide certificate-based authentication. If `SSL_CLIENT_AUTHENTICATION` is `FALSE`, the client authenticates itself with a username/password combination. If `SSL_CLIENT_AUTHENTICATION` is `TRUE` and the client provides a username/password, the connection handshake will fail.

The following table gives a brief overview of the options that the client has for authentication.

- The columns represent the options available if you have chosen to use SSL for data integrity.
- The rows demonstrate the three authentication vehicles: login protocol, credentials, and certificates.
- The table entries detail the different methods you must employ when implementing the client-side authentication type.

Authentication vehicle	NON-SSL transport	SSL transport
Providing username and password using the login protocol	<ul style="list-style-type: none"> ■ Implicit method: Set JNDI property to <code>NON_SSL_LOGIN</code>; provide username and password in JNDI properties. ■ Explicit method: Create a Login object with username and password. 	<ul style="list-style-type: none"> ■ Implicit method: Set JNDI property to <code>SSL_LOGIN</code>; provide username and password in JNDI properties. ■ Explicit method: Create a Login object with username and password.
Providing username and password using credentials	Not supported because the password would transmit in the clear.	Set JNDI property to <code>SSL_CREDENTIAL</code> ; username and password are implicitly sent to the server in the handshake.
Providing certificates	Not supported because certificates require an SSL transport.	Set JNDI property to <code>SSL_CLIENT_AUTH</code> ; provide client certificate, CA certificate, and private key in JNDI properties. Pure CORBA objects use <code>AuroraCertificateManager</code> class to specify certificates, CA certificate, and private key.

As the table demonstrates, most of the authentication options include setting an appropriate value in JNDI properties.

Using JNDI for Authentication

To set up client-side authentication using JNDI, you set the `javax.naming.Context.SECURITY_AUTHENTICATION` attribute to one of the following values:

- `ServiceCtx.NON_SSL_LOGIN`—A plain IIOP connection is used. Because SSL is not used, all data flowing over the line is not encrypted. Thus, to protect the password, the client uses the login protocol to authenticate itself. In addition, the server does not provide SSL certificates to the client to identify itself.
- `ServiceCtx.SSL_LOGIN`—An SSL-enabled IIOP connection is used. All data flowing over the transport is encrypted. If you do not want to provide a certificate for the client authentication, use the login protocol to provide the username and password.

Because this is an SSL connection, the server sends its certificate identity to the client. The client is responsible for verifying the server's certificate, if interested, for server authentication. Optionally, the client can set up trust points for the server's certificate to be verified against.

- `ServiceCtx.SSL_CREDENTIAL`—An SSL-enabled IIOP connection is used. All data flowing over the transport is encrypted. The client provides the username and password without using the login protocol for client authentication to the server. The username and password are automatically passed to the server in a security context, on the first message.

Note: The client's password is not encrypted, as it is with SSL. It might be slightly more efficient than `SSL_LOGIN`, where encrypting a password over an SSL connection is redundant.

The server provides its certificate identity to the client. The client is responsible for verifying the server's certificate, if interested, for server authentication.

- `ServiceCtx.SSL_CLIENT_AUTH`—An SSL-enabled IIOP connection is used. All data flowing over the transport is encrypted. The client provides appropriate certificates for client-side authentication to the server. In addition, the server provides its certificate identity to the client. If interested, the client is responsible for authorizing the server's certificate.

- Nothing is specified. The client must activate the login protocol explicitly before activating and invoking methods on a server-side object. Use this method when a client must connect to an existing session and invoke methods on an existing object. See the `demo/examples/corba/session/sharedsession` example for more information. The username and password in the initial context environment are automatically passed as parameters to the login object's `authenticate()` method.

Within each of these options, you choose to do one or more of the following:

Client authentication	<ul style="list-style-type: none"> ■ authenticate itself to the server using login protocol ■ authenticate itself to the server using straight username and password ■ authenticate itself to the server using SSL certificates
Server authentication	<ul style="list-style-type: none"> ■ authenticate itself to the client using SSL certificates

For information on how to implement each of these methods for client or server authentication, see the following sections:

- [Providing Username and Password for Client-Side Authentication](#)
- [Using Certificates for Client Authentication](#)
- [Server-Side Authentication](#)

Providing Username and Password for Client-Side Authentication

The client authenticates itself to the database server either through a username/password or by supplying appropriate certificates. The username/password can be supplied either through Oracle's login protocol or credentials over the SSL transport connection.

- Provide a username and password by setting JNDI properties, which implicitly sets these values in a login protocol. Set `SECURITY_AUTHENTICATION` to `ServiceCtx.SSL_LOGIN` or `ServiceCtx.NON_SSL_LOGIN`.
- Provide a username and password through credentials. The username and password are provided implicitly and are shipped to the server over the encrypted SSL transport. Set `SECURITY_AUTHENTICATION` to `serviceCtx.SSL_CREDENTIAL`.
- Provide a username and password in an explicitly activated login protocol.

Note: The `Login` class serves as an implementation of the client side of the login handshaking protocol and as a proxy object for calling the server login object. This component is packaged in the `aurora_client.jar` file. All Oracle8i ORB applications must import this library.

Username Sent by Setting JNDI Properties for the Login Protocol

A client can use the login protocol to authenticate itself to the Oracle8i data server. You can use the login protocol either with or without SSL encryption, because a secure handshaking encryption protocol is built in to the login protocol.

If your application requires an SSL connection for client-server data security, specify the **SSL_LOGIN** service context value for the `SECURITY_AUTHENTICATION` property that is passed when the JNDI initial context is obtained. The following example defines the connection to be SSL-enabled for the login protocol. Notice that the username and password are set.

```
Hashtable env = new Hashtable();
env.put(javax.naming.Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(javax.naming.Context.SECURITY_PRINCIPAL, username);
env.put(javax.naming.Context.SECURITY_CREDENTIALS, password);
env.put(javax.naming.Context.SECURITY_AUTHENTICATION, ServiceCtx.SSL_LOGIN);
Context ic = new InitialContext(env);
...
```

If your application does not use an SSL connection, specify **NON_SSL_LOGIN** within the `SECURITY_AUTHENTICATION` parameter as shown below:

Note: The login handshaking is secured by encryption, but the remainder of the client-server interaction is not secure.

```
env.put(javax.naming.Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
```

When you specify values for all four JNDI Context variables—`URL_PKG_PREFIXES`, `SECURITY_PRINCIPAL`, `SECURITY_CREDENTIALS`, and `SECURITY_AUTHENTICATION`—the first invocation of the `Context.lookup()` method performs a login automatically.

If the client setting up the connection is not using JNDI look up because it already has an IOR, the user that gave them the IOR for the object should have also passed in a `Login` object that exists in the same session as the active object. You must

provide the username and password in the `authenticate` method of the `Login` object, before invoking the methods on the active object.

Logging In and Out of the JServer Session If the session owner wishes to exit the session, the owner can use the `logout` method of the `LogoutServer` object, which is pre-published as `"/etc/logout"`. You use the `LogoutServer` object to exit the session. Only the session owner is allowed to logout. Any other owner receives a `NO_PERMISSION` exception.

The `LogoutServer` object is analogous to the `LoginServer` object, which is pre-published as `"/etc/login"`. You can use the `LoginServer` object to retrieve the `Login` object, which is used to authenticate to the server. This is an alternative method to using the `Login` object within the JNDI lookup.

The following example shows how a client can authenticate using the `LoginServer` object and can exit the session through the `LogoutServer` object.

```
import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.AuroraServices.LogoutServer;
...
// To log in using the LoginServer object
LoginServer loginServer = (LoginServer)ic.lookup(serviceURL + "/etc/login");
Login login = new Login(loginServer);
System.out.println("Logging in ..");
login.authenticate(user, password, null);
...
//To logout using the LogoutServer
LogoutServer logout = (LogoutServer)ic.lookup(serviceURL + "/etc/logout");
logout.logout();
```

Username Sent Implicitly by using Credentials

Using the `ServiceCtx.SSL_CREDENTIAL` authentication type means that the username, password, and, potentially, a role are passed to the server on the first request. Because this information is passed over an SSL connection, the password is encrypted by the transfer protocol, and there is no need for the handshaking that the `Login` protocol uses. This is slightly more efficient and is recommended for SSL connections.

Username Sent by Explicitly Activating a Login Object

You can explicitly create and populate a `Login` object for the database login. Typically, you would do this if you wanted to create and use more than a single

session from a client. The following example shows a client creating and logging on to two different sessions. To do this, you must perform the following steps:

1. Create the initial context.
2. Perform a look up on a URL for the destination database.
3. On this database service context, create two subcontexts—one for each session.
4. Login to each session using a Login object, providing a username and password.

Note: The username and password for both sessions are identical because the destination database is the same database. If the client were to connect to two different databases, the username and password may need to be different for logging on.

```
// Prepare a simplified Initial Context as we are going to do
// everything by hand
Hashtable env = new Hashtable ();
env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
Context ic = new InitialContext (env);

// Get a SessionCtx that represents a database instance
ServiceCtx service = (ServiceCtx)ic.lookup (serviceURL);

// Create and authenticate a first session in the instance.
SessionCtx session1 = (SessionCtx)service.createSubcontext (":session1");
LoginServer login_server1 = (LoginServer)session1.activate ("etc/login");
Login login1 = new Login (login_server1);
login1.authenticate (user, password, null);

// Create and authenticate a second session in the instance.
SessionCtx session2 = (SessionCtx)service.createSubcontext (":session2");
LoginServer login_server2 = (LoginServer)session2.activate ("etc/login");
Login login2 = new Login (login_server2);
login2.authenticate (user, password, null);

// Activate one Hello object in each session
Hello hello1 = (Hello)session1.activate (objectName);
Hello hello2 = (Hello)session2.activate (objectName);
```

Using Certificates for Client Authentication

Client authentication through certificates requires the client sending a certificate or certificate chain to the server; the server verifies that the client is truly who the client said it was and that it is trusted.

Note: All certificates, trustpoints, and the private key should be in base-64 encoded format.

You set up the client for certificate authentication through one of the following methods:

- [Specifying Certificates in a File](#)
- [Specifying Certificates in Individual JNDI Properties](#)
- [Specifying Certificates using AuroraCertificateManager](#)

Specifying Certificates in a File

You can set up a file that contains the user certificate, the issuer certificate, the entire certificate chain, an encrypted private key, and the trustpoints. Once created, you can specify that the client use the file during connection handshake for client authentication.

1. Create the client certificate file—This file can be created through an export feature in the Wallet Manager. The Oracle Wallet Manager has an option that creates this file. You must populate a wallet using the Wallet Manager before requesting that the file is created.

After you create a valid wallet, bring up the Wallet Manager and perform the following:

- From the menu bar pull down, click on Operations > Export Wallet.
- Within the filename field, enter the name that you want the certificate file known as.

This creates a base-64 encoded file that contains all certificates, keys, and trustpoints that you added within your wallet. For information on how to create the wallet, see the *Oracle Advanced Security Administrator's Guide*.

2. Specify the client certificates file for the connection—Within the client code, set the `SECURITY_AUTHENTICATION` property to `ServiceCtx.SSL_CLIENT_AUTH`. Provide the appropriate certificates and

trustpoints for the server to authenticate against. Specify the filename and decrypting key in the JNDI properties, as follows:

Values	Set in JNDI Property
Name of the certificate file	SECURITY_PRINCIPAL
Key for decrypting the private key	SECURITY_CREDENTIAL

The following code is an example of how to set up the JNDI properties to define the client certificate file:

```
Hashtable env = new Hashtable();
env.put ( javax.naming.Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
env.put ( javax.naming.Context.SECURITY_PRINCIPAL, <filename> );
env.put ( javax.naming.Context.SECURITY_CREDENTIAL, <decrypting_key> );
env.put ( javax.naming.Context.SECURITY_AUTHENTICATION,
        ServiceCtx.SSL_CLIENT_AUTH );
Context ic = new InitialContext(env);
...

```

For example, if your decrypting key is `welcome12` and the certificate file is `credsFile`, the following two lines would specify these values within the JNDI context:

```
env.put ( Context.SECURITY_CREDENTIALS, "welcome12" );
env.put ( Context.SECURITY_PRINCIPAL, "credsFile" );

```

Specifying Certificates in Individual JNDI Properties

You can provide each certificate, private key, and trust point programmatically, by setting each item individually within JNDI properties. Once you populate the JNDI properties with the user certificate, issuer (Certificate Authority) certificate, encrypted private key, and trust points, they are used during connection handshake for authentication. To identify client-side authentication, set the `SECURITY_AUTHENTICATION` property to `serviceCtx.SSL_CLIENT_AUTH`.

Note: Only a single issuer certificate can be set through JNDI properties.

You can choose any method for setting up your certificates within the JNDI properties. All authorization information values must be set up before initializing the context.

The following example declares the certificates as a static variable. However, this is just one of many options. Your certificate must be base-64 encoded. For example, in the following code, the `testCert_base64` is a base-64 encoded client certificate declared as a static variable. The other variables for CA certificate, private key, and so on, are not shown, but they are defined similarly.

Note: When setting individual certificates as static variables, any certificates for Oracle8i parties do not have any separators. However, if you are setting a certificate for a Visigenic ORB (as the client callback object does in a callback scenario), the certificate must be delineated by "BEGIN CERTIFICATE" and "END CERTIFICATE" identifying lines. See the Visigenic documentation for the format of these strings.

```
final private static String testCert_base64 =
    "MIICejCCAeOgAwIBAgICAmowDQYJKoZIhvcNAQEEBQAwazELMAkGALUEBhMCMVVMx" +
    "DzANBgNVBAoTBk9yYWNsZTEoMCYGA1UECzMFRW50ZXJwcm1zZSBBChBSaWNhdGlv" +
    "biBTZXJ2aWNlczEhMB8GALUEAaMYRUFTUUEgQ2VydG1maWNhdGUGU2VydmlVYMB4X" +
    "DTk5MDgxNzE2MjIxMloXDTAwMDIxMzE2MjIxMlowgYUxCzAJBgNVBAYTA1VIMRsw" +
    "GQYDVQQKEwJPcmFjbGUgQ29ycG9yYXRpb24xPDA6BgNVBAsUMYoqIFNlY3VyaXR5" +
    "IFRFRU1RJTkgQU5EIEVWQUxVQVRJT04gT05MWSB2ZXJzaW9uMiAqKjEhMBkGALUE" +
    "AxQsdGVzdEB1cy5vcnFjbGUuY29tMHwwDQYJKoZIhvcNAQEEBQADawAwaAJhANG1" +
    "Kk2K7u00tI/UBYrmTe89LVRrG83Eb0/wY3xWGelkBeEUTwW57a26u2M9LZAfmT91" +
    "e8Afksqc4qQW23Sjxyo40bQK3Kth6y1NJgovBgfMu1YGtDHaSn2VEg8p58g+nwID" +
    "AQABozYwNDARBg1ghkgBhvCAQEEBAMCAwHwYDVR0jBBGwFoAUDCHwEuJfIFXD" +
    "a7tuYN08b0w1EYwwDQYJKoZIhvcNAQEEBQADgYEARC5rWKge5trqgZ18onldinCg" +
    "Fof6D/qFT9b6Cex5JK3a2dEekg/P/KqDINyifIZL0DV7z/XCK6PQDLwYcVqSSK/m" +
    "487qjdh+zM5X+1DaJ+ROhqOOX54UpiAhAleRMdLT5KuXV6AtAx6Q2mc8k9bzFzww" +
    "eR3uI+i5Tn0dKgxhCZU=\n";

Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.SSL_CLIENT_AUTH);
//decrypting key
env.put(Context.SECURITY_CREDENTIALS, "welcome12");

// you may also set the certificates individually, as shown bellow.
//User certificate
env.put(ServiceCtx.SECURITY_USER_CERT, testCert_base64);
//Certificate Authority's certificate
env.put(ServiceCtx.SECURITY_CA_CERT, caCert_base64);
//Private key
```

```
env.put(ServiceCtx.SECURITY_ENCRYPTED_PKEY, encryptedPrivateKey_base64);  
// setup the trust point  
env.put(ServiceCtx.SECURITY_TRUSTED_CERT, trustedCert);  
  
Context ic = new InitialContext(env);
```

Specifying Certificates using `AuroraCertificateManager`

CORBA clients that do not use JNDI can use `AuroraCertificateManager` for setting the user and issuer certificates, the encrypted private key, and the trust points.

`AuroraCertificateManager` maintains certificates for your application. For the certificates to be passed on the SSL handshake for the connection, you must set the certificates before an SSL connection is made. Setting up a certificate in this manner is only required if the following is true:

- The client sets its certificates through `AuroraCertificateManager` if client-side authentication is required, and the client does not want to use JNDI properties for setting certificates.
- The server sets its certificates through `AuroraCertificateManager` if it is executing a callout or a callback. The typical server-side authentication for a simple client/server exchange is taken care of by the database wallet. However, if this server intends to act as a client by executing a callout or callback, it needs to set certificates identifying itself; it cannot use the database certificate that is contained in the wallet.

`AuroraCertificateManager` Class

The methods offered by this object allow you to:

- Set the SSL protocol version. The default is Undetermined.
- Set the private key and certificate chain.
- Require that client applications authenticate themselves by presenting their certificate chain. This method is used only by servers.

Invoking the `ORB.resolve_initial_references` method with the parameter `SSLCertificateManager` will return an object that can be narrowed to a `AuroraCertificateManager`. [Example 6-1](#) shows a code example of the following methods.

addTrustedCertificate

This method adds the specified certificate as a trusted certificate. The certificate must be in DER encoded format. The client adds trustpoints through this method for server-side authentication.

When your client wants to authenticate a server, the server sends its certificate chain to the client. You might not want to check every certificate in the chain. For example, you have a chain composed of the following certificates: Certificate Authority, enterprise, business unit, a company site, and a user. If you trust the company site, you would check the user's certificate, but you might stop checking the chain when you get to the company site's certificate, because you accept the certificates above the company sites in the hierarchical chain.

Syntax

```
void addTrustedCertificate(byte[] derCert);
```

Parameter	Description
derCert	The DER encoded byte array containing the certificate.

requestClientCertificate

This method is invoked by servers that wish to require certificates from client applications. This method is not intended for use by client applications.

Note: The `requestClientCertificate` method is not currently required, because the `SQLNET.ORA` and `LISTENER.ORA` configuration parameter `SSL_CLIENT_AUTHENTICATION` performs its function.

Syntax

```
void requestClientCertificate(boolean need);
```

Parameter	Description
need	If true, the client must send a certificate for authentication. If false, no certificate is requested from the client.

setCertificateChain

This method sets the certificate chain for your client application or server object and can be invoked by clients or by servers. The certificate chain always starts with the Certificate Authority certificate. Each subsequent certificate is for the issuer of the preceding certificate. The last certificate in the chain is the certificate for the user or process.

Syntax

```
void setCertificateChain(byte[][] derCertChain)
```

Parameter	Description
derCertChain	A byte array containing an array of certificates.

setEncryptedPrivateKey

This method sets the private key for your client application or server object. You must specify the key in PKCS5 or PKCS8 format.

Syntax

```
void setEncryptedPrivateKey(byte[] key, String password);
```

Parameter	Description
key	The byte array that contains the encrypted private key.
password	A string containing a password for decrypting the private key.

setProtocolVersion

This method sets the SSL protocol version that can be used for the connection. A 2.0 Client trying to establish an SSL connection with a 3.0 Server will fail and the converse. We recommend using `Version_Undetermined`, because it lets the peers establish an SSL connection whether they are using the same protocol version or not. `SSL_Version_Undetermined` is the default value.

Syntax

```
void setProtocolVersion(int protocolVersion);
```

Parameter	Description
protocolVersion	<p>The protocol version being specified. The value you supply is defined in <code>oracle.security.SSL.OracleSSLProtocolVersion</code>. This class defines the following values:</p> <ul style="list-style-type: none"> ▪ <code>SSL_Version_Undetermined</code>: Version is undetermined. This is used to connect to SSL 2.0 and SSL 3.0 peers. This is the default version. ▪ <code>SSL_Version_3_0_With_2_0_Hello</code>: Not supported. ▪ <code>SSL_Version_3_0</code>: Used to connect to 3.0 peers only. ▪ <code>SSL_Version_2_0</code>: Not supported.

Example 6-1 Setting SSL Security Information Using `AuroraCertificateManager`

This example does the following:

1. Retrieve the `AuroraCertificateManager`.
2. Initialize this client's SSL information:
 - a. Set the certificate chain through `setCertificateChain`.
 - b. Set the trustpoint through `addTrustedCertificate`.
 - c. Set the private key through `setEncryptedPrivateKey`.

```
// Get the certificate manager
AuroraCertificateManager cm = AuroraCertificateManagerHelper.narrow(
    orb.resolve_initial_references("AuroraSSLCertificateManager"));

BASE64Decoder decoder = new BASE64Decoder();
byte[] userCert = decoder.decodeBuffer(testCert_base64);
byte[] caCert = decoder.decodeBuffer(caCert_base64);

// Set my certificate chain, ordered from CA to user.
byte[][] certificates = {
    caCert, userCert
};
cm.setCertificateChain(certificates);
cm.addTrustedCertificate(caCert);

// Set my private key.
byte[] encryptedPrivateKey =
decoder.decodeBuffer(encryptedPrivateKey_base64);

cm.setEncryptedPrivateKey(encryptedPrivateKey, "welcome12");
```

Server-Side Authentication

The server can require a different type of authentication depending on its role. If you are utilizing the database as a server in a typical client/server environment, you use certificates that are set within a wallet for the database for server-side authentication. However, if you are using the server to callout to another object or callback to an object on the client, the server is now acting as a client and so requires its own identifying certificates. That is, in a callout or callback scenario, the server cannot use the wallet generated for database server-side authentication.

Server activity	Authentication method
Typical client/server	Use database wallet generated by Oracle Wallet Manager
Callout to another object	Set identifying certificates using either JNDI properties or <code>AuroraCurrentManager</code> class.
Callback to client object	Set identifying certificates using <code>AuroraCurrentManager</code> class.

The following sections describe this in more detail:

- Typical Client/Server
- Callouts using Security
- Callbacks using Security

Typical Client/Server

Server-side authentication takes place when the server provides certificates for authentication to the client. When requested, the server will authenticate itself to the client, also known as server-side authentication, by providing certificates to the client. The SSL layer authenticates both peers during the connection handshake. The client requests server-side authentication by setting any of the `SSL_*` values in the JNDI property. See ["Using JNDI for Authentication"](#) on page 6-8 for more information on these JNDI values.

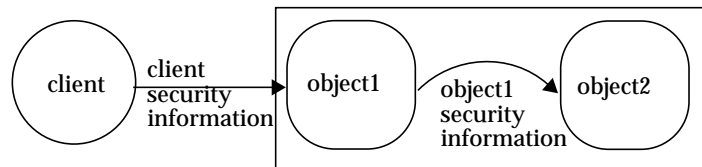
For server-side authentication, you must set up a database wallet with the appropriate certificates, using the Wallet Manager. See the *Oracle Advanced Security Administrator's Guide* for information on how to create a wallet.

Note: If the client wants to verify the server against trustpoints or authorize the server, it is up to the client to set up its trustpoints and parse the server's certificates for authorization. See "[Authorization](#)" on page 6-26 for more information.

Callouts using Security

A callout is when a Java object loaded within the database invokes a method within another Java object. If the original call from the client required a certain level of security—certificate-based or username/password security—the server object is also required to provide the same level of security information for itself before invoking the method on the second server object.

Figure 6–2 *Server callout requires security*



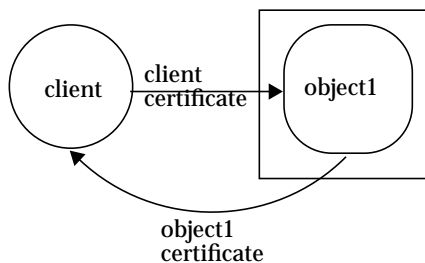
- **Username/password:** If the client sent a username/password combination for authenticating to the database, the server object is also required to send its own username/password combination to the second object. The server object cannot forward along the client's username/password combination, but must supply its own. You can set the username/password combination in the same manner as the client. See "Providing Username and Password for Client-Side Authentication" on page 6-9 for more information.
- **Certificate-based:** Similarly, if the client sent certificates for authentication, the server object must do the same. Additionally, the server must create and send its own certificates, it cannot forward on the client's certificates for authentication. You set up your server object certificates using either the appropriate JNDI properties or the `AuroraCertificateManager` as discussed in "Using Certificates for Client Authentication" on page 6-13.

Callbacks using Security

A callback is when the client passes the server object an object reference to an object that exists on the client. As shown in Figure 6–3, the server object receives the object

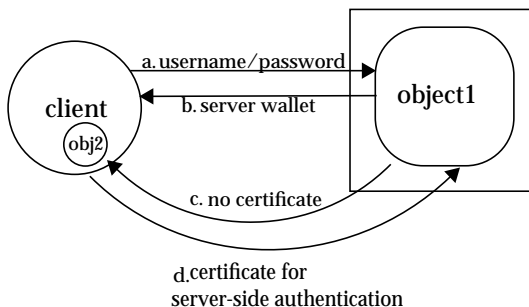
reference and invokes methods. This effectively calls out of the server and back to an object located in the client. See "[Debugging Techniques](#)" on page 2-25 for more information on callbacks.

Figure 6-3 *Server callout requires security*



The type of security you can use for callbacks is certificate-based security over SSL. When you add SSL security to callbacks, you can have one of two situations:

1. Server-side authentication only



- a. The client is not required to authenticate itself with a certificate. However, it must still authenticate itself to the database using a username/password combination.
- b. The server, since server-side authentication is always required with SSL, authenticates itself to the client by providing certificates contained in the database wallet.
- c. When the server calls back to the client, it acts as a client; thus, it is not required to provide certificates for authentication.

- d. The called object, although contained in the client, is the server object in the callback scenario. Thus, since server-side authentication rules hold, the callback object must provide certificates to authenticate itself.

Example 6–2 Callback code with server-side authentication only

The following code shows the client code that performs (a) and (d) steps above. The first half of the client code sets up a username and password for authenticating itself to the database. It retrieves the server object. However, before it invokes the server's method, the last half of the code sets up the client callback object by setting certificates, initializing the BOA, and instantiating the callback object. Finally, the server method is invoked.

```
public static void main (String[] args) throws Exception {
    String serviceURL = args [0];
    String objectName = args [1];
    String user = args [2];
    String password = args [3];

    //set up username/password for authentication to database. Set up
    //security to be SSL_LOGIN - login authentication for client and server-side
    //authentication.
    Hashtable env = new Hashtable ();
    env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    env.put (Context.SECURITY_PRINCIPAL, user);
    env.put (Context.SECURITY_CREDENTIALS, password);
    env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.SSL_LOGIN);
    Context ic = new InitialContext (env);

    // Get the server object before preparing the client object.
    // You have to do it in this order to get the ORB initialized correctly
    Server server = (Server)ic.lookup (serviceURL + objectName);

    // Create the client object and export it to the ORB in the client
    // First, set up the ORB properties for the callback object
    java.util.Properties props = new java.util.Properties();
    props.put("ORBservices", "oracle.aurora.ssl");

    BASE64Decoder decoder = new BASE64Decoder();

    // Initialize the ORB.
    com.visigenic.vbroker.orb.ORB orb = (com.visigenic.vbroker.orb.ORB)
        oracle.aurora.jndi.orb_dep.Orb.init(args, props);

    // Get the certificate manager
```

```
AuroraCertificateManager certificateManager =
    AuroraCertificateManagerHelper.narrow(
        orb.resolve_initial_references("AuroraSSLCertificateManager"));

// Set up client callback certificate chain, ordered from user to CA.
byte[] userCert = decoder.decodeBuffer(testCert_base64);
byte[] caCert = decoder.decodeBuffer(caCert_base64);

// Set my certificate chain, ordered from CA to user.
byte[][] certificates = { caCert, userCert };
cm.setCertificateChain(certificates);
cm.addTrustedCertificate(caCert);

// Set client callback object's private key.
byte[] encryptedPrivateKey=decoder.decodeBuffer(encryptedPrivateKey_base64);

cm.setEncryptedPrivateKey(encryptedPrivateKey, "welcome12");

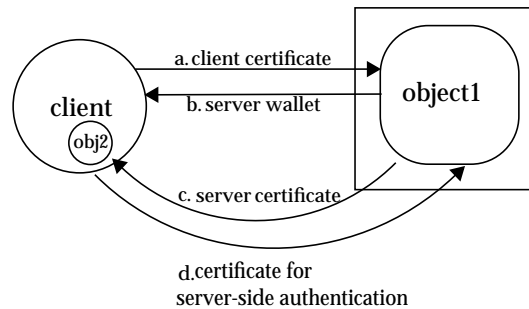
// Initialize the BOA with SSL
org.omg.CORBA.BOA boa = orb.BOA_init("AuroraSSLTSession", null);

//Instantiate the client callback object
ClientImpl client = new ClientImpl ();

//register callback object with BOA
boa.obj_is_ready (client);

// Invoke the server method, passing the client to call us back
System.out.println (server.hello (client));
}
}
```


2. Client-side and Server-side authentication



- a. The client is required to authenticate itself with a certificate.
- b. The server, since server-side authentication is always required with SSL, authenticates itself to the client by providing certificates contained in the database wallet.
- c. When the server calls back to the client, it acts as a client; thus, it is required to provide its own certificates for authentication.
- d. The called object, although contained in the client, is the server object in the callback scenario. Thus, since server-side authentication rules hold, the callback object must provide certificates to authenticate itself.

The code for the client shown in Example 6-2 is the same for this scenario, except that instead of providing a username and password, the client provides certificates.

Since client-side authentication is required and because the server is acting as a client, the server code sets up identifying certificates for itself before invoking the callback object. The server must create and send its own certificates, it cannot forward on the client's certificates for authentication. You set up your server object certificates using either the appropriate JNDI properties or the `AuroraCertificateManager` as discussed in "Using Certificates for Client Authentication" on page 6-13.

Example 6-3 Server code in callback with client-side authentication

The following server code does the following:

1. Retrieve the Oracle8i ORB reference by invoking the `init` method.
2. Retrieve the `AuroraCertificateManager`

3. Set certificates and key through `AuroraCertificateManager` methods.

4. Invoke the client callback method, `hello`.

```
public String hello (Client client) {
    BASE64Decoder decoder = new BASE64Decoder();
    com.visigenic.vbroker.orb.ORB orb = (com.visigenic.vbroker.orb.ORB)
        oracle.aurora.jndi.orb_dep.Orb.init();

    try {
        // Get the certificate manager
        AuroraCertificateManager cm = AuroraCertificateManagerHelper.narrow(
            orb.resolve_initial_references("AuroraSSLCertificateManager"));

        byte[] userCert = decoder.decodeBuffer(testCert_base64);
        byte[] caCert = decoder.decodeBuffer(caCert_base64);

        // Set my certificate chain, ordered from CA to user.
        byte[][] certificates = { caCert, userCert };
        cm.setCertificateChain(certificates);

        // Set my private key.
        byte[] encryptedPrivateKey =
            decoder.decodeBuffer(encryptedPrivateKey_base64);

        cm.setEncryptedPrivateKey(encryptedPrivateKey, "welcome12");

    } catch (Exception e) {
        e.printStackTrace();
        throw new org.omg.CORBA.INITIALIZE( "Couldn't initialize SSL context");
    }

    return "I Called back and got: " + client.helloBack ();
}
```

Authorization

The SSL layer authenticates the peers during the connect handshake. After the handshake, you can be assured that the peer is authenticated to be who they said they are. In addition, since the server has specified, within an Oracle wallet, its trustpoints, the SSL adapter on the server will authorize the client. However, the client has the option of how much authorization is done against the server.

- The client can direct the SSL layer to authorize the server by setting up trustpoints.

- The client can authorize the server itself by extracting the server's certificate chain and parsing through the chain.

Setting up Trust Points

The server automatically has trustpoints established through the installed Oracle Wallet. The trustpoints in the wallet are used to verify the client's certificates. However, if the client wants to verify the server's certificates against certain trustpoints, it can set up its these trustpoints, as follows:

- If server-side authentication is requested, the client does not have any certificates set. Thus, to verify the server's certificates, the client can set a single trustpoint through JNDI, or if it is a pure CORBA application—that does not use JNDI—can add trustpoints through the `AuroraCertificateManager.addTrustedCertificate` method. See [Example 6-4](#) on how to set a single trustpoint through JNDI.
- If client-side authentication is requested, the client has set up certificates. Thus, the client can add trustpoints to the file that contains its certificates, can add a single trustpoint through JNDI, or if it is a pure CORBA application—that does not use JNDI—can add trustpoints through the `AuroraCertificateManager.addTrustedCertificate` method.

If the client does not set up trust points, it does not hinder the authorization. That is, JServer assumes that the client trusts the server.

Example 6-4 Verifying Trustpoints

The following example shows how the client sets up its trustpoints through JNDI. The JNDI `SECURITY_TRUSTED_CERT` property can take only a single certificate.

```
// setup the trust point
env.put(ServiceCtx.SECURITY_TRUSTED_CERT, trustedCert);
```

Parsing through the Server's Certificate Chain

The client retrieves the certificates to perform any authorization checks. In the past, you could retrieve the single issuer certificate. Now, you receive the entire issuer certificate chain. You must parse the certificate chain for the information that you need. You can parse the chain through the `AuroraCurrent` object.

Note: You must configure the database and listener to be SSL-enabled, as described in Chapter 3, "Configuring IIOP Applications".

Note: JDK 1.1 certificate classes were contained within `javax.security.cert`. In JDK 1.2, these classes moved to `java.security.cert`.

`AuroraCurrent` contains three methods for retrieving and managing the certificate chain. For creating and parsing the certificate chain, you can use the `X509Cert` class methods. For information on this class, see Sun Microsystems's JDK documentation. Note that the `X509Cert` class manipulates the certificate chain differently in JDK 1.1 than in Java 2.

The `AuroraCurrent` class methods are as follows:

- `getPeerDERCertChain`—obtain the peer's certificate chain, which enables you to verify that the peer is authorized to access your application methods.
- `getNegotiatedProtocolVersion`—obtain the SSL protocol version being used by the connection, to verify the versioning.
- `getNegotiatedCipherSuite`—obtain the cipher suite used to encrypt messages passed over the connection, to verify that the encryption is strong enough for your purposes.

When the handshake occurs, the protocol version and the type of encryption used is negotiated. The type of encryption can be full or limited encryption, which complies with the United States legal restrictions. After the handshake completes, the `AuroraCurrent` can retrieve what was resolved in the negotiation.

AuroraCurrent Class

The following describes the methods contained within `AuroraCurrent`. See [Example 6-5](#) for a code example of these methods.

`getNegotiatedCipherSuite`

This method obtains the type of encryption negotiated in the handshake with the peer.

Syntax

```
String getNegotiatedCipherSuite(org.omg.CORBA.Object peer);
```

Parameter	Description
peer	The peer from which you obtain the negotiated cipher.

Returns

A string one of the following values:

Export ciphers:

- SSL_RSA_EXPORT_WITH_RC4_40_MD5
- SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
- SSL_DH_anon_EXPORT_WITH_RC4_40_MD5
- SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA
- SSL_RSA_WITH_NULL_SHA
- SSL_RSA_WITH_NULL_MD5

Domestic ciphers

- SSL_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_RSA_WITH_RC4_128_SHA
- SSL_RSA_WITH_RC4_128_MD5
- SSL_RSA_WITH_DES_CBC_SHA
- SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
- SSL_DH_anon_WITH_RC4_128_MD5
- SSL_DH_anon_WITH_DES_CBC_SH

getPeerDERCertificateChain

This method obtains the peer's certificate chain. After retrieving the chain, you can parse through the certificates within the chain, to authorize the peer to your application.

Syntax

```
byte [] [] getPeerDERCertificateChain(org.omg.CORBA.Object peer);
```

Parameter	Description
peer	The peer from which you obtain its certificate chain.

Returns

A byte array containing an array of certificates.

getNegotiatedProtocolVersion

This method obtains the negotiated SSL protocol version of a peer.

Syntax

```
String getNegotiatedProtocolVersion(org.omg.CORBA.Object peer);
```

Parameter	Description
peer	The peer from which you obtain the negotiated protocol version.

Returns

A string with one of the following values:

- `SSL_Version_Undetermined`
- `SSL_Version_3_0`

Example 6–5 Retrieving a Peer's SSL information for Authorization

This example shows how to authorize a peer by retrieving the certificate information using the `AuroraCurrent` object.

1. To retrieve an `AuroraCurrent` object, invoke the `ORB.resolve_initial_references` method with `AuroraSSLCurrent` as the argument.
2. Retrieve the SSL information from the peer through `AuroraCurrent` methods: `getNegotiatedCipherSuite`, `getNegotiatedProtocolVersion`, and `getPeerDERCertChain`.
3. Authorize the peer. You can authorize the peer based on its certificate chain.

Note: This example uses the `x509Certificate` class methods for parsing the certificate chain and is specific to Java 2. If you are using Java 1.1, you must use the `x509Certificate` class methods specific to Java 1.1.

```

static boolean verifyPeerCert(org.omg.CORBA.Object obj) throws Exception
{
    org.omg.CORBA.ORB orb = oracle.aurora.jndi.orb_dep.Orb.init();

    // Get the SSL current
    AuroraCurrent current = AuroraCurrentHelper.narrow
        (orb.resolve_initial_references("AuroraSSLCurrent"));

    // Check the cipher
    System.out.println("Negotiated Cipher: " +
        current.getNegotiatedCipherSuite(obj));
    // Check the protocol version
    System.out.println("Protocol Version: " +
        current.getNegotiatedProtocolVersion(obj));
    // Check the peer's certificate
    System.out.println("Peer's certificate chain : ");
    byte [] [] certChain = current.getPeerDERCertChain(obj);

    //Parse through the certificate chain using the X509Certificate methods
    System.out.println("length : " + certChain.length);
    System.out.println("Certificates: ");
    CertificateFactory cf = CertificateFactory.getInstance("X.509");

    //For each certificate in the chain
    for(int i = 0; i < certChain.length; i++) {
        ByteArrayInputStream bais = new ByteArrayInputStream(certChain[i]);
        Certificate xcert = cf.generateCertificate(bais);
        System.out.println(xcert);
        if(xcert instanceof X509Certificate)
        {
            X509Certificate x509Cert = (X509Certificate)xcert;
            String globalUser = x509Cert.getSubjectDN().getName();
            System.out.println("DN out of the cert : " + globalUser);
        }
    }

    return true;
}

```

Note: The `x509Certificate` class is a Java 2 class. See Sun Microsystems's documentation for more information. In addition, you can find information in the javadoc for `javax.net.ssl`.

Transaction Handling

This chapter covers transaction management for CORBA applications. The CORBA developer can choose to use one of the following transactional APIs provided:

- Java Transaction API (JTA) by Sun Microsystems is a method for creating global transactions in a pure Java environment. JTA can be used in either a single or two-phase commit transaction. In addition, it can be demarcated either from the client or the server object.
- Java Transaction Service (JTS) is a mapping of a subset of the OMG Object Transaction Service (OTS) API that is supplied with Oracle8i JServer. The CORBA developer invokes a transaction service to enable transactional properties for distributed objects in either a Java or non-Java environment. JTS can only be used in a single-phase commit transaction. In addition, it only supports client-side demarcation.

In Oracle8i, Java Transaction API (JTA) 1.0.1 for managing transactions. This chapter assumes that you have a working knowledge of JTA. The discussion focuses mostly on examples and explaining the differences between the Sun Microsystems JTA specification and the Oracle JTA implementation. See

<http://www.javasoft.com> for the Sun Microsystems JTA specification.

- [Transaction Overview](#)
- [JTA Server-Side Demarcation](#)
- [JTA Client-Side Demarcation](#)
- [Configuring Two-Phase Commit Engine](#)
- [Creating DataSource Objects Dynamically](#)
- [Setting the Transaction Timeout](#)
- [Java Transaction Service](#)

- [Transaction Service Interfaces](#)
- [JDBC Restrictions](#)

Transaction Overview

Transactions manage changes to multiple databases within a single application as a unit of work. That is, if you have an application that manages data within one or more databases, you can ensure that all changes in all databases are committed at the same time if they are managed within a transaction.

Transactions are described in terms of ACID properties, which are as follows:

- *Atomic*: all changes to the database made in a transaction are rolled back if any change fails.
- *Consistent*: the effects of a transaction take the database from one consistent state to another consistent state.
- *Isolated*: the intermediate steps in a transaction are not visible to other users of the database.
- *Durable*: when a transaction is completed (committed or rolled back), its effects persist in the database.

The JTA implementation, specified by Sun Microsystems, relies heavily on the JDBC 2.0 specification and XA architecture. The result is a complex requirement on applications in order to ensure that the transaction is managed completely across all databases. Sun Microsystems's specifies Java Transaction API (JTA) 1.0.1 and JDBC 2.0 on <http://www.javasoft.com>.

You should be aware of the following when using JTA within the Oracle8i environment:

- [Global and Local Transactions](#)
- [Demarcating Transactions](#)
- [Transaction Context Propagation](#)
- [Two-Phase Commit](#)
- [Enlisting Resources](#)
- [JTA Limitations](#)

Global and Local Transactions

Whenever your application connected to a database using JDBC or a SQL server, you were creating a transaction. However, the transaction involved only the single database and all updates made to the database were committed at the end of these changes. This is referred to as a local transaction.

A global transaction involves a complicated set of management objects—objects that track all of the objects and databases involved in the transaction. These global transaction objects—`TransactionManager` and `Transaction`—track all objects and resources involved in the global transaction. At the end of the transaction, the `TransactionManager` and `Transaction` objects ensure that all database changes are atomically committed at the same time.

Demarcating Transactions

A transaction is said to be demarcated, which means that each transaction has a definite start and stop point. For example, in an interactive tool such as SQL*Plus, each SQL DML statement implicitly begins a new transaction, if it is not already part of a transaction. A transaction ends when a SQL `COMMIT` or `ROLLBACK` statement is issued.

In a distributed object application, transactions are demarcated differently if the originator is the client or the server. Where the transaction originates defines the transaction as *client-side demarcated* or *server-side demarcated*. See "[JTA Client-Side Demarcation](#)" on page 7-9 and "[JTA Server-Side Demarcation](#)" on page 7-7 for more information.

The client or server object programmatically demarcates the transaction by executing the appropriate `begin` or `commit` methods. Explicit demarcation is discussed further in "[JTA Client-Side Demarcation](#)" on page 7-9.

Note: The originating client or object that starts the transaction must also end the transaction with a commit or rollback. However, the originator can end the transaction in a different method than the originating method. For example, if the client begins the transaction, calls out to a server object, the client must end the transaction after the invoked method returns. The invoked server object cannot end the transaction.

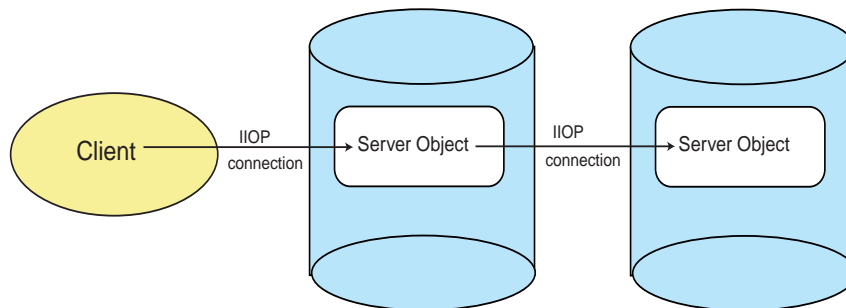
Transaction Context Propagation

When you begin a transaction within either a client or a server instance, JTA denotes the originator in the transaction manager. As the transaction involves more objects and resources, the transaction manager tracks all of these objects and resources in the transaction and manages the transaction for these entities.

When an object calls another object, in order for the invoked object to be included in the transaction, JTA propagates the transaction context to the invoked object. Propagation of the transaction context is necessary for including the invoked object into the global transaction.

As shown in [Figure 7-1](#), if the client begins a global transaction, calls a server object in the database, the transaction context is propagated to the server object. If the server object supports transactions, this object is attached to the transaction manager as involved in the global transaction. If this server object invokes another server object, within the same or a remote database, the transaction context is propagated to this object as well. This ensures that all objects that are supposed to be involved in the global transaction are tracked by the transaction manager.

Figure 7-1 *Connection to an Object over IIOP*



Enlisting Resources

Each resource, including databases, that you want managed in the global transaction must be enlisted. The Oracle8i JTA implementation automatically enlists all databases if you open a JDBC connection to the database within the context of a global transaction. The *Oracle8i JDBC Developer's Guide* contains more information on how to open a JDBC connection to a database within a transaction.

An object—client or server object—can only enlist a database within the global transaction through JDBC 2.0 methods within the `DataSource` object. The `getConnection` method must be invoked after the `begin` method of the `UserTransaction` object.

If your transaction involves more than one database, you must specify an Oracle8i database as the two-phase commit engine. See "[Configuring Two-Phase Commit Engine](#)" on page 7-15 for more information.

Two-Phase Commit

One of the primary advantages for a global transaction is the number of objects and database resources managed as a single unit within the transaction. If your global transaction involves more than one database resource, you must specify a two-phase commit engine, which is an Oracle8i database designated to manage the changes to all databases within the transaction. The two-phase commit engine is responsible for ensuring that when the transaction ends, all changes to all databases are either totally committed or fully rolled back.

On the other hand, if your global transaction has multiple server objects, but only a single database resource, you do not need to specify a two-phase commit engine. The two-phase commit engine is required only to synchronize the changes for multiple databases. If you have only a single database, single-phase commit can be performed by the transaction manager.

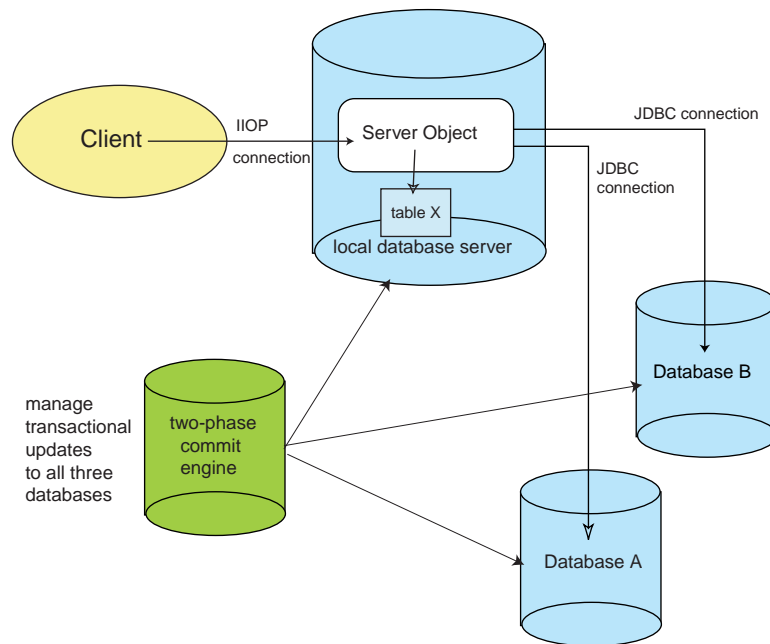
Note: Your two-phase commit engine can be any Oracle8i database. It can be the database where your server object exists, or even a database that is not involved in the transaction at all. See "[Configuring Two-Phase Commit Engine](#)" on page 7-15 for a full explanation of the two-phase commit engine setup.

[Figure 7-2](#) shows three databases enlisted in a global transaction and another database that is designated as the two-phase commit engine. All databases, including the local database, are enlisted when a JDBC connection is opened after

the global transaction starts. See ["Enlisting Resources"](#) on page 7-5 for more information on database enlistment.

When the global transaction ends, the two-phase commit engine ensures that all changes made to the databases A, B, and the local are committed or rolled back simultaneously.

Figure 7-2 Two-Phase Commit for Global Transactions



JTA Limitations

The following are the portions of the JTA specification that Oracle8i does not support.

Nested Transactions

Nested transactions are not supported in this release. If you attempt to begin a new transaction before committing or rolling back any existing transaction, the transaction service throws a `NotSupportedException` exception.

Interoperability

The transaction services supplied with this release do not interoperate with other JTA implementations.

JTA Server-Side Demarcation

To retrieve any objects or database resources, you can perform in-session activation or remote lookup.

- **In-session activation:** Server objects can be local or remote, the `UserTransaction` is always local (only necessary for bean-managed transactional beans), and `DataSource` objects can be local or remote. For local retrieval of any of these objects, you can activate these objects within this session. The namespace is always local, so the `lookup` requires only the JNDI name. In addition, the initial context can be created without any set environment.
- **Remote retrieval:** The server object and/or the `DataSource` object is remote, so you must still provide all of the same information that was provided in the client scenario: authentication information, namespace URL, the `"jdbc_access://"` prefix, and registering the `OracleDriver`. For remote retrieval, perform exactly as demonstrated in the ["JTA Client-Side Demarcation"](#) on page 7-9.

Example 7-1 Server-Side Demarcation for Single-Phase Commit

Before starting the client, you must first bind the `UserTransaction` and `DataSource` objects in the namespace.

Bind UserTransaction Object in the Namespace

You bind the `UserTransaction` object in the namespace through the `bindut` command of the `sess_sh` tool. To bind a `UserTransaction` object to the name `"/test/myUT"` in the namespace located on `nsHost`, execute the following:

```
sess_sh -service jdbc:oracle:thin:@nsHost:5521:ORCL -user SCOTT -password TIGER
& bindut /test/myUT
```

Verify that the user bound with the `UserTransaction` has `FORCE ANY TRANSACTION` granted to the user that bound this object. This privilege enables the user to commit this transaction. In this example, you would execute the following:

```
GRANT FORCE ANY TRANSACTION TO SCOTT
```

Note: The client needs the same information to retrieve the `UserTransaction` as you give within the `bindut` command.

Bind DataSource Object in the Namespace

Use the `bindds` command of the `sess_sh` tool to bind an `DataSource` object in the namespace. The full command is detailed in the *Oracle8i Java Tools Reference*.

To bind a `DataSource` object for a single-phase commit transaction with the `empHost` database to the name `"/test/empDatabase"` in the namespace located on `nsHost`, execute the following:

```
sess_sh -service jdbc:oracle:thin:@nsHost:5521:ORCL -user SCOTT -password TIGER
& bindds /test/empDatabase -url jdbc:oracle:thin:@empHost:5521:ORCL -dstype jta
```

After binding the `DataSource` object in the namespace, the server can enlist the database within a global transaction.

Note: If using more than one database, you will need to setup for a two-phase commit. See "[Configuring Two-Phase Commit Engine](#)" on page 7-15 for more information.

Developing the Server Application

The following example demonstrates a server object performing an in-session lookup of the `UserTransaction` and `DataSource` objects. This example uses a single phase commit transaction.

Note: To modify this for two-phase commit, supply a username and password within the environment passed into the initial context constructor.

```
ic = new InitialContext ( );

// lookup the usertransaction
UserTransaction ut = (UserTransaction)ic.lookup ("/test/myUT");
...
ut.begin ();

// Retrieve the DataSource
```



```
DataSource ds = (DataSource)ic.lookup ("/test/empDB");

// Get connection to the database through DataSource.getConnection
Connection conn = ds.getConnection ();
```

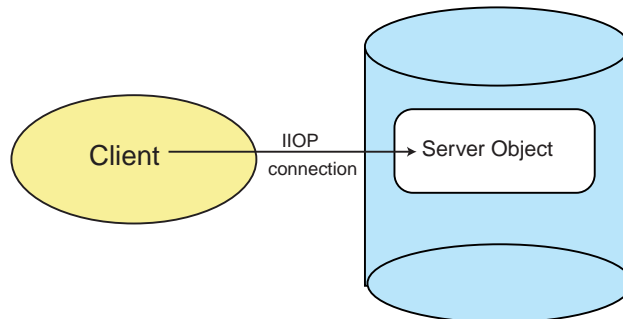
JTA Client-Side Demarcation

For JTA, client-side demarcated transactions are explicitly demarcated programmatically through the `UserTransaction` object, which must be bound with the `bindut` command into the namespace. With client-side transaction demarcation, the client controls the transaction. The client starts a global transaction by invoking the `UserTransaction begin` method; it ends the transaction by invoking either the `commit` or `rollback` methods.

In addition, the client must always set up an environment including a `Hashtable` with authentication information and namespace location URL. It must also register an `OracleDriver` when retrieving the transaction objects from the namespace.

Figure 7-3 shows a client invoking a server object. The client starts a global transaction, then invokes the object. The transactional context is propagated to include the server object.

Figure 7-3 Client Demarcated Global Transaction



The following must occur for the client to demarcate the transaction:

1. Initialize a `Hashtable` environment with the namespace address and authentication information.
2. Register the `OracleDriver`.

3. Retrieve the `UserTransaction` object from the namespace within the client logic. When you retrieve the `UserTransaction` object from any client, the URL must consist of `"jdbc_access://"` prefix before the JNDI name.
4. Start the global transaction within the client using `UserTransaction.begin()`.
5. Retrieve the server object.
6. Invoke any object methods to be included in the transaction.
7. End the transaction through `UserTransaction.commit()` or `UserTransaction.rollback()`.

Example 7-2 shows a client that invokes a server object within the transaction.

Example 7-2 Employee Client Code for Client Demarcated Transaction

After binding the `UserTransaction` object, your client code can retrieve the `UserTransaction` object and start a global transaction. Since the client is retrieving the `UserTransaction` object from a remote site, the lookup requires authentication information, location of the namespace, the `OracleDriver` registration, and the `"jdbc_access://"` prefix.

```
import employee.*;
import java.sql.DriverManager;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.transaction.UserTransaction;
import java.sql.SQLException;
import javax.naming.NamingException;
import oracle.aurora.jndi.jdbc_access.jdbc_accessURLContextFactory;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

public class Client
{
    public static void main (String[] args) throws Exception
    {
        UserTransaction ut = null;
        EmployeeInfo info;
        String sessiiopURL = args [0];
        String objectName = args [1];

        //Set up the service URL to where the UserTransaction object
        //is bound. Since from the client, the connection to the database
        //where the namespace is located can be communicated with over either
```

```
//a Thin or OCI8 JDBC driver. This example uses a Thin JDBC driver.
String namespaceURL = "jdbc:oracle:thin:@nsHost:1521:ORCL";

// lookup usertransaction object in the namespace
try {
    //1.(a) Authenticate to the database.
    // create InitialContext and initialize for authenticating client
    Hashtable env = new Hashtable ();
    env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    env.put (Context.SECURITY_PRINCIPAL, "SCOTT");
    env.put (Context.SECURITY_CREDENTIALS, "TIGER");
    env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
    //1.(b) Specify the location of the namespace where the transaction objects
    // are bound.
    env.put(jdbc_accessURLContextFactory.CONNECTION_URL_PROP, namespaceURL);
    Context ic = new InitialContext (env);

    //2. Register a JDBC OracleDriver. Required for JDBC connection to retrieve
    // UserTransaction from namespace.
    DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());

    //3. Retrieve the UserTransaction object from JNDI namespace
    ut = (UserTransaction)ic.lookup ("jdbc_access://test/myUT");

    //4. Start the transaction
    ut.begin();

    //5. Retrieve the server object reference
    // lookup employee object in the namespace
    Employee employee = (Employee)ic.lookup
        ("sess_iiop://myhost:1521:orcl/test/employee");

    //6. Perform bean business logic.
    // retrieve the info
    info = employee.getEmployee ("SCOTT");
    System.out.println ("Before Update: " + info.name + " " + info.salary);

    // change the salary and update it
    System.out.println ("Increase by 10%");
    info.salary += (info.salary * 10) / 100;
    employee.updateEmployee (info);

    //7. End the transaction
    //Commit the updated value
    ut.commit ();
}
```

```
}
```

The transaction context is propagated to the object when the client invokes it.

JTA Client-Side Demarcation Including Databases

The previous example showed how a transaction context was propagated to server objects from a client within the JTA global transaction. When you execute the server object, the transaction is propagated over the IIOP transport layer. In addition to invoking IIOP server objects, you may wish to update databases over JDBC connections. The databases that the object accesses must be enlisted to be included within the global transaction.

If you access an Oracle8i database from the server that should be included in the transaction, you must open the connection to the database after the global transaction starts.

Note: At this time, the Oracle JTA implementation does not support including non-Oracle databases in a global transaction.

This section shows how you enlist databases using a JDBC connection in tandem with the IIOP server object propagation.

To include a remote database within the transaction from a client, you must use a `DataSource` object, which has been bound in the namespace as a JTA `DataSource`. Then, invoke the `getConnection` method of the `DataSource` object after the transaction has started, and the database is included in the global transaction.

The following must occur in the client runtime to demarcate the transaction:

1. Initialize a `Hashtable` environment with the namespace address and authentication information.
2. Register the `OracleDriver`.
3. Retrieve the `UserTransaction` object from the namespace within the client logic. When you retrieve the `UserTransaction` object from the client, the URL must consist of "jdbc_access://" prefix before the JNDI name.
4. Start the global transaction within the client using `UserTransaction.begin()`.
5. Enlist any database resources to be included in the transaction by opening a connection to the specified database, as follows:

- a. Retrieve the `DataSource` object from the namespace within the client logic. When you retrieve the `DataSource` object from any client, the URL must consist of `"jdbc_access://"` prefix before the JNDI name.
 - b. Open a connection to the database through `DataSource.getConnection` method.
6. Retrieve the object reference.
 7. Invoke any object methods to be included in the transaction.
 8. Invoke SQL DML statements against any enlisted databases.
 9. End the transaction through `UserTransaction.commit()` or `UserTransaction.rollback()`.

Example 7-3 shows a client that invokes a server object and enlists a single database within the transaction.

Example 7-3 Employee Client Code for Client Demarcated Transaction

Before starting the client, you must first bind the `UserTransaction` and `DataSource` objects in the namespace. The following example follows the steps listed in "[JTA Client-Side Demarcation Including Databases](#)" on page 7-12.

```
import employee.*;
import java.sql.DriverManager;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.transaction.UserTransaction;
import java.sql.SQLException;
import javax.naming.NamingException;
import oracle.aurora.jndi.jdbc_access.jdbc_accessURLContextFactory;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

public class Client
{
    public static void main (String[] args) throws Exception
    {
        UserTransaction ut = null;
        EmployeeInfo info;
        String sessiiopURL = args [0];
        String objectName = args [1];
        String dsName = args [2];

        //Set up the service URL to where the UserTransaction object
```

```
//is bound. Since from the client, the connection to the database
//where the namespace is located can be communicated with over either
//a Thin or OCI8 JDBC driver. This example uses a Thin JDBC driver.
String namespaceURL = "jdbc:oracle:thin:@nsHost:1521:ORCL";

// lookup usertransaction object in the namespace
try {
    //1.(a) Authenticate to the database.
    // create InitialContext and initialize for authenticating client
    Hashtable env = new Hashtable ();
    env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    env.put (Context.SECURITY_PRINCIPAL, "SCOTT");
    env.put (Context.SECURITY_CREDENTIALS, "TIGER");
    env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
    //1.(b) Specify the location of the namespace where the transaction objects
    // are bound.
    env.put(jdbc_accessURLContextFactory.CONNECTION_URL_PROP, namespaceURL);
    Context ic = new InitialContext (env);

    //2. Register a JDBC OracleDriver. This is a requirement for retrieving
    // the UserTransaction and DataSource objects from the namespace over
    // a JDBC connection.
    DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());

    //3. Retrieve the UserTransaction object from JNDI namespace
    ut = (UserTransaction)ic.lookup ("jdbc_access://test/myUT");

    //4. Start the transaction
    ut.begin();

    //5.(a) Retrieve the DataSource (that was previously bound with bindds in
    // the namespace. After retrieving the DataSource...
    // get a connection to a database. You need to provide authentication info
    // for a remote database lookup, similar to what you would do from a client.
    // In addition, if this was a two-phase commit transaction, you must provide
    // the username and password.
    DataSource ds = (DataSource)ic.lookup ("jdbc_access://test/empDB");

    //5.(b). Get connection to the database through DataSource.getConnection
    // in this case, the database requires the same username and password as
    // set in the environment.
    Connection conn = ds.getConnection ();

    //6. Retrieve the server object reference
    // lookup employee object in the namespace
```

```

Employee employee = (Employee)ic.lookup (sessiopURL + objectName);

//7 (a). Perform bean business logic.
// retrieve the info
info = employee.getEmployee ("SCOTT");
System.out.println ("Before Update: " + info.name + " " + info.salary);

// change the salary and update it
System.out.println ("Increase by 10%");
info.salary += (info.salary * 10) / 100;
employee.updateEmployee (info);

//7 (b). Execute SQL statements against the enlisted database.
Statement stmt = conn.createStatement ();
int cnt = stmt.executeUpdate ("insert into my_tab values (39304)");

//8. Close the database connection.
conn.close ();

//9. End the transaction
//Commit the updated value
ut.commit ();
}

```

Configuring Two-Phase Commit Engine

If you have more than a single database involved in your transaction, you must designate a two-phase commit engine for managing all changes to all databases involved in the transaction. A two-phase commit engine is responsible for contacting all of the databases at the end of the transaction and managing the commit or rollback of all updates to all included databases. Thus, this two-phase commit engine must have access to database links to each database included within the transaction.

To configure for a two-phase commit, your system administrator must do the following:

1. Designate an Oracle8i database as the two-phase commit engine.
2. Configure database links from the two-phase commit engine to each database that may be involved in the global transaction. This is necessary for the two-phase commit engine to communicate with each database at the end of the transaction.

3. Provide the database link name in the `-dblink` option of `bindds` for each individual database when binding that database's `DataSource` into the namespace.

```
bindds /test/empDatabase -url jdbc:oracle:thin:@empHost:5521:ORCL
      -dstype jta -dblink 2pcToEmp.oracle.com
```

4. Provide the two-phase commit engine's fully-qualified database address, username, and password when binding the `UserTransaction` into the namespace.

```
bindut /test/myUT -url sess_iiop://dbsun.mycompany.com:2481:ORCL
      -user SCOTT -password TIGER
```

Note: Verify that the user bound with the `UserTransaction` has `FORCE ANY TRANSACTION` granted to the user that bound this object. This privilege enables the user to commit this transaction. In this example, you would execute the following:

```
GRANT FORCE ANY TRANSACTION TO SCOTT
```

Once all of this configuration is complete, your application differs from the single-phase commit scenario in the following aspects:

- If you are demarcating your transaction from the client, you can chose to not supply the username and password in the `UserTransaction` binding; but instead, provide the username/password when retrieving the `UserTransaction` object within the `Hashtable` in the `InitialContext` used when looking up the `UserTransaction`.

The following example shows a server object that performs an in-session activation to retrieve both the `UserTransaction` and `DataSource` objects that have been bound locally. The `UserTransaction` was bound with the two-phase commit engine's URL, username, and password. The `DataSource` objects were all bound with the proper database links.

```
//with the environment set, create the initial context.
InitialContext ic = new InitialContext ();
UserTransaction ut = (UserTransaction)ic.lookup ("/test/myUT");

//With the same username and password for the 2pc engine,
// lookup the local datasource and a remote database.
DataSource localDS = (DataSource)ic.lookup ("/test/localDS");

//remote lookup requires environment setup
```



```

Hashtable env = new Hashtable ();
env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put (Context.SECURITY_PRINCIPAL, user);
env.put (Context.SECURITY_CREDENTIALS, password);
env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
env.put (jdbc_accessURLContextFactory.CONNECTION_URL_PROP, namespaceURL);
Context ic = new InitialContext (env);

//Register a JDBC OracleDriver.
DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
//retrieve the DataSource for the remote database
DataSource remoteDS = (DataSource)ic.lookup ("jdbc_access://test/NewYorkDS");

//retrieve connections to both local and remote databases
Connection localConn = localDS.getConnection ();
Connection remoteConn = remoteDS.getConnection ();
...
//close the connections
localConn.close();
remoteConn.close();

//end the transaction
ut.commit();

```

Creating DataSource Objects Dynamically

If you want to bind only a single `DataSource` object in the namespace to be used for multiple database resources, you must do the following:

1. Bind the `DataSource` without specifying the URL, host, port, SID, or driver type. Thus, you execute the `bindds` tool with only the `-dstype jta` option, as follows:

```

sess_sh -service jdbc:oracle:thin:@nsHost:5521:ORCL -user SCOTT -password
TIGER
& bindds /test/empDatabase -dstype jta

```

2. Retrieve the `DataSource` in your code. When you perform the lookup, you must cast the returned object to `OracleJTADDataSource` instead of `DataSource`. The Oracle-specific version of the `DataSource` class contains methods to set the `DataSource` properties.
3. Set the following properties:
 - URL with the `OracleJTADDataSource.setURL` method

- Host, port, SID, and driver type if you did not set the URL with the following `OracleJTADataSource` methods: `setURL`, `setDatabaseName`, `setPortNumber`, and `setDriverType`
 - Database link if using two-phase commit engine with the `OracleJTADataSource.setDBLink` method
 - Username and password if need to provide authentication information for a two-phase commit engine. This information could have been provided on the initial context environment or can be provided in the `getConnection` method. However, if you want to set it with the `OracleJTADataSource` methods, you can through the `setUser` and `setPassword` methods.
4. Retrieve the connection through the `OracleJTADataSource.getConnection` method as indicated in the other examples.

Example 7-4 Retrieving Generic DataSource

The following example retrieves a generically bound `DataSource` from the namespace using in-session lookup and initializes all relevant fields.

```
//retrieve an in-session generic DataSource object
OracleJTADataSource ds = (OracleJTADataSource)ic.lookup ("/test/genericDS");

//set all relevant properties for my database
//URL is for a local database so use the KPRB URL
ds.setURL ("jdbc:oracle:kprb:");
//Used in two-phase commit, so provide the fully qualified database link that
//was created from the two-phase commit engine to this database
ds.setDBLink("localDB.oracle.com");

//Finally, retrieve a connection to the local database using the DataSource
Connection conn = ds.getConnection ();
```

Setting the Transaction Timeout

A global transaction automatically has an idle timeout of 60 seconds. If the object attached to the transaction is idle for over the timeout limit, the transaction is rolled back. To initialize a different timeout, set the timeout value—in seconds—through the `setTransactionTimeout` method before the transaction is begun. If you change the timeout value after the transaction begins, it will not affect the current transaction. The following example sets the timeout to 2 minutes (120 seconds) before the transaction begins.

```
//create the initial context
InitialContext ic = new InitialContext ( );

//retrieve the UserTransaction object
ut = (UserTransaction)ic.lookup ("/test/myUT");

//set the timeout value to 2 minutes
ut.setTimeout (120);

//begin the transaction
ut.begin

//Update employee table with new employees
updateEmployees(emp, newEmp);

//end the transaction.
ut.commit ();
```

Java Transaction Service

With JTS, you demarcate the transaction off of a transaction context, which you can retrieve from the `TransactionService` object. The transaction context contains the `begin`, `commit`, `rollback`, `suspend`, and `resume` methods. One of the disadvantages to JTS is that you cannot use a two-phase commit engine to coordinate changes to multiple databases. The advantage to JTS is that you can suspend and resume the transaction. Also, because it is specific to CORBA, you can use either Java or non-Java languages in your application.

This implementation of JTS does not manage distributed transactions. Transaction control distributed among multiple database servers, with support for the required two-phase commit protocol, is only available within the JTA implementation.

The JTS transaction API supplied with Oracle8i JServer manages only one resource: an Oracle8i database session. A transaction exists within only a single server, which means that it cannot span multiple servers or multiple database sessions in a single service. Transaction contexts are never propagated outside a server. If a server object calls out to another server, the transaction context is not carried along. However, a transaction can involve one or many objects. The transaction can encompass one or many methods within these objects.

Whether you demarcate the transaction on the client or the server, the following must occur:

1. Initialize the `TransactionService` object.

Oracle8i automatically initializes this object for any server objects; thus, only the client must explicitly initialize this object. The initialization is accomplished through the `AuroraTransactionService.initialize` method.

2. Retrieve the `TransactionService` object through the static `TS.getTS` method.
3. Retrieve the current transaction context through the `TransactionService.getCurrent` method.
4. Manage the transaction through the following transaction context (`Current` class) methods: `begin`, `commit`, `rollback`, `rollback_only`, `suspend`, `resume`.

JTS Client-Side Demarcation

The only difference between client and server-side demarcation is that the client must initialize the `TransactionService` object before retrieving it. The client initializes a `TransactionService` object on the intended server. Since JTS can only manage a transaction within a single server, the client should invoke server objects that exist only on that single server. In addition, any SQL statements executed against the database should also be solely applied to the same server.

The following example demonstrates the steps required for a client-side demarcation:

1. Initialize the `TransactionService` object. The initialization is accomplished through the `AuroraTransactionService.initialize` method.
2. Retrieve the `TransactionService` object through the static `TS.getTS` method.
3. Retrieve the current transaction context through the `TransactionService.getCurrent` method.
4. Manage the transaction through the following transaction context (`Current` class) methods: `begin`, `commit`, `rollback`, `rollback_only`, `suspend`, `resume`.

Example 7-5 Client-Side Demarcation for JTS Example

```
import employee.*;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jts.client.AuroraTransactionService;
import oracle.aurora.jts.util.*;
import javax.naming.Context;
```

```
import javax.naming.InitialContext;
import java.util.Hashtable;

public class Client
{
    public static void main (String[] args) throws Exception {
        if (args.length != 4) {
            System.out.println ("usage: Client serviceURL objectName user password");
            System.exit (1);
        }
        String serviceURL = args [0];
        String objectName = args [1];
        String user = args [2];
        String password = args [3];

        //The environment must be setup with the correct authentication
        //and prefix information before you create the initial context
        Hashtable env = new Hashtable ();
        env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        env.put (Context.SECURITY_PRINCIPAL, user);
        env.put (Context.SECURITY_CREDENTIALS, password);
        env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
        Context ic = new InitialContext (env);

        //provide the initial context and the service URL of the server
        AuroraTransactionService.initialize (ic, serviceURL);

        //Since JTS can only manage transactions on a single server, the
        //destination server object exists on the same server as the transaction
        //service. Thus, you use the same service URL to retrieve the object.
        Employee employee = (Employee)ic.lookup (serviceURL + objectName);
        EmployeeInfo info;

        //Use the static method getTS to retrieve the TransactionService and the
        //static method getCurrent to retrieve the current transaction context.
        //Off of the Current object, you can start the transaction with the begin
        //method. All three methods have been combined as follows:
        TS.getTS ().getCurrent ().begin ();

        //invoke a method on the retrieved server object. Since the object exists
        //on the transaction server, it is included in the transaction.
        info = employee.getEmployee ("SCOTT");
        System.out.println (info.name + " " + " " + info.salary);
        System.out.println ("Increase by 10%");
        info.salary += (info.salary * 10) / 100;
    }
}
```

```
employee.updateEmployee (info);
info = employee.getEmployee ("SCOTT");
System.out.println (info.name + " " + " " + info.salary);

//Finally, commit the transaction with the Current.commit method.
TS.getTS ().getCurrent ().commit (true);
}
}
```

JTS Server-Side Demarcation

Oracle8i initializes the TransactionService for any server object. In the same manner as the client, the server must invoke only other server objects on the same server. SQL statements should also only be applied to the same database.

The following example demonstrates the steps required for a client-side demarcation:

1. Retrieve the TransactionService object through the static `TS.getTS` method.
2. Retrieve the current transaction context through the `TransactionService.getCurrent` method.
3. Manage the transaction through the following transaction context (`Current` class) methods: `begin`, `commit`, `rollback`, `rollback_only`, `suspend`, `resume`.

Example 7-6 Server-Side Demarcation for JTS Example

```
package employeeServer;

import employee.*;
import java.sql.*;
import oracle.aurora.jts.util.*;
import org.omg.CosTransactions.*;

public class EmployeeImpl extends _EmployeeImplBase
{
    Control txn;

    public EmployeeInfo getEmployee (String name) throws SQLException {
        //When the client invokes the getEmployee method, the transaction is started
        //Retrieve the Transaction service through the static getTS method.
        //Retrieve the current transaction context through the getCurrent method.
        //And start the transaction with the Current.begin method. These have
```

```

//been combined into one statement....
TS.getTS ().getCurrent ().begin ();

//Retrieve the employee information given the employee name.
int empno = 0;
double salary = 0.0;
#sql { select empno, sal into :empno, :salary from emp
        where ename = :name };

//At this point, we suspend the transaction to return the employee
//information to the client.
txn = TS.getTS().getCurrent().suspend();
return new EmployeeInfo (name, empno, (float)salary);
}

public void updateEmployee (EmployeeInfo employee) throws SQLException {
//After the client retrieves the employee info, it invokes the updateEmp
//method to change any values.
//The transaction is resumed in this method through the Current.resume,
//which requires the Control object returned on the suspend method.
TS.getTS().getCurrent().resume(txn);

//update the employee's information.
#sql { update emp set ename = :(employee.name), sal = :(employee.salary)
        where empno = :(employee.number) };

//Once finished, complete the transaction with the Current.commit method.
TS.getTS ().getCurrent ().commit (true);
}

```

JTS Limitations

The implementations of JTS that is supplied for this Oracle8i release is intended to support client-side transaction demarcation. It has limitations that you should be aware of when designing your application.

No Distributed Transactions

This implementation of JTS does not manage distributed transactions. Transaction control distributed among multiple database servers, with support for the required two-phase commit protocol, is only available within the JTA implementation.

Resources

The JTS transaction API supplied with Oracle8i JServer manages only one resource: an Oracle8i database session. A transaction cannot span multiple servers or multiple database sessions in a single service.

Transaction contexts are never propagated outside a server. If a server object calls out to another server, the transaction context is not carried along.

However, a transaction can involve one or many objects. The transaction can encompass one or many methods of these objects. The scope of a transaction is defined by a *transaction context* that is shared by the participating objects. For example, your client can invoke one or more objects on the same server within a single session or several objects on the same server within multiple sessions.

Nested Transactions

Nested transactions are not supported in this release. If you attempt to begin a new transaction before committing or rolling back any existing transaction, the transaction service throws a `SubtransactionsUnavailable` exception.

Timeouts

Methods of the JTS that support transaction timeout, such as `setTimeout()`, do not work in this release. You can invoke them from your code, and no exception is thrown, but they have no effect.

Interoperability

The transaction services supplied with this release do not interoperate with other OTS implementations.

Transaction Service Interfaces

Oracle8i supports a version of the JTS. The JTS is a Java mapping of the OMG Object Transaction Service (OTS). There are two classes that the application developer can use:

- [TransactionService](#)
- `UserTransaction`, implemented by `oracle.aurora.jts.client.AuroraTransactionService`

TransactionService

Use the `TransactionService` to initialize a transaction context on the client. Include the `AuroraTransactionService` package in your Java client source with the following **import** statements:

```
import oracle.aurora.jts.client.AuroraTransactionService;
import javax.jts.*;
import oracle.aurora.jts.util.*;
```

These classes are included in the library file `aurora_client.jar`, which must be in the CLASSPATH when compiling and executing all source files that use the JTS.

There is only one method in this package that you can call:

```
public synchronized static void initialize(Context initialContext,
                                           String serviceName)
```

This method initializes the transaction context on a client. The parameters are:

<code>initialContext</code>	The context object returned by a JNDI Context constructor.
<code>serviceName</code>	The complete service name. For example <code>sess_iiop://localhost:2481:ORCL</code>

An example of using `initialize()` is:

```
Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(Context.SECURITY_PRINCIPAL, "scott");
env.put(Context.SECURITY_CREDENTIALS, "tiger");
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
Context initialContext = new InitialContext(env);
AuroraTransactionService.initialize
    (initialContext, "sess_iiop://localhost:2481:ORCL");
```

Using The Java Transaction Service

JTS contains methods that a client-side or server-side object uses to begin transactions, commit or roll back a transaction, and perform utility functions such as setting the transaction timeout. JTS methods should be used in CORBA clients server objects.

The following sections describe the JTS APIs:

- [Required Import Statements](#)

- [Java Transaction Service Methods](#)
- [Current Transaction Methods](#)

Required Import Statements

To use the JTS methods, include the following import statements in your source:

```
import oracle.aurora.jts.util.TS;
import javax.jts.util.*;
import org.omg.CosTransactions.*;
```

The `oracle.aurora.jts.util` package is included in the library file `aurora_client.jar`, which must be in the `CLASSPATH` for all Java sources that use the JTS.

You use the static methods in the `TS` class to retrieve the transaction service.

Java Transaction Service Methods

The JTS includes the following methods:

```
public static synchronized TransactionService getTS()
```

1. The `getTS` method returns a transaction service object.
2. Once a transaction service has been obtained, you can invoke the static method `getCurrent()` on it to return a `Current` pseudo-object, the transaction context.
3. Finally, you can invoke methods to begin, suspend, resume, commit, or roll back the current transaction on the `Current` pseudo-object.

Here is an example that begins a new transaction on a client, starting with getting the JNDI initial context:

```
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jts.client.AuroraTransactionService;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;
...
Context ic = new InitialContext(env);
...
AuroraTransactionService.initialize(ic, serviceURL);
...
Employee employee = (Employee)ic.lookup (serviceURL + objectName);
EmployeeInfo info;
```

```
oracle.aurora.jts.util.TS.getTS().getCurrent().begin();
```

If there is no transaction service available, then `getTS()` throws a `NoTransactionService` exception.

Current Transaction Methods

The methods that you can call to control transactions on the current transaction context are the following:

```
public void begin()
```

Begins a new transaction.

Can throw these exceptions:

- `NoTransactionService`—if you have not initialized a transaction context.
- `SubtransactionsUnavailable`—if you invoke a `begin()` before the current transaction has been committed or rolled back.

See the section "[TransactionService](#)" on page 7-25 for information about initialization.

```
public Control suspend()
```

Suspends the current transaction in the session. Returns a `Control` transaction context pseudo-object. You must save this object reference for use in any subsequent `resume()` invocations. Invoke `suspend()` in this way:

```
org.omg.CosTransactions.Control c =
    oracle.aurora.jts.util.TS.getTS().getCurrent().suspend();
```

`suspend()` can throw these exceptions:

- `NoTransactionService`—if you have not initialized a transaction context.
- `TransactionDoesNotExist`—if not in an active transaction context. This can occur if a `suspend()` follows a previous `suspend()`, with no intervening `resume()`.

If `suspend()` is invoked outside of a transaction context, then a `NoTransactionService` exception is thrown. If `suspend()` is invoked before `begin()` has been invoked, or after a `suspend()`, the a exception is thrown.

```
public void resume(Control which)
```

Resumes a suspended transaction. Invoke this method after a `suspend()`, in order to resume the specified transaction context. The `which` parameter must be the transaction `Control` object that was returned by the previous matching `suspend()` invocation in the same session. For example:

```
org.omg.CosTransactions.Control c =
    oracle.aurora.jts.util.TS.getTS().getCurrent().suspend();
... // do some non-transactional work
oracle.aurora.jts.util.TS.getTS().getCurrent().resume(c);
```

`resume()` can throw:

- `InvalidControl`—if the `which` parameter is not valid, or is null.

```
public void commit(boolean report_heuristics)
```

Commits the current transaction. Set the `report_heuristics` parameter to **false**.

(The `report_heuristics` parameter is set to **true** for extra information on two-phase commits. Because this release of JServer does not support the two-phase commit protocol for distributed objects, use of the `report_heuristics` parameter is not meaningful. It is included for compatibility with future releases.)

`commit()` can throw:

- `HeuristicMixed`—if `report_heuristics` was set **true**, and a two-phase commit is in progress.
- `HeuristicHazard`—if `report_heuristics` was set **true**, and a two-phase commit is in progress.

The `HeuristicMixed` and `HeuristicHazard` exceptions are documented in the OTS specification.

If there is no active transaction, `commit()` throws a `NoTransaction` exception.

```
public void rollback()
```

Rolls back the effects of the current transaction.

Invoking `rollback()` has the effect of ending the transaction, so invoking any JTS method except `begin()` after a `rollback()` throws a `NoTransaction` exception.

If not in a transaction context, `rollback()` throws the `NoTransaction` exception.

```
public void rollback_only() throws NoTransaction {
```

`rollback_only()` modifies the transaction associated with the current thread so that the only possible outcome is to roll back the transaction. If not in a transaction context, `rollback_only()` throws the `NoTransaction` exception.

```
public void set_timeout(int seconds)
```

This method is not supported, and has no effect if invoked. The default timeout value is 60 seconds in all cases.

```
public Status get_status()
```

You can call `get_status()` at any time to discover the status of the current transaction. Possible return values are:

- `javax.transaction.Status.StatusActive`
- `javax.transaction.Status.StatusMarkedRollback`
- `javax.transaction.Status.StatusNoTransaction`

The complete set of status **ints** is defined in `javax.transaction.Status`.

```
public String get_transaction_name() {
```

Invoke `get_transaction_name()` to see the name of the transaction, returned as a `String`. If this method is invoked before a `begin()`, after a `rollback()`, or outside of a transaction context, it returns a null string.

For More Information on JTS

Information on the Java Transaction Service is available at:

<http://java.sun.com/products/jts/index.html>

The Sun JTS specification is available at:

<http://java.sun.com/products/jta/index.html>

The OTS specification is part of the CORBA services specification. Chapter 10 (individually downloadable) contains the OTS specification. Get it at:

<http://www.omg.org/library/csindx.html>

JDBC Restrictions

If you are using JDBC calls in your CORBA server object to update a database, and you have an active transaction context, you should *not* also use JDBC to perform transaction services, by calling methods on the JDBC connection. Do *not* code JDBC transaction management methods. For example:

```
Connection conn = ...  
...  
conn.commit(); // DO NOT DO THIS!!
```

Doing so will cause a `SQLException` to be thrown. Instead, you must commit using the `UserTransaction` object retrieved to handle the global transaction. When you commit using the JDBC connection, you are instructing a local transaction to commit, not the global transaction. When the connection is involved in a global transaction, trying to commit a local transaction within the global transaction causes an error to occur.

In the same manner, you must also avoid doing direct SQL commits or rollbacks through JDBC. Code the object to either handle transactions directly using the `UserTransaction` interface.

Example Code: CORBA

Oracle8i JServer installs several samples under the `demo` directory. Some of these samples are included in this appendix for your perusal.

The examples in the `demo` directory include a UNIX makefile and Windows NT batch file to compile and run each example. You need a Java-enabled Oracle8i database with the standard EMP and DEPT demo tables to run the examples.

The emphasis in these short examples is on demonstrating features of the ORB and CORBA, not on elaborate Java coding techniques. Each of the examples includes a README file that tell you what files the example contains, what the example does, and how to compile and run the example.

- [Basic Example](#)
- [IFR Example](#)
- [Callback Example](#)
- [TIE Example](#)
- [Pure CORBA Client](#)
- [JTA Examples](#)
- [JTS Transaction Example](#)
- [SSL Examples](#)
- [Session Example](#)
- [Applet Example](#)

Basic Example

The following is a Bank example that demonstrates a simple CORBA application. Included is the README, the IDL, the server code, and the client code. Refer to the `demo/corba/basic` directory for the Makefile.

README

bank demonstrates:

This is an Oracle8i-compatible version of the VisiBroker Bank example. The major differences from the Vb example are:

- (1) There is no server main loop. For Oracle8i the "wait-for-activation" loop is part of the IIOP presentation (MTS server).
- (2) `_boa.connect(object)` is used instead of the less portable `_boa_obj_is_ready(object)` in the server object implementation to register the new Account objects.
- (3) The client program contains the code necessary to lookup the AccountManager object (published under `/test/myBank`) and activate it, and to authenticate the client to the server. (Note that object activation and authentication, via `NON_SSL_LOGIN`, happen "under the covers" so to speak on the `lookup()` method invocation.)
- (4) There is also a tie implementation of this example, with the server being `AccountManagerImplTie.java`.

Bank.IDL

```
// Bank.idl

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


Server

The server code is implemented in the following:

AccountManagerImpl.java

```
package bankServer;

import java.util.*;

public class AccountManagerImpl
    extends Bank._AccountManagerImplBase {

    public synchronized Bank.Account open(String name) {

        // Lookup the account in the account dictionary.
        Bank.Account account = (Bank.Account) _accounts.get(name);

        // If there was no account in the dictionary, create one.
        if(account == null) {

            // Make up the account's balance, between 0 and 1000 dollars.
            float balance = Math.abs(_random.nextInt()) % 100000 / 100f;

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

            _orb().connect(account);

            // Print out the new account.
            // This just goes to the system trace file for Oracle 8i.
            System.out.println("Created " + name + "'s account: " + account);

            // Save the account in the account dictionary.
            _accounts.put(name, account);
        }
        // Return the account.
        return account;
    }

    private Dictionary _accounts = new Hashtable();
    private Random _random = new Random();
}
```

AccountImpl.java

```
package bankServer;

public class AccountImpl extends Bank._AccountImplBase {
    public AccountImpl(float balance) {
        _balance = balance;
    }
    public float balance() {
        return _balance;
    }
    private float _balance;
}
```

AccountManagerImplTie.java

```
package bankServer;

import java.util.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class AccountManagerImplTie
    implements Bank.AccountManagerOperations,
    ActivatableObject {

    public synchronized Bank.Account open(String name) {

        // Lookup the account in the account dictionary.
        Bank.Account account = (Bank.Account) _accounts.get(name);

        // If there was no account in the dictionary, create one.
        if(account == null) {

            // Make up the account's balance, between 0 and 1000 dollars.
            float balance = Math.abs(_random.nextInt()) % 100000 / 100f;

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

            org.omg.CORBA.ORB.init().BOA_init().obj_is_ready(account);

            // Print out the new account.
            // This just goes to the system trace file for Oracle 8i.
            System.out.println("Created " + name + "'s account: " + account);

            // Save the account in the account dictionary.
        }
    }
}
```

```

        _accounts.put(name, account);
    }
    // Return the account.
    return account;
}

public org.omg.CORBA.Object _initializeAuroraObject () {
    return new Bank._tie_AccountManager(this);
}

private Dictionary _accounts = new Hashtable();
private Random _random = new Random();
}

```

Client.java

```

// Client.java opens the account through the AccountManager class and manages
// the account through the Account class */

import bankServer.*;
import Bank.*;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class Client
{
    public static void main (String[] args) throws Exception {
        if (args.length != 5) {
            System.out.println("usage: Client serviceURL objectName user password "
+ "accountName");
            System.exit(1);
        }
        String serviceURL = args [0];
        String objectName = args [1];
        String user = args [2];
        String password = args [3];
        String name = args [4];

        Hashtable env = new Hashtable();
        env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    }
}

```

```
env.put(Context.SECURITY_PRINCIPAL, user);
env.put(Context.SECURITY_CREDENTIALS, password);
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);

Context ic = new InitialContext(env);

AccountManager manager =
    (AccountManager)ic.lookup (serviceURL + objectName);

// Request the account manager to open a named account.
Bank.Account account = manager.open(name);

// Get the balance of the account.
float balance = account.balance();

// Print out the balance.
System.out.println
    ("The balance in " + name + "'s account is $" + balance);
}
}
```

IFR Example

The following example shows how to use the IFR. Soft copy is located at `demo/corba/basic/bankWithIFR`.

Bank.IDL

```
// Bank.idl

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

Server

The server code is implemented in the `AccountManager`, `Account`, and `TIE` classes.

AccountManagerImpl.java

```
package bankServer;

import java.util.*;

public class AccountManagerImpl
    extends Bank._AccountManagerImplBase {

    public synchronized Bank.Account open(String name) {

        // Lookup the account in the account dictionary.
        Bank.Account account = (Bank.Account) _accounts.get(name);

        // If there was no account in the dictionary, create one.
        if(account == null) {

            // Make up the account's balance, between 0 and 1000 dollars.
            float balance = Math.abs(_random.nextInt()) % 100000 / 100f;

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

            _orb().connect(account);

            // Print out the new account.
            // This just goes to the system trace file for Oracle 8i.
            System.out.println("Created " + name + "'s account: " + account);

            // Save the account in the account dictionary.
            _accounts.put(name, account);
        }
        // Return the account.
        return account;
    }

    private Dictionary _accounts = new Hashtable();
    private Random _random = new Random();
}
```

AccountImpl.java

```
package bankServer;

public class AccountImpl extends Bank._AccountImplBase {
```

```
public AccountImpl(float balance) {
    _balance = balance;
}
public float balance() {
    return _balance;
}
private float _balance;
}
```

AccountManagerImplTie.java

```
package bankServer;

import java.util.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class AccountManagerImplTie
    implements Bank.AccountManagerOperations,
    ActivatableObject {

    public synchronized Bank.Account open(String name) {

        // Lookup the account in the account dictionary.
        Bank.Account account = (Bank.Account) _accounts.get(name);

        // If there was no account in the dictionary, create one.
        if(account == null) {

            // Make up the account's balance, between 0 and 1000 dollars.
            float balance = Math.abs(_random.nextInt()) % 100000 / 100f;

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

            org.omg.CORBA.ORB.init().BOA_init().obj_is_ready(account);

            // Print out the new account.
            // This just goes to the system trace file for Oracle 8i.
            System.out.println("Created " + name + "'s account: " + account);

            // Save the account in the account dictionary.
            _accounts.put(name, account);
        }
    }
}
```

```

        // Return the account.
        return account;
    }

    public org.omg.CORBA.Object _initializeAuroraObject () {
        return new Bank._tie_AccountManager(this);
    }

    private Dictionary _accounts = new Hashtable();
    private Random _random = new Random();
}

```

Client

The client code is facilitated in the following:

- [Client.java](#)
- [PrintIDL.java](#)

Client.java

```

import bankServer.*;
import Bank.*;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;
import org.omg.CORBA.Repository;

public class Client
{
    public static void main (String[] args) throws Exception {
        if (args.length != 5) {
            System.out.println("usage: Client serviceURL objectName user password "
+ "accountName");
            System.exit(1);
        }
        String serviceURL = args [0];
        String objectName = args [1];
        String user = args [2];
        String password = args [3];
    }
}

```

```
String name = args [4];

Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(Context.SECURITY_PRINCIPAL, user);
env.put(Context.SECURITY_CREDENTIALS, password);
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);

Context ic = new InitialContext(env);

AccountManager manager =
    (AccountManager)ic.lookup (serviceURL + objectName);

// Request the account manager to open a named account.
Bank.Account account = manager.open(name);

// Get the balance of the account.
float balance = account.balance();

// Print out the balance.
System.out.println
    ("The balance in " + name + "'s account is $" + balance);

System.out.println("Calling the implicit method get_interface()");
org.omg.CORBA.InterfaceDef intf = (org.omg.CORBA.InterfaceDef)
    account._get_interface_def();
System.out.println("intf = " + intf.name());

System.out.println("Now explicitly looking up for IFR and printing the");
System.out.println("whole repository");
System.out.println("");

Repository rep = (Repository)ic.lookup(serviceURL + "/etc/ifr");

new PrintIDL(org.omg.CORBA.ORB.init()).print(rep);

}
}
```

PrintIDL.java

```
import java.io.PrintStream;
import java.util.Vector;
import java.io.DataInputStream;
import org.omg.CORBA.Repository;
```



```
public class PrintIDL
{
    private static org.omg.CORBA.ORB _orb;
    private static PrintStream _out = System.out;
    private static int _indent;

    public PrintIDL (org.omg.CORBA.ORB orb) {
        _orb = orb;
    }
    private void println(Object o) {
        for(int i = 0; i < _indent; i++) {
            _out.print(" ");
        }
        _out.println(o);
    }

    private String toIdl(org.omg.CORBA.IDLType idlType) {
        org.omg.CORBA.Contained contained =
org.omg.CORBA.ContainedHelper.narrow(idlType);
        return contained == null ?
            idlType.type().toString() :
            contained.absolute_name();
    }

    public void print(org.omg.CORBA.Container container) throws
org.omg.CORBA.UserException {
        org.omg.CORBA.Contained[] contained =
            container.contents(org.omg.CORBA.DefinitionKind.dk_all, true);
        for(int i = 0; i < contained.length; i++) {
            {
                org.omg.CORBA.ContainedPackage.Description description =
                    contained[i].describe();
                org.omg.CORBA.portable.OutputStream output =
                    _orb.create_output_stream();
                org.omg.CORBA.ContainedPackage.DescriptionHelper.write(output,
                    description);
                org.omg.CORBA.portable.InputStream input =
                    output.create_input_stream();
                org.omg.CORBA.ContainedPackage.Description description2 =
                    org.omg.CORBA.ContainedPackage.DescriptionHelper.read(input);
                org.omg.CORBA.Any any1 = _orb.create_any();
                org.omg.CORBA.ContainedPackage.DescriptionHelper.insert(any1,
                    description);
            }
        }
    }
}
```

```
org.omg.CORBA.Any any2 = _orb.create_any();
org.omg.CORBA.ContainedPackage.DescriptionHelper.insert(any2,
    description2);
if(!any1.equals(any1) ||
    !any1.equals(any2) ||
    !any2.equals(any2) ||
    !any2.equals(any1)) {
    System.out.println("\n*** The descriptions were not equal (1) ***
\n");
}
org.omg.CORBA.ContainedPackage.Description description3 =
    org.omg.CORBA.ContainedPackage.DescriptionHelper.extract(any2);
if(description.kind != description2.kind ||
    !description.value.equals(description3.value)) {
    System.out.println("\n*** The descriptions were not equal (2) ***
\n");
}
}
switch(contained[i].def_kind().value()) {
case org.omg.CORBA.DefinitionKind._dk_Attribute:
    printAttribute(org.omg.CORBA.AttributeDefHelper.narrow(contained[i]));
    break;
case org.omg.CORBA.DefinitionKind._dk_Constant:
    printConstant(org.omg.CORBA.ConstantDefHelper.narrow(contained[i]));
    break;
case org.omg.CORBA.DefinitionKind._dk_Exception:
    printException(org.omg.CORBA.ExceptionDefHelper.narrow(contained[i]));
    break;
case org.omg.CORBA.DefinitionKind._dk_Interface:
    printInterface(org.omg.CORBA.InterfaceDefHelper.narrow(contained[i]));
    break;
case org.omg.CORBA.DefinitionKind._dk_Module:
    printModule(org.omg.CORBA.ModuleDefHelper.narrow(contained[i]));
    break;
case org.omg.CORBA.DefinitionKind._dk_Operation:
    printOperation(org.omg.CORBA.OperationDefHelper.narrow(contained[i]));
    break;
case org.omg.CORBA.DefinitionKind._dk_Alias:
    printAlias(org.omg.CORBA.AliasDefHelper.narrow(contained[i]));
    break;
case org.omg.CORBA.DefinitionKind._dk_Struct:
    printStruct(org.omg.CORBA.StructDefHelper.narrow(contained[i]));
    break;
case org.omg.CORBA.DefinitionKind._dk_Union:
    printUnion(org.omg.CORBA.UnionDefHelper.narrow(contained[i]));
```

```

        break;
    case org.omg.CORBA.DefinitionKind._dk_Enum:
        printEnum(org.omg.CORBA.EnumDefHelper.narrow(contained[i]));
        break;
    case org.omg.CORBA.DefinitionKind._dk_none:
    case org.omg.CORBA.DefinitionKind._dk_all:
    case org.omg.CORBA.DefinitionKind._dk_Typedef:
    case org.omg.CORBA.DefinitionKind._dk_Primitive:
    case org.omg.CORBA.DefinitionKind._dk_String:
    case org.omg.CORBA.DefinitionKind._dk_Sequence:
    case org.omg.CORBA.DefinitionKind._dk_Array:
    default:
        break;
    }
}
}

private void printConstant(org.omg.CORBA.ConstantDef def) throws
    org.omg.CORBA.UserException {
    println("const " + toIdl(def.type_def()) + " " + def.name() + " = " +
        def.value() + ";");
}

private void printStruct(org.omg.CORBA.StructDef def) throws
    org.omg.CORBA.UserException {
    println("struct " + def.name() + " {");
    _indent++;
    org.omg.CORBA.StructMember[] members = def.members();
    for(int j = 0; j < members.length; j++) {
        println(toIdl(members[j].type_def) + " " + members[j].name + ";");
    }
    _indent--;
    println("};");
}

private void printUnion(org.omg.CORBA.UnionDef def) throws
    org.omg.CORBA.UserException {
    println("union " + def.name() + " switch(" +
    toIdl(def.discriminator_type_def()) + ") {");
    org.omg.CORBA.UnionMember[] members = def.members();
    int default_index = def.type().default_index();
    _indent++;
    for(int j = 0; j < members.length; j++) {
        if(j == default_index) {
            println("default:");
        }
    }
}

```

```
    }
    else {
        println("case " + members[j].label + ":");
    }
    _indent++;
    println(toIdl(members[j].type_def) + " " + members[j].name + ";");
    _indent--;
}
_indent--;
println("};");
}

private void printException(org.omg.CORBA.ExceptionDef def) throws
    org.omg.CORBA.UserException {
    println("exception " + def.name() + " {");
    _indent++;
    org.omg.CORBA.StructMember[] members = def.members();
    for(int j = 0; j < members.length; j++) {
        println(toIdl(members[j].type_def) + " " + members[j].name + ";");
    }
    _indent--;
    println("};");
}

private void printEnum(org.omg.CORBA.EnumDef def) throws
    org.omg.CORBA.UserException {
    org.omg.CORBA.TypeCode type = def.type();
    println("enum " + type.name() + " {");
    _indent++;
    int count = type.member_count();
    for(int j = 0; j < count; j++) {
        println(type.member_name(j) + ((j == count - 1) ? "" : ","));
    }
    _indent--;
    println("};");
}

private void printAlias(org.omg.CORBA.AliasDef def) throws
    org.omg.CORBA.UserException {
    org.omg.CORBA.IDLType idlType = def.original_type_def();
    String arrayBounds = "";
    while(true) {
        // This is a little strange, since the syntax of typedef'ed
        // arrays is stupid.
        org.omg.CORBA.ArrayDef arrayDef =
```

```

        org.omg.CORBA.ArrayDefHelper.narrow(idlType);
        if(arrayDef == null) {
            break;
        }
        arrayBounds += "[" + arrayDef.length() + "];";
        idlType = arrayDef.element_type_def();
    }
    println("typedef " + toIdl(idlType) + " " + def.name() + arrayBounds + ");");
}

private void printAttribute(org.omg.CORBA.AttributeDef def) throws
    org.omg.CORBA.UserException {
    String readonly = def.mode() == org.omg.CORBA.AttributeMode.ATTR_READONLY ?
        "readonly " : "";
    println(readonly + "attribute " + toIdl(def.type_def()) + " " + def.name() +
        ");");
}

private void printOperation(org.omg.CORBA.OperationDef def) throws
    org.omg.CORBA.UserException {
    String oneway = def.mode() == org.omg.CORBA.OperationMode.OP_ONEWAY ?
        "oneway " : "";
    println(oneway + toIdl(def.result_def()) + " " + def.name() + "(");
    _indent++;
    org.omg.CORBA.ParameterDescription[] parameters = def.params();
    for(int k = 0; k < parameters.length; k++) {
        String[] mode = { "in", "out", "inout" };
        String comma = k == parameters.length - 1 ? "" : ",";
        println(mode[parameters[k].mode.value()] + " " +
            toIdl(parameters[k].type_def) + " " +
            parameters[k].name + comma);
    }
    _indent--;
    org.omg.CORBA.ExceptionDef[] exceptions = def.exceptions();
    if(exceptions.length > 0) {
        println(" raises (");
        _indent++;
        for(int k = 0; k < exceptions.length; k++) {
            String comma = k == exceptions.length - 1 ? "" : ",";
            println(exceptions[k].absolute_name() + comma);
        }
        _indent--;
    }
    println(");");
}
}

```

```
private void printInterface(org.omg.CORBA.InterfaceDef ndef) throws
    org.omg.CORBA.UserException {
    String superList = "";
    {
        org.omg.CORBA.InterfaceDef[] base_interfaces = ndef.base_interfaces();
        if(base_interfaces.length > 0) {
            superList += " :";
            for(int j = 0; j < base_interfaces.length; j++) {
                String comma = j == base_interfaces.length - 1 ? "" : ",";
                superList += " " + base_interfaces[j].absolute_name() + comma;
            }
        }
    }
    println("interface " + ndef.name() + superList + " {");
    _indent++;
    print(ndef);
    _indent--;
    println("};");
}

private void printModule(org.omg.CORBA.ModuleDef def) throws
    org.omg.CORBA.UserException {
    println("module " + def.name() + " {");
    _indent++;
    print(def);
    _indent--;
    println("};");
}
}
```

Callback Example

The callback example is available online at demo/examples/corba/basic/callback.

README

Overview
=====

callback shows a CORBA server object that calls back to the client-side object. It works by activating a new object in the client-side ORB, using the Basic Object Adapter (BOA), and `boa.obj_is_ready()`, and sending a reference to

that object to the CORBA server object.

Source files
=====

client.idl

The CORBA IDL that defines the client-side object, that will be called from the server.

```
interface Client
    wstring helloBack()
```

server.idl

The CORBA IDL that defines the server-side object, that will be called from the client, and that will in turn call back to the client.

```
interface Server
    wstring hello (in client::Client object)
```

Since the object is registered on the client side, and is not published in the database, to perform a callback the server object must have a reference to the client-side object. In this example, the server is called with a reference to the object that has been registered with the client-side Basic Object Adapter (BOA) as a parameter.

Client.java

Invoke the client program from a command prompt, and pass it four arguments, the

- service URL (service ID, hostname, port, and SID if port is a listener)
- name of the published bean to lookup and instantiate
- username
- password that authenticates the client to the Oracle8i database server

For example:

```
% java -classpath LIBs Client sess_iiop://localhost:2222 \  
    /test/myHello scott tiger
```

where LIBs is the classpath that must include

```
$ORACLE_HOME/lib/aurora_client.jar  
#If using Java 2, use classes12.zip instead of classes111.zip  
$ORACLE_HOME/jdbc/lib/classes111.zip  
$ORACLE_HOME/lib/vbjorb.jar  
$ORACLE_HOME/lib/vbjapp.jar  
$JAVA_HOME/lib/classes.zip
```

The client code performs the following steps:

- gets the arguments passed on the command line
- creates a new JNDI Context (InitialContext())
- looks up the published CORBA 'Server' object to find and activate it
- starts up the ORB on the client system (ORB.init())
- gets the basic object adapter object (BOA)
- instantiates a new client callback object (new ClientImpl()), and registers it with the object adapter (boa.obj_is_ready(client))
- invokes the hello() method on the server object, passing it the reference to the client callback object

It is important to do the lookup() before initializing the ORB on the Client side: The lookup call initializes the ORB in a way that's compatible with Oracle 8i. The following org.omg.CORBA.ORB.init() call does not initialize a new ORB instance but just returns the orb that was initialized by the lookup call.

The client prints:

```
I Called back and got: Hello Client World!
```

which is the concatenation of the strings returned by the server object, and the called-back client-side object.

```
serverServer/ServerImpl.java  
-----
```

This class implements the server interface. The code has one method, hello(), which returns its own String ("I called back and got: ") plus the String that it gets as the return from the callback to the client.

clientServer/ClientImpl.java

This class implements the client interface. It has a public constructor, which is required, and a single method, helloBack(), which simply returns the String "Hello Client World!" to the client that called it (the server object 'server' in this case).

Compiling and Running the Example

=====

UNIX

Enter the command 'make all' or simply 'make' in the shell to compile, load, and deploy the objects, and run the client program. Other targets are 'run' and 'clean'.

Make sure that a shell environment variable ORACLE_HOME is set to point to the home location of the Oracle installation. This is operating system dependent, so see the Installation documentation that came with your system for the location. Also, review the README file for the Oracle database, and the README file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

Windows NT

On Windows NT, run the batch file makeit.bat from a DOS command prompt to compile, load, and deploy the objects. Run the batch file runit.bat to run the client program, and see the results.

Make sure that the environment variables %ORACLE_HOME%, %CLASSPATH%, and %SERVICE% are set appropriately for the DOS command window. You can set these as either user or system environment variables from the Control Panel. Double click on System in the Control Panel then on the Environment tab to set these variables. Start a new DOS window after setting environment variable values.

See the Installation documentation that came with your Oracle8i system for the values of these variables. Also, review the README file for the Oracle database, and the README file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

You can also set an environment variable %JAVA_HOME% to point to the root of your Java JDK. For example, SET JAVA_HOME=C:\JDK1.1.6.

IDL Files

Client.IDL

```
module client {
    interface Client {
        wstring helloBack ();
    };
};
```

Server.IDL

```
#include <client.idl>

module server {
    interface Server {
        wstring hello (in client::Client object);
    };
};
```

Server

ServerImpl.java

```
/* $Header: ServerImpl.java 14-mar-00.13:48:31 ielayyan Exp $ */
/* Copyright (c) Oracle Corporation 2000. All Rights Reserved. */
package serverServer;

import server.*;
import client.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class ServerImpl extends _ServerImplBase implements ActivatableObject
{
    public String hello (Client client) {
        return "I Called back and got: " + client.helloBack ();
    }
}
```

```

    }

    public org.omg.CORBA.Object _initializeAuroraObject () {
        return this;
    }
}

```

Client

The client invokes the server object, which calls back to another object on the client-side. The originating client is implemented in [Client.java](#). The client-side callback object is implemented in [ClientImpl.java](#).

Client.java

```

import server.*;
import client.*;
import clientServer.*;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class Client
{
    public static void main (String[] args) throws Exception {
        if (args.length != 4) {
            System.out.println ("usage: Client serviceURL objectName user password");
            System.exit (1);
        }
        String serviceURL = args [0];
        String objectName = args [1];
        String user = args [2];
        String password = args [3];

        Hashtable env = new Hashtable ();
        env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        env.put (Context.SECURITY_PRINCIPAL, user);
        env.put (Context.SECURITY_CREDENTIALS, password);
        env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
        Context ic = new InitialContext (env);

        // Get the server object before preparing the client object

```

```
// You have to do it in that order to get the ORB initialized correctly
Server server = (Server)ic.lookup (serviceURL + objectName);

// Create the client object and publish it to the orb in the client
//org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init ();
com.visigenic.vbroker.orb.ORB orb = oracle.aurora.jndi.orb_dep.Orb.init();
org.omg.CORBA.BOA boa = orb.BOA_init ();
ClientImpl client = new ClientImpl ();
boa.obj_is_ready (client);

// Pass the client to the server that will call us back
System.out.println (server.hello (client));
}
}
```

ClientImpl.java

```
/* $Header: ClientImpl.java 14-mar-00.13:48:25 ielayyan Exp $ */
/* Copyright (c) Oracle Corporation 2000. All Rights Reserved. */
package clientServer;

import client.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class ClientImpl extends _ClientImplBase implements ActivatableObject
{
    public String helloBack () {
        return "Hello Client World!";
    }

    public org.omg.CORBA.Object _initializeAuroraObject () {
        return this;
    }
}
```

TIE Example

This example demonstrates how to use the TIE mechanism.

README

Overview
=====

This is a CORBA TIE (delegation) implementation of the helloworld example. See the readme for that example for more information. It uses the `_initializeAuroraObject()` method to return a class delegate, rather than the object itself.

Source files
=====

hello.idl

(See the helloworld example readme file.)

Client.java

(See the helloworld example readme file.)

helloServer/HelloImpl.java

Implements the IDL-specified Hello interface. The interface has one method, `helloWorld()`, that returns a String to the caller.

Note that the class definition *implements* the IDL-generated HelloOperations interface, rather than extending `_HelloImplBase`, as in the helloworld example.

The class also implements the Aurora ActivateableObject interface. ActivateableObject has only one method: `_initializeAuroraObject()`, which returns the class to be activated by the BOA.

This class performs no database access.

Client-side output
=====

The client prints the returned String "Hello World!" and then exits immediately.

Compiling and Running the Example

=====

UNIX

Enter the command 'make all' or simply 'make' in the shell to compile, load, and deploy the objects, and run the client program. Other targets are 'run' and 'clean'.

Make sure that a shell environment variable ORACLE_HOME is set to point to the home location of the Oracle installation. This is operating system dependent, so see the Installation documentation that came with your system for the location. Also, review the README file for the Oracle database, and the README file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

Windows NT

On Windows NT, run the batch file makeit.bat from a DOS command prompt to compile, load, and deploy the objects. Run the batch file runit.bat to run the client program, and see the results.

Make sure that the environment variables %ORACLE_HOME%, %CLASSPATH%, and %SERVICE% are set appropriately for the DOS command window. You can set these as either user or system environment variables from the Control Panel. Double click on System in the Control Panel then on the Environment tab to set these variables. Start a new DOS window after setting environment variable values.

See the Installation documentation that came with your Oracle8i system for the values of these variables. Also, review the README file for the Oracle database, and the README file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

You can also set an environment variable %JAVA_HOME% to point to the root of your Java JDK. For example, SET JAVA_HOME=C:\JDK1.1.6.

Hello.IDL

```
module hello {
    interface Hello {
        wstring helloWorld ();
    };
};
```

Server Code - HelloImpl.java

```
package helloServer;

import hello.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class HelloImpl implements HelloOperations, ActivatableObject
{
    public String helloWorld () {
        return "Hello World!";
    }

    public org.omg.CORBA.Object _initializeAuroraObject () {
        return new _tie_Hello (this);
    }
}
```

Client.java

```
import hello.Hello;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class Client
{
    public static void main (String[] args) throws Exception {
        if (args.length != 4) {
            System.out.println ("usage: Client serviceURL objectName user password");
            System.exit (1);
        }
        String serviceURL = args [0];
        String objectName = args [1];
```

```
String user = args [2];
String password = args [3];

Hashtable env = new Hashtable ();
env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put (Context.SECURITY_PRINCIPAL, user);
env.put (Context.SECURITY_CREDENTIALS, password);
env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
Context ic = new InitialContext (env);

Hello hello = (Hello)ic.lookup (serviceURL + objectName);
System.out.println (hello.helloWorld ());
}
}
```

Pure CORBA Client

This example uses CORBA Naming Service to retrieve any objects instead of JNDI.

README

Overview
=====

This example is a variant of the VisiBroker for Java "bank" example, which simply creates and publishes a factory CORBA object AccountManager that generates Account objects with some random balance. The Account object has a method, balance(), that returns the account "balance".

This example differs from the other basic CORBA examples in this set in that it does not use JNDI to lookup and activate the published object. It uses the CosNaming name service instead, along with the classes that are part of the oracle.aurora.AuroraServices package.

Source files
=====

bank.idl

The CORBA IDL for the example. Defines a single interface Bank with two interfaces: Account and AccountManager. The AccountManager has a single method, open(), that creates an Account object, and returns it to the caller. The Account class has a single method, balance(), that returns the random balance in the account.

bankServer.AccountManagerImpl.java

The Java code that implements the AccountManager class.

bankServer.AccountImpl.java

The Java code that implements the Account class.

Client.java

You invoke the client program from a command prompt, and pass it five arguments. For example:

```
% java -classpath LIBs Client localhost 2481 ORCL scott tiger
```

where LIBs is the classpath that must include

```
$ORACLE_HOME/lib/aurora_client.jar
#If using Java 2, use classes12.zip instead of classes111.zip
$ORACLE_HOME/jdbc/lib/classes111.zip
$ORACLE_HOME/lib/vbjorb.jar
$ORACLE_HOME/lib/vbjapp.jar
$JAVA_HOME/lib/classes.zip
```

(Note: for NT users, the environment variables might be %ORACLE_HOME% and %JAVA_HOME%.)

The client program looks up and accesses the published AccountManager using the following steps:

- (1) Get argument values from the invocation that set:
 - (a) the hostname of the server machine
 - (b) the GIOP listener port on that server
 - (c) the database SID for the Oracle server
 - (d) the username in the instance

(e) the password

(2) Uses the standard `resolve_initial_references()` method on the ORB to obtain the `NameService` of 8i. There are various properties that control how this initial name service is obtained, as explained in the doc and in the example code. The Oracle name service object is an instance of a `PublishingContext`, which is an Oracle-specific extension to the `CosNaming NamingContext` interface. But you can resolve arbitrary references to any service also using `resolve_initial_references()`.

(The `PublishingContext` class, along with the other Oracle-specific classes used in this example, is documented in the JavaDoc that accompanies this EJB/CORBA product.)

(3) Gets and uses a server login object, which is published under the standard name `/etc/login` in all Java-enabled Oracle8i databases. This is done in the following steps:

- (a) Set the `/etc` directory and the login object name as members of a `CosNaming NameComponent` array.
- (b) Using this array, resolve the component as a Java Object.
- (c) Narrow it to be a published object type.
- (d) Activate and get the object using the Oracle-specific `activate_no_helper()` method.
- (e) Narrow it to an Oracle `LoginServer` object.
- (f) Create a client login proxy object.
- (g) Authenticate the client using the login object.

(4) Lookup and activate an `AccountManager` class, which is published (by the Makefile) as `/test/bank`. The steps are the same as those for the login object.

(You can appreciate now how much the Oracle `JNDI lookup()` method is doing for you, as it performs steps (2), (3), and (4) in one invocation.)

(5) Get a new `Account` object, call the `balance()` method on it to get the "balance", and print the value to the client console.

Output
=====

The printed output is something like:

The balance in Jack.B.Quick's account is \$786.68

The actual balance amount is a random number, and will be different each time you run this program.

Compiling and Running the Example

=====

UNIX

Enter the command 'make all' or simply 'make' in the shell to compile, load, and deploy the objects, and run the client program. Other targets are 'run' and 'clean'.

Make sure that a shell environment variable ORACLE_HOME is set to point to the home location of the Oracle installation. This is operating system dependent, so see the Installation documentation that came with your system for the location. Also, review the README file for the Oracle database, and the README file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

Windows NT

On Windows NT, run the batch file makeit.bat from a DOS command prompt to compile, load, and deploy the objects. Run the batch file runit.bat to run the client program, and see the results.

Make sure that the environment variables %ORACLE_HOME%, %CLASSPATH%, and %SERVICE% are set appropriately for the DOS command window. You can set these as either user or system environment variables from the Control Panel. Double click on System in the Control Panel then on the Environment tab to set these variables. Start a new DOS window after setting environment variable values.

See the Installation documentation that came with your Oracle8i system for the values of these variables. Also, review the README file for the Oracle database, and the README file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

You can also set an environment variable `%JAVA_HOME%` to point to the root of your Java JDK. For example, `SET JAVA_HOME=C:\JDK1.1.6`.

Bank.IDL

```
// Bank.idl

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

Server Code

AccountManagerImpl.java

```
package bankServer;

// import the idl-generated classes
import Bank.*;

import java.util.Dictionary;
import java.util.Random;
import java.util.Hashtable;

// Corba specific imports
import org.omg.CORBA.Object;

// Aurora-orb specific imports
import oracle.aurora.AuroraServices.ActivatableObject;

public class AccountManagerImpl
    extends _AccountManagerImplBase
    implements ActivatableObject
{
    private Dictionary _accounts = new Hashtable ();
    private Random _random = new Random ();

    // Constructors
    public AccountManagerImpl () { super (); }
    public AccountManagerImpl (String name) { super (name); }

    public Object _initializeAuroraObject () {
```

```
        return new AccountManagerImpl ("BankManager");
    }

    public synchronized Account open (String name) {
        // Lookup the account in the account dictionary.
        Account account = (Account) _accounts.get (name);

        // If there was no account in the dictionary, create one.
        if (account == null) {
            // Make up the account's balance, between 0 and 1000 dollars.
            float balance = Math.abs (_random.nextInt ()) % 100000 / 100f;

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

            // Make the object available to the ORB.
            _orb ().connect (account);

            // Print out the new account.
            System.out.println ("Created " + name + "'s account: " + account);

            // Save the account in the account dictionary.
            _accounts.put (name, account);
        }

        // Return the account.
        return account;
    }
}
```

AccountImpl.java

```
package bankServer;

import Bank.*;

public class AccountImpl extends _AccountImplBase {
    private float _balance;

    public AccountImpl () { _balance = (float) 100000.00; }
    public AccountImpl (float balance) { _balance = balance; }
    public float balance () { return _balance; }
}
```

Client.java

```
import java.lang.Exception;

import org.omg.CORBA.Object;
import org.omg.CORBA.SystemException;
import org.omg.CosNaming.NameComponent;

import oracle.aurora.client.Login;
import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.AuroraServices.LoginServerHelper;
import oracle.aurora.AuroraServices.PublishedObject;
import oracle.aurora.AuroraServices.PublishingContext;
import oracle.aurora.AuroraServices.PublishedObjectHelper;
import oracle.aurora.AuroraServices.PublishingContextHelper;

import Bank.Account;
import Bank.AccountManager;
import Bank.AccountManagerHelper;

public class Client {
    public static void main(String args[]) throws Exception {
        // Parse the args
        if (args.length < 4 || args.length > 5 ) {
            System.out.println ("usage: Client host port username password <sid>");
            System.exit(1);
        }
        String host = args[0];
        String port = args[1];
        String username = args[2];
        String password = args[3];
        String sid = null;
        if(args.length == 5)
            sid = args[4];

        // Declarations for an account and manager
        Account account = null;
        AccountManager manager = null;
        com.visigenic.vbroker.orb.ORB orb;
        PublishingContext rootCtx = null;

        // access the Aurora Names Service
        try {
            // Initialize the ORB
            String initref;
```

```

        initref = (sid == null) ? "iioploc://" + host + ":" + port :
"iioploc://" + host + ":" + port + ":" + sid;
        System.setProperty("ORBDefaultInitRef", initref);

        /*
        * Alternatively the following individual properties can be set
        * which take precedence over the URL above
        System.setProperty("ORBBootHost", host);
        System.setProperty("ORBBootPort", port);
        if(sid != null)
System.setProperty("ORACLE_SID", sid);
        */

        /*
        * Some of the other properties that you can set
        System.setProperty("ORNameServiceBackCompat", "false");
        System.setProperty("USE_SERVICE_NAME", "true");
        System.setProperty("ORBUseSSL", "true");
        System.setProperty("TRANSPORT_TYPE", "sess_iiop");
        */

orb = oracle.aurora.jndi.orb_dep.Orb.init();
// Get the Name service Object reference
rootCtx = PublishingContextHelper.narrow(orb.resolve_initial_references(
"NameService"));
// Get the pre-published login object reference
PublishedObject loginPubObj = null;
LoginServer serv = null;
NameComponent[] nameComponent = new NameComponent[2];
nameComponent[0] = new NameComponent ("etc", "");
nameComponent[1] = new NameComponent ("login", "");

// Lookup this object in the Name service
Object loginCorbaObj = rootCtx.resolve (nameComponent);

// Make sure it is a published object
loginPubObj = PublishedObjectHelper.narrow (loginCorbaObj);

// create and activate this object (non-standard call)
loginCorbaObj = loginPubObj.activate_no_helper ();
serv = LoginServerHelper.narrow (loginCorbaObj);

// Create a client login proxy object and authenticate to the DB
Login login = new Login (serv);
login.authenticate (username, password, null);

```

```
// Now create and get the bank object reference
PublishedObject bankPubObj = null;
nameComponent[0] = new NameComponent ("test", "");
nameComponent[1] = new NameComponent ("bank", "");

// Lookup this object in the name service
Object bankCorbaObj = rootCtx.resolve (nameComponent);

// Make sure it is a published object
bankPubObj = PublishedObjectHelper.narrow (bankCorbaObj);

// create and activate this object (non-standard call)
bankCorbaObj = bankPubObj.activate_no_helper ();
manager = AccountManagerHelper.narrow (bankCorbaObj);

account = manager.open ("Jack.B.Quick");

float balance = account.balance ();
System.out.println ("The balance in Jack.B.Quick's account is $"
                    + balance);
} catch (SystemException e) {
    System.out.println ("Caught System Exception: " + e);
    e.printStackTrace ();
} catch (Exception e) {
    System.out.println ("Caught Unknown Exception: " + e);
    e.printStackTrace ();
}
}
}
```

JTA Examples

Single-Phase Commit JTA Transaction Example

Employee.IDL

```
module employee {
    struct EmployeeInfo {
        wstring name;
        long number;
        double salary;
    };
};
```



```

};

exception SQLException {
    wstring message;
};

interface Employee {
    void setUpDSConnection (in wstring dsName) raises (SQLException);
    EmployeeInfo getEmployee (in wstring name) raises (SQLException);
    void updateEmployee (in EmployeeInfo name) raises (SQLException);
};
};

```

Client.java

```

import employee.*;

import java.sql.DriverManager;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.transaction.UserTransaction;

import java.sql.SQLException;
import javax.naming.NamingException;

import oracle.aurora.jndi.jdbc_access.jdbc_accessURLContextFactory;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

public class Client
{
    public static void main (String[] args) throws Exception
    {
        if (args.length != 7)
        {
            System.out.println ("usage: Client sessiiopURL jdbcURL objectName " +
                "user password userTxnName dataSrcName");
            System.exit (1);
        }
        String sessiiopURL = args [0];
        String jdbcURL = args [1];
        String objectName = args [2];
        String user = args [3];
        String password = args [4];
        String utName = args [5];
    }
}

```

```
String dsName = args [6];

// lookup usertransaction object in the namespace
UserTransaction ut = lookupUserTransaction (user, password,
                                             jdbcURL, utName);

// lookup employee object in the namespace
Employee employee = lookupObject (user, password, sessiopURL, objectName);
EmployeeInfo info;

// for (int ii = 0; ii < 10; ii++)
// {
// start a transaction
ut.begin ();

// set up the DS on the server
employee.setUpDSConnection (dsName);

// retrieve the info
info = employee.getEmployee ("SCOTT");
System.out.println ("Before Update: " + info.name + " " + info.salary);

// change the salary and update it
System.out.println ("Increase by 10%");
info.salary += (info.salary * 10) / 100;
employee.updateEmployee (info);

// commit the changes
ut.commit ();

// NOTE: you can do this before the commit of the previous transaction
// (without starting a txn) then it becomes part of the first
// global transaction.
// start another transaction to retrieve the updated info
ut.begin ();

// Since, you started a new transaction, the DS needs to be
// enlisted with the 'new' transaction. Hence, setup the DS on the server
employee.setUpDSConnection (dsName);

// try to retrieve the updated info
info = employee.getEmployee ("SCOTT");
System.out.println ("After Update: " + info.name + " " + info.salary);

// commit the second transaction
```

```
        ut.commit ();
    }

    private static UserTransaction lookupUserTransaction (String user,
        String password,
        String jdbcURL,
        String utName)
    {
        UserTransaction ut = null;
        try {
            Hashtable env = new Hashtable ();
            env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
            env.put (Context.SECURITY_PRINCIPAL, user);
            env.put (Context.SECURITY_CREDENTIALS, password);
            env.put (jdbc_accessURLContextFactory.CONNECTION_URL_PROP, jdbcURL);
            Context ic = new InitialContext (env);

            DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver ());

            ut = (UserTransaction)ic.lookup ("jdbc_access://" + utName);
        } catch (NamingException e) {
            e.printStackTrace ();
        } catch (SQLException e) {
            e.printStackTrace ();
        }
        return ut;
    }

    private static Employee lookupObject (String user, String password,
        String sessiopURL, String objectName)
    {
        Employee emp = null;
        try {
            Hashtable env = new Hashtable ();
            env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
            env.put (Context.SECURITY_PRINCIPAL, user);
            env.put (Context.SECURITY_CREDENTIALS, password);
            env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
            Context ic = new InitialContext (env);

            System.out.println ("Trying to lookup: " + sessiopURL + objectName);
            emp = (Employee)ic.lookup (sessiopURL + objectName);
        } catch (NamingException e) {
            e.printStackTrace ();
        }
    }
}
```

```
        return emp;
    }
}
```

EmployeeServer.sqlj

```
package employeeServer;

import employee.*;

import java.sql.Connection;
import java.sql.DataSource;
import java.sql.SQLException;
import java.util.Hashtable;

import javax.naming.Context;
import javax.naming.InitialContext;

import javax.naming.NamingException;

public class EmployeeImpl
    extends _EmployeeImplBase
{
    Context ic = null;
    DataSource ds = null;
    Connection conn = null;

    private void setInSessionLookupContext ()
        throws NamingException
    {
        Hashtable env = new Hashtable ();
        env.put (Context.INITIAL_CONTEXT_FACTORY,
            "oracle.aurora.namespace.InitialContextFactoryImpl");
        ic = new InitialContext (env);
        // ic = new InitialContext ();
    }

    public void setUpDSCConnection (String dsName)
        throws SQLException
    {
        try {
            if (ic == null)
                setInSessionLookupContext ();

            // get a connection to the local DB
```

```
        ds = (DataSource)ic.lookup (dsName);

        // get a connectoin to the local DB
        conn = ds.getConnection ();
    } catch (NamingException e) {
        e.printStackTrace ();
        throw new SQLException ("setUpDSConnection failed:" + e.toString ());
    } catch (SQLException e) {
        e.printStackTrace ();
        throw new SQLException ("setUpDSConnection failed:" + e.toString ());
    }
}

public EmployeeInfo getEmployee (String name)
    throws SQLException
{
    try {
        if (conn == null)
            throw new SQLException ("getEmployee: conn is null");

        int empno = 0;
        double salary = 0.0;
        #sql { select empno, sal into :empno, :salary from emp
                where ename = :name };
        return new EmployeeInfo (name, empno, (float)salary);
    } catch (SQLException e) {
        throw new SQLException (e.getMessage ());
    }
}

public void updateEmployee (EmployeeInfo employee)
    throws SQLException
{
    if (conn == null)
        throw new SQLException ("updateEmployee: conn is null");

    try {
        #sql { update emp set ename = :(employee.name), sal = :(employee.salary)
                where empno = :(employee.number) };
    } catch (SQLException e) {
        throw new SQLException (e.getMessage ());
    }
}
}
```

Two-Phase Commit JTA Transaction Example

Employee.IDL

```
module employee {
    struct EmployeeInfo {
        wstring name;
        long number;
        double salary;
    };

    exception SQLException {
        wstring message;
    };

    interface Employee {
        void initialize (in wstring user, in wstring password,
            in wstring serviceURL, in wstring objectName,
            in wstring utName, in wstring localDSName,
            in wstring remoteDSName) raises (SQLException);

        EmployeeInfo getEmployee (in wstring empName) raises (SQLException);
        void updateEmployee (in EmployeeInfo empInfo) raises (SQLException);

        EmployeeInfo getRemoteEmpInfo (in wstring name) raises (SQLException);
        void updateRemoteEmployee (in EmployeeInfo empInfo) raises (SQLException);
    };
};
```

Client.java

```
import employee.*;

import java.sql.DriverManager;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.transaction.UserTransaction;

import java.sql.SQLException;
import javax.naming.NamingException;

import oracle.aurora.jndi.jdbc_access.jdbc_accessURLContextFactory;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
```

```
public class Client
{
    public static void main (String[] args) throws Exception
    {
        if (args.length != 7)
        {
            System.out.println ("usage: Client sessiopURL objectName user password"
                + " userTxnName localDataSrcName remoteDataSrcName");
            System.exit (1);
        }
        String sessiopURL = args [0];
        String objectName = args [1];
        String user = args [2];
        String password = args [3];
        String utName = args [4];
        String localDSName = args [5];
        String remoteDSName = args [6];

        Hashtable env = new Hashtable ();
        env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        env.put (Context.SECURITY_PRINCIPAL, user);
        env.put (Context.SECURITY_CREDENTIALS, password);
        env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
        Context ic = new InitialContext (env);

        Employee employee = null;
        EmployeeInfo info;

        try {
            employee = (Employee)ic.lookup (sessiopURL + objectName);
            employee.initialize (user, password, sessiopURL, objectName, utName,
                localDSName, remoteDSName);

            info = employee.getEmployee ("SCOTT");
            System.out.println (info.name + " " + " " + info.salary);
            System.out.print ("Increase by 10% to ");
            info.salary += (info.salary * 10) / 100;
            System.out.println (info.salary);
            employee.updateEmployee (info);
        } catch (SQLException e) {
            System.out.println (" Got SQLException: " + e.toString ());
        }
    }
}
```

Server

```
package employeeServer;

import employee.*;

import java.sql.Connection;
import java.sql.DataSource;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.Hashtable;
import java.sql.SQLException;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.transaction.UserTransaction;
import javax.naming.NamingException;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.transaction.xa.OracleJTADDataSource;

public class EmployeeImpl extends _EmployeeImplBase
{
    Context inSessionLookupctx = null;
    UserTransaction ut = null;
    DataSource localDS = null;
    DataSource remoteDS = null;
    Connection localConn = null;
    Connection remoteConn = null;
    String utName = null;
    String localDSName = null;
    String remoteDSName = null;
    String user = null;
    String pwd = null;
    String serviceURL = null;
    String objectName = null;
    EmployeeInfo localEmpInfo = null;

    private void setInSessionLookupContext ()
        throws NamingException
    {
        // NOTE: here we need to set env as 2-phase coord needs
        //       user/pwd to be set (branches is optional)
        Hashtable env = new Hashtable ();
        env.put (Context.SECURITY_PRINCIPAL, user);
    }
}
```



```
env.put (Context.SECURITY_CREDENTIALS, pwd);
inSessionLookupctx = new InitialContext (env);
}

public void initialize (String user, String password, String serviceURL,
    String objectName, String utName, String localDSName,
    String remotedDSName)
    throws SQLException
{
    try {
        // set the local variables
        this.user = user;
        this.pwd = password;
        this.objectName = objectName;
        this.utName = utName;
        this.localDSName = localDSName;
        this.remoteDSName = remotedDSName;
        this.serviceURL = serviceURL;

        // se up a ctx to lookup the local/in-session objects
        if (inSessionLookupctx == null)
            setInSessionLookupContext ();

        // lookup the usertransaction
        ut = (UserTransaction)inSessionLookupctx.lookup (utName);

        // get a connection to the local DB
        localDS = (OracleJTADataSource)inSessionLookupctx.lookup (localDSName);

        // get a connection to the local DB
        remoteDS = (OracleJTADataSource)inSessionLookupctx.lookup (remotedDSName);
    } catch (NamingException e) {
        e.printStackTrace ();
        throw new SQLException ("setUpDSCConnection failed:" + e.toString ());
    }
}

private void getConnections ()
    throws SQLException
{
    if (localDS == null)
        throw new SQLException ("local DataSource is NOT set correctly");
    if (remoteDS == null)
        throw new SQLException ("remote DataSource is NOT set correctly");
    if (user == null || pwd == null)
```

```
        throw new SQLException ("user/pwd is NOT set correctly");

        localDS.setURL ("jdbc:oracle:kprb:");
        localConn = localDS.getConnection ();
        System.out.println ("remoteDS.getURL: " + remoteDS.getURL ());
        remoteConn = remoteDS.getConnection (user, pwd);
    }

    private void startTrans ()
        throws SQLException
    {
        try {
            if (ut == null)
                throw new SQLException ("startTrans: userTransaction is null");

            ut.begin ();
        } catch (Exception e) {
            throw new SQLException ("startTrans failed:" + e.toString ());
        }
    }

    private void commitTrans ()
        throws SQLException
    {
        try {
            ut.commit ();
        } catch (Exception e) {
            throw new SQLException ("commitTrans failed:" + e.toString ());
        }
    }

    public EmployeeInfo getLocalEmpInfo (String name)
        throws SQLException
    {
        try {
            if (localConn == null)
                throw new SQLException ("getLocalEmpInfo: localConn is null");
            int empno = 0;
            double salary = 0.0;
            #sql { select empno, sal into :empno, :salary from emp
                    where ename = :name };
            System.out.println (" Local (" + name + ", " + salary + ")");
            return new EmployeeInfo (name, empno, (float)salary);
        } catch (SQLException e) {
            e.printStackTrace ();
        }
    }
}
```

```
        throw new SQLException ("getRemoteEmpInfo SQLException: " + e.toString ());
    }
}

public EmployeeInfo getRemoteEmpInfo (String name)
    throws SQLException
{
    try {
        if (remoteConn== null)
            throw new SQLException ("getRemoteEmpInfo: remoteConn is null");
        int empno = 0;
        double salary = 0.0;

        PreparedStatement ps = remoteConn.prepareStatement
            ("select empno, sal from emp where ename = ?");

        ps.setString (1, name);
        ResultSet rs = ps.executeQuery ();

        while (rs.next ())
        {
            empno = rs.getInt (1);
            salary = rs.getDouble (2);
        }
        System.out.println (" Remote (" + name + ", " + salary + ")");
        return new EmployeeInfo (name, empno, (float)salary);
    } catch (SQLException e) {
        e.printStackTrace ();
        throw new SQLException ("getRemoteEmpInfo SQLException: " + e.toString ());
    }
}

public EmployeeInfo getEmployee (String name)
    throws SQLException
{
    System.out.println ("getEmployee: begin");

    try {
        this.startTrans ();
        // get a connection to the local and remote DB
        this.getConnections ();
    } catch (SQLException e) {
        e.printStackTrace ();
        throw new SQLException ("getEmployee SQLException: " + e.toString ());
    }
}
```

```
// get info for localEmployee = smith
localEmpInfo = this.getLocalEmpInfo ("SMITH");

// get info for the remote employee
EmployeeInfo info = this.getRemoteEmpInfo (name);
System.out.println (" Remote (" + info.name + ", " + info.salary + ")");

this.commitTrans ();
System.out.println ("getEmployee: end");
return info;
}

public void updateEmployee (EmployeeInfo empInfo) throws SQLException
{
    System.out.println ("updateEmployee: begin");
    this.startTrans ();

    try {
        this.getConnections ();

        System.out.println (" Before updating: ");
        this.getLocalEmpInfo ("SMITH");
        this.getRemoteEmpInfo ("SCOTT");

        localEmpInfo.salary -= 0.1 * localEmpInfo.salary;
        this.updateLocalEmployee (localEmpInfo);

        System.out.println (" calling to update " + empInfo.name + " salary");
        updateRemoteEmployee (empInfo);

        System.out.println ("updateEmployee: After updating: ");
        this.getLocalEmpInfo ("SMITH");
        this.getRemoteEmpInfo ("SCOTT");
    } catch (SQLException e) {
        e.printStackTrace ();
        throw new SQLException ("updateEmployee SQLException: " + e.toString ());
    }

    this.commitTrans ();
    System.out.println ("updateEmployee: end");
}

public void updateLocalEmployee (EmployeeInfo empInfo)
    throws SQLException
```

```
{
    System.out.println ("updateLocalEmployee: begin");
    try {
        this.getLocalEmpInfo ("SMITH");

        #sql { update emp set ename = :(empInfo.name), sal = :(empInfo.salary)
              where empno = :(empInfo.number) };

        System.out.println (" after updating " + empInfo.name + " salary");
        this.getLocalEmpInfo (empInfo.name);
    } catch (SQLException e) {
        e.printStackTrace ();
        throw new SQLException (e.toString ());
    }

    System.out.println ("updateLocalEmployee: end");
}

public void updateRemoteEmployee (EmployeeInfo empInfo)
    throws SQLException
{
    System.out.println ("updateRemoteEmployee: begin");

    try {
        PreparedStatement ps = remoteConn.prepareStatement
            ("update emp set ename = ?, sal = ? where empno = ?");
        ps.setString (1, empInfo.name);
        ps.setDouble (2, empInfo.salary);
        ps.setInt (3, empInfo.number);
        ps.executeUpdate ();

        System.out.println (" after updating " + empInfo.name + " salary");
        this.getRemoteEmpInfo (empInfo.name);
    } catch (SQLException e) {
        e.printStackTrace ();
        throw new SQLException (e.toString ());
    }

    System.out.println ("updateRemoteEmployee: end");
}
}
```

JTS Transaction Example

README

Overview

=====

The serversideJTS example shows how to do transaction management for CORBA server objects from the server object, using the XA JTS methods.

Compare this example with the clientside example, in which all transaction management is done on the client.

This example also shows a server object that uses SQLJ in its methods.

Source files

=====

employee.idl

The CORBA IDL for the example. Defines:

An EmployeeInfo struct

A SQLException exception

An Employee interface, with

EmployeeInfo getEmployee(in wstring name)

EmployeeInfo getEmployeeForUpdate(in wstring name)

void updateEmployee(in EmployeeInfo name)

The SQLException exception is used so that SQLException messages can be passed back to the client.

Client.java

You invoke the client program from a command prompt, and pass it four arguments, the

- service URL (service ID, hostname, port, and SID if port is a listener)
- name of the published server object to lookup and instantiate
- username
- password that authenticates the client to the Oracle8i database server

For example:

```
% java -classpath LIBs Client sess_iiop://localhost:2481:ORCL \  
    /test/myEmployee scott tiger
```

where LIBs is the classpath that must include

```
$ORACLE_HOME/lib/aurora_client.jar  
#If using Java 2, use classes12.zip instead of classes111.zip  
$ORACLE_HOME/jdbc/lib/classes111.zip  
$ORACLE_HOME/lib/vbjorb.jar  
$ORACLE_HOME/lib/vbjapp.jar  
$JAVA_HOME/lib/classes.zip
```

The client code is almost exactly the same as the code in `../clientside/Client.java`, but without the JTS transaction calls.

The client code performs the following steps:

- gets the arguments passed on the command line
- creates a new JNDI Context (`InitialContext()`)
- initializes the Aurora transaction service
- looks up the `myEmployee` CORBA published object on the server
(this step also authenticates the client using `NON_SSL_LOGIN` and activates the server object)
- gets and prints information about the employee `SCOTT`
- decreases `SCOTT`'s salary by 10%
- updates the `EMP` table with the new salary by calling the `updateEmployee()` method on the `employee` object
- gets and prints the new information

The printed output is:

```
Beginning salary = 3000.0  
Decrease by 10%  
Final Salary = 2700.0
```

Note that the starting value is taken from the `EMP` table when the example starts to run, so you may see a different salary amount. The new salary amount is written back to the database, and will be used as the

new starting amount if you run this example again.

employeeServer/EmployeeImpl.sqlj

Implements the Employee interface. This file implements the three methods specified in the IDL: `getEmployee()`, `getEmployeeForUpdate()`, and `updateEmployee()`, using SQLJ for ease of DML coding.

EmployeeImpl also adds two private methods, `commitTrans()` and `startTrans()`, that perform XA JTS transaction management from the server.

Note that on the server there is no need to call `AuroraTransactionService.initialize()` to initialize the transaction manager. This is done automatically by the server ORB.

If the SQLJ code throws a `SQLException`, it is caught, and a CORBA-defined `SQLException` is thrown. This in turn would be propagated back to the client, where it is handled.

Compiling and Running the Example

=====

UNIX

Enter the command 'make all' or simply 'make' in the shell to compile, load, and deploy the objects, and run the client program. Other targets are 'run' and 'clean'.

Make sure that a shell environment variable `ORACLE_HOME` is set to point to the home location of the Oracle installation. This is operating system dependent, so see the Installation documentation that came with your system for the location. Also, review the `README` file for the Oracle database, and the `README` file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

Windows NT

On Windows NT, run the batch file `makeit.bat` from a DOS command prompt to compile, load, and deploy the objects. Run the batch file `runit.bat` to run the client program, and see the results.

Make sure that the environment variables `%ORACLE_HOME%`, `%CLASSPATH%`, and `%SERVICE%` are set appropriately for the DOS command window. You can set these as either user or system environment variables from the Control Panel. Double click on System in the Control Panel then on the Environment tab to set these variables. Start a new DOS window after setting environment variable values.

See the Installation documentation that came with your Oracle8i system for the values of these variables. Also, review the README file for the Oracle database, and the README file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

You can also set an environment variable `%JAVA_HOME%` to point to the root of your Java JDK. For example, `SET JAVA_HOME=C:\JDK1.1.6`.

Employee.IDL

```
module employee {
    struct EmployeeInfo {
        wstring name;
        long number;
        double salary;
    };

    exception SQLException {
        wstring message;
    };

    interface Employee {
        EmployeeInfo getEmployee (in wstring name) raises (SQLException);
        EmployeeInfo getEmployeeForUpdate (in wstring name) raises (SQLException);
        void updateEmployee (in EmployeeInfo name) raises (SQLException);
    };
};
```

Client.java

```
import employee.*;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class Client
{
    public static void main (String[] args) throws Exception {
        if (args.length != 4) {
            System.out.println ("usage: Client serviceURL objectName user password");
            System.exit (1);
        }
        String serviceURL = args [0];
        String objectName = args [1];
        String user = args [2];
        String password = args [3];

        // get the handle to the InitialContext
        Hashtable env = new Hashtable ();
        env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        env.put (Context.SECURITY_PRINCIPAL, user);
        env.put (Context.SECURITY_CREDENTIALS, password);
        env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
        Context ic = new InitialContext (env);

        // This is using Server-side TX services, specifically, JTS/XA TX:

        // get handle to the object and it's info
        Employee employee = (Employee)ic.lookup (serviceURL + objectName);

        // get the info about a specific employee
        EmployeeInfo info = employee.getEmployee ("SCOTT");
        System.out.println ("Beginning salary = " + info.salary);
        System.out.println ("Decrease by 10%");
        // do work on the object or it's info
        info.salary -= (info.salary * 10) / 100;

        // call update on the server-side
        employee.updateEmployee (info);
    }
}
```

```

        System.out.println ("Final Salary = " + info.salary);
    }
}

```

Server

```

package employeeServer;

import employee.*;
import java.sql.*;

import oracle.aurora.jts.util.*;
import org.omg.CosTransactions.*;

public class EmployeeImpl extends _EmployeeImplBase
{
    Control c;

    private void startTrans () throws SQLException {
        try {
            TS.getTS ().getCurrent ().begin ();
        } catch (Exception e) {
            throw new SQLException ("begin failed:" + e);
        }
    }

    private void commitTrans () throws SQLException {
        try {
            TS.getTS ().getCurrent ().commit (true);
        } catch (Exception e) {
            throw new SQLException ("commit failed:" + e);
        }
    }

    public EmployeeInfo getEmployee (String name) throws SQLException {
        try {
            startTrans ();

            int empno = 0;
            double salary = 0.0;
            #sql { select empno, sal into :empno, :salary from emp
                where ename = :name };
            c = TS.getTS().getCurrent().suspend();
            return new EmployeeInfo (name, empno, (float)salary);
        } catch (SQLException e) {

```

```
        throw new SQLException (e.getMessage ());
    } catch (Exception e) {
        throw new SQLException (e.getMessage());
    }
}

public EmployeeInfo getEmployeeForUpdate (String name) throws SQLException {
    try {
        startTrans ();

        int empno = 0;
        double salary = 0.0;
        #sql { select empno, sal into :empno, :salary from emp
                where ename = :name for update };
        return new EmployeeInfo (name, empno, (float)salary);
    } catch (SQLException e) {
        throw new SQLException (e.getMessage ());
    }
}

public void updateEmployee (EmployeeInfo employee) throws SQLException {
    try {
        TS.getTS().getCurrent().resume(c);

        #sql { update emp set ename = :(employee.name), sal = :(employee.salary)
                where empno = :(employee.number) };
        commitTrans ();
    } catch (SQLException e) {
        throw new SQLException (e.getMessage ());
    } catch (Exception e) {
        throw new SQLException (e.getMessage ());
    }
}
}
```

SSL Examples

Client-Side Authentication

README

Overview

=====

This is a very simple CORBA example using client side ssl for login. The helloWorld server object merely returns a greeting plus the Java VM version number to the client.

The purpose of the example is to show how to use ssl client side authentication for logins.

Setup required

You need to open the encrypted wallet(ewallet.der) provided in this directory using the wallet manager tool provided by Oracle. The password is welcome12. Copy the cleartext into TNS_ADMIN directory and restart the database and listeners.

The encrypted wallet(ewallet.der) is only valid for Solaris platforms. For other platforms, you should generate the wallet using Oracle's owm tool.

This test also requires B64 encoded wallet(cert.txt) which is already present in this directory. You can also generate your own using Oracle's owmgui tool and using export option in the tool.

The parameter SSL_CLIENT_AUTHENTICATION in \$TNSADMIN/sqlnet.ora should be set to TRUE before restarting the database and listeners

The setup also requires creation of global user guest. The script to create global user is in this directory(create.sh). This script prompts for username and password of a privileged user as input to this script.

The Makefile has loadjava that loads all the classes into scott's schema whereas the client program is executed as the user guest. Hence loadjava has a "-grant guest" to grant the privileges to guest.

Source files

=====

hello.idl

The CORBA IDL for the example. Defines a single interface Hello with a single method helloWorld(). The interface is defined in the Module named 'hello', which determines the name of the directory in which the idl2java compiler

places the generated files.

The `helloWorld()` method returns a CORBA `wstring`, which maps to a Java `String` type:

```
module hello
  interface Hello
    wstring helloWorld()
```

Client.java

You invoke the client program from a command prompt, and pass it three arguments, the

- service URL (service ID, hostname, port, and SID if port is a listener)
- name of the published bean to lookup and instantiate
- credentials file - the B64 encoded wallet for the user. This is a generated wallet at Oracle site.

The password for the wallet is hardcoded as "welcome12"

For example:

```
% java -classpath LIBS Client sess_iiop://localhost:2222 /test/myHello cert.txt
```

where LIBS is the classpath that must include

```
$ORACLE_HOME/lib/aurora_client.jar
#If using Java 2, use classes12.zip instead of classes111.zip
$ORACLE_HOME/jdbc/lib/classes111.zip
$ORACLE_HOME/lib/vbjorb.jar
$ORACLE_HOME/lib/vbjapp.jar
$JAVA_HOME/lib/classes.zip
```

(Note: for NT users, the environment variables would be `%ORACLE_HOME%` and `%JAVA_HOME%.`)

The client code performs the following steps:

- gets the arguments passed on the command line
- puts the authentication type and values into env context
- creates a new JNDI Context (`InitialContext()`)
- looks up the published CORBA server object to find and activate it
- invokes the `helloWorld()` method on the hello object and prints the results

The printed output is:

Hello client, your javavm version is 8.1.5.

helloServer/HelloImpl.java

Implements the IDL-specified Hello interface. The interface has one method, helloWorld(), that returns a String to the caller.

helloWorld() invokes System.getProperty("oracle.server.version") to get the version number of the Java VM.

This object performs no database access.

Compiling and Running the Example

=====

UNIX

Enter the command 'make all' or simply 'make' in the shell to compile, load, and deploy the objects, and run the client program. Other targets are 'run' and 'clean'.

Make sure that a shell environment variable ORACLE_HOME is set to point to the home location of the Oracle installation. This is operating system dependent, so see the Installation documentation that came with your system for the location. Also, review the README file for the Oracle database, and the README file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

Windows NT

On Windows NT, run the batch file makeit.bat from a DOS command prompt to compile, load, and deploy the objects. Run the batch file runit.bat to run the client program, and see the results.

Make sure that the environment variables %ORACLE_HOME%, %CLASSPATH%, and %SERVICE% are set appropriately for the DOS command window. You can set these as either user or system environment variables from the Control Panel. Double click on System in the Control Panel then on the Environment tab to set these variables. Start a new DOS window after setting environment variable values.

See the Installation documentation that came with your Oracle8i system for the values of these variables. Also, review the README file for the Oracle database, and the README file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

You can also set an environment variable %JAVA_HOME% to point to the root of your Java JDK. For example, SET JAVA_HOME=C:\JDK1.1.6.

Hello.IDL

```
module hello {
    interface Hello {
        wstring helloWorld ();
    };
};
```

Client.java

```
import hello.Hello;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class Client
{

    public static void main (String[] args) throws Exception {
        if (args.length != 3) {
            System.out.println("usage: Client serviceURL objectName credsFile");
            System.exit(1);
        }
        String serviceURL = args [0];
        String objectName = args [1];
        String credsFile = args [2];
```



```

Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.SSL_CLIENT_AUTH);
env.put(Context.SECURITY_CREDENTIALS, "welcomel2");

// Simply specify a file that contains all the credential info. This is
// the file generated by the wallet manager tool.
env.put(Context.SECURITY_PRINCIPAL, credsFile);

/*
// As an alternative, you may also set the credentials individually, as
// shown below.
env.put(ServiceCtx.SECURITY_USER_CERT, testCert_base64);
env.put(ServiceCtx.SECURITY_CA_CERT, caCert_base64);
env.put(ServiceCtx.SECURITY_ENCRYPTED_PKEY, encryptedPrivateKey_base64);
//System.getProperties().put("AURORA_CLIENT_SSL_DEBUG", "true");
*/

Context ic = new InitialContext(env);

Hello hello = (Hello) ic.lookup(serviceURL + objectName);
System.out.println(hello.helloWorld());
}
}

```

Server

```

package helloServer;

import hello.*;

public class HelloImpl extends _HelloImplBase {
    public String helloWorld() {
        String v = System.getProperty("oracle.server.version");
        return "Hello client, your javavm version is " + v + ".";
    }
}

```

Server-Side Authentication

This example includes setting a trustpoint. If you do not want to involve trustpoints, just remove the section of the code that sets the trustpoint.

README

Overview

=====

This is a very simple CORBA example using server side ssl for login. The helloWorld server object merely returns a greeting plus the Java VM version number to the client.

The purpose of the example is to show how to use ssl server side authentication for logins.

Setup required

You need to open the encrypted wallet(ewallet.der) provided in this directory using the wallet manager tool provided by Oracle. The password is welcome12. Copy the cleartext into TNS_ADMIN directory and restart the database and listeners.

The encrypted wallet(ewallet.der) is only valid for Solaris platforms. For other platforms, you should generate the wallet using Oracle's own tool. And if you generate the wallet, be sure to change the trust point provided in the client file Client.java too.

The parameter SSL_CLIENT_AUTHENTICATION in \$TNSADMIN/sqlnet.ora should be set to false before restarting the database and listeners.

You may also generate a wallet using Oracle tool owngui. The cleartext wallet that you will be using should be binary compatible with the machine you are using to run this sample.

There is also a hard coded trustpoint within Client.java. This trust point matches with the one in the server's wallet. You may replace with your trust point if needed. But this trust point should be matching with the one in the server's wallet.

This example runs with JDK1.1 libraries. If you are using JDK 1.2 libraries, then you should comment out JDK1.1 code and uncomment JDK1.2 code.

Source files

=====

hello.idl

The CORBA IDL for the example. Defines a single interface Hello with a single method helloWorld(). The interface is defined in the Module named 'hello', which determines the name of the directory in which the idl2java compiler places the generated files.

The helloWorld() method returns a CORBA wstring, which maps to a Java String type:

```
module hello
  interface Hello
    wstring helloWorld()
```

Client.java

You invoke the client program from a command prompt, and pass it four arguments, the

- service URL (service ID, hostname, port, and SID if port is a listener)
- name of the published bean to lookup and instantiate
- username
- password

For example:

```
% java -classpath LIBS Client sess_iiop://localhost:2222
/test/sslHelloServerAuthWithTP
```

where LIBS is the classpath that must include

```
$ORACLE_HOME/lib/aurora_client.jar
#If using Java 2, use classes12.zip instead of classes111.zip
$ORACLE_HOME/jdbc/lib/classes111.zip
$ORACLE_HOME/lib/vbjorb.jar
$ORACLE_HOME/lib/vbjapp.jar
$JAVA_HOME/lib/classes.zip
```

(Note: for NT users, the environment variables would be %ORACLE_HOME% and %JAVA_HOME%.)

The client code performs the following steps:

- gets the arguments passed on the command line

- puts the authentication type and values into env context
- creates a new JNDI Context (InitialContext())
- looks up the published CORBA server object to find and activate it
- invokes the helloWorld() method on the hello object and prints the results

The printed output is:

```
Hello client, your javavm version is 8.1.5.
```

```
helloServer/HelloImpl.java
```

```
-----
```

Implements the IDL-specified Hello interface. The interface has one method, helloWorld(), that returns a String to the caller.

helloWorld() invokes System.getProperty("oracle.server.version") to get the version number of the Java VM.

This object performs no database access.

```
Compiling and Running the Example
```

```
=====
```

```
UNIX
```

```
----
```

Enter the command 'make all' or simply 'make' in the shell to compile, load, and deploy the objects, and run the client program. Other targets are 'run' and 'clean'.

Make sure that a shell environment variable ORACLE_HOME is set to point to the home location of the Oracle installation. This is operating system dependent, so see the Installation documentation that came with your system for the location. Also, review the README file for the Oracle database, and the README file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

```
Windows NT
```

```
-----
```

On Windows NT, run the batch file `makeit.bat` from a DOS command prompt to compile, load, and deploy the objects. Run the batch file `runit.bat` to run the client program, and see the results.

Make sure that the environment variables `%ORACLE_HOME%`, `%CLASSPATH%`, and `%SERVICE%` are set appropriately for the DOS command window. You can set these as either user or system environment variables from the Control Panel. Double click on System in the Control Panel then on the Environment tab to set these variables. Start a new DOS window after setting environment variable values.

See the Installation documentation that came with your Oracle8i system for the values of these variables. Also, review the README file for the Oracle database, and the README file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

You can also set an environment variable `%JAVA_HOME%` to point to the root of your Java JDK. For example, `SET JAVA_HOME=C:\JDK1.1.6`.

Hello.IDL

```
module hello {
    interface Hello {
        wstring helloWorld ();
    };
};
```

Client.java

```
import hello.Hello;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;
import oracle.aurora.ssl.*;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jndi.sess_iiop.SessionCtx;
import oracle.aurora.AuroraServices.LoginServer;
//import java.security.cert.*; // Needs JDK 1.2; won't compile in JDK 1.1
import javax.security.cert.*; // for JDK 1.1
import java.io.*;
```

```
public class Client
{
    private static String trustedCert =
"MIIBNjCB4aADAgECAhEAv/poeUAh5DxtXZSkZAIunDANBjkqhkiG9w0BAQQFADAcMQswCQYDVQQG"+
"EwJVUzENMAsgALUEAxQEUK9PVDaeFw05OTExMTcxODQ1NDNaFw0wMjAyMDIxODQ1NDNaMBwxCzAJ"+
"BgnVBAYTAlVIMQ0wCwYDVQQDFARST09UMFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBANEzeul7saeh"+
"q60fGp4Ya0IZ4C2GkUFFmVxBIqqfvgXUyqifrz7ZsrmxoeEYmng+OWxhwToykILGUYR4ngMgF78C"+
"AwEAATANBgkqhkiG9w0BAQQFAANBAMzDmFK2/QAxgn085SLQ+bmYBatuji2YPVDgmMYa3ebhFgUe"+
"I7CKLQTxFg1Y2bw71LFww0Mi9cxwrR+Lt9jhnes=";

    static boolean verifyPeerCert(org.omg.CORBA.Object obj) throws Exception
    {
        org.omg.CORBA.ORB orb = oracle.aurora.jndi.orb_dep.Orb.init();

        // Get the SSL current
        AuroraCurrent current = AuroraCurrentHelper.narrow
            (orb.resolve_initial_references("AuroraSSLCurrent"));

        // Check the cipher
        System.out.println("Negotiated Cipher: " +
            current.getNegotiatedCipherSuite(obj));
        // Check the protocol version
        System.out.println("Protocol Version: " +
            current.getNegotiatedProtocolVersion(obj));
        // Check the peer's certificate
        System.out.println("The account obj's certificate chain : ");
        byte [] [] certChain = current.getPeerDERCertChain(obj);
        System.out.println("length : " + certChain.length);
        System.out.println("Certificates: ");

        /*
        // JDB 1.2 way
        CertificateFactory cf = CertificateFactory.getInstance("X.509");
        for(int i = 0; i < certChain.length; i++) {
            ByteArrayInputStream bais = new ByteArrayInputStream(certChain[i]);
            Certificate xcert = cf.generateCertificate(bais);
            System.out.println(xcert);
            if(xcert instanceof X509Certificate)
            {
                X509Certificate x509Cert = (X509Certificate)xcert;
                String globalUser = x509Cert.getSubjectDN().getName();
                System.out.println("DN out of the cert : " + globalUser);
            }
        }
        */
    }
}
```

```
    }
  }
  */

  // JDK 1.1 way

  java.security.Security.setProperty("cert.provider.x509v1",
    "oracle.security.cert.X509CertificateImpl");

  for(int i = 0; i < certChain.length; i++) {
    javax.security.cert.X509Certificate cert =
      javax.security.cert.X509Certificate.getInstance(certChain[i]);

    String globalUser = cert.getSubjectDN().getName();
    System.out.println("DN out of the cert : " + globalUser);
  }

  return true;
}

public static void main (String[] args) throws Exception {
  if (args.length != 2) {
    System.out.println("usage: Client serviceURL objectName");
    System.exit(1);
  }
  String serviceURL = args [0];
  String objectName = args [1];

  Hashtable env = new Hashtable();
  env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
  env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.SSL_LOGIN);
  env.put(Context.SECURITY_PRINCIPAL, "scott");
  env.put(Context.SECURITY_CREDENTIALS, "tiger");

  // setup the trust point
  env.put(ServiceCtx.SECURITY_TRUSTED_CERT, trustedCert);

  Context ic = new InitialContext(env);

  // Make an SSL connection to the server first. If the connection
  // succeeds, then inspect the server's certificate, since we haven't
  // specified a trust point.
  // Get a SessionCtx that represents a database instance
```

```
ServiceCtx service = (ServiceCtx)ic.lookup (serviceURL);
SessionCtx session1 = (SessionCtx)service.createSubcontext (":session1");
// Lookup login object for the purpose of getting hold of some corba
// object needed for verifyPeerCert(). We should provide an extension
// to just getting the NS object, for this purpose.
LoginServer obj = (LoginServer) session1.activate("/etc/login");

if(!verifyPeerCert(obj))
    throw new org.omg.CORBA.COMM_FAILURE("Verification of Peer cert failed");

// Now that we trust the server, let's go ahead and do our business.
session1.login();
Hello hello = (Hello) session1.activate(objectName);
System.out.println(hello.helloWorld());
}
}
```

Server

```
package helloServer;

import hello.*;

public class HelloImpl extends _HelloImplBase {
    public String helloWorld() {
        String v = System.getProperty("oracle.server.version");
        return "Hello client, your javavm version is " + v + ".";
    }
}
```

Session Example

You can manage sessions in multiple ways, which are all discussed in "[Session Management Scenarios](#)" on page 4-18. The example presented here demonstrates how to access two sessions from a single client.

README

Overview
=====

Twosessions demonstrates a client that instantiates two separate sessions in the server, and calls methods on objects in each session. It also demos use of the login object for client authentication.

Compare this example to the `../examples/corba/session/clientserverserver` example, in which the client instantiates a server object, and that server object then instantiates a second server object in a different session.

Source files
=====

hello.idl

The CORBA IDL for the example. The IDL for the Hello object simply defines two methods:

```
interface Hello
    wstring helloWorld ();
    void setMessage (in wstring message);
```

which must be implemented by the `helloServer.HelloImpl.java` code.

Client.java

You invoke the client program from a command line prompt, and pass it four arguments: the service URL (service ID, hostname, and port), the name of the published object to lookup and instantiate, and a username and password that authenticate the client to the Oracle8i database server.

For example:

```
% java -classpath LIBS Client sess_iiop://localhost:2222 scott tiger
```

where LIBS is the classpath that must include

```
$ORACLE_HOME/lib/aurora_client.jar
#If using Java 2, use classes12.zip instead of classes111.zip
$ORACLE_HOME/jdbc/lib/classes111.zip
$ORACLE_HOME/lib/vbjorb.jar
$ORACLE_HOME/lib/vbjapp.jar
$JAVA_HOME/lib/classes.zip
```

The client first obtains a service context in the normal way, by getting a JNDI Context object, and looking up the service context on it, using the service URL (e.g., `sess_iiop://localhost:2222`). The service context is then used to create new named sessions, `:session1` and `:session2`. On each session, a login server object is instantiated, then a login client is obtained, and the `authenticate()` method on the login client is used to authenticate the client.

Note that this form of authentication is what happens automatically when a server object is instantiated, and the JNDI context is obtained by passing in the username, password, optional database role, and the value `NON_SSL_LOGIN` in the environment's hashtable.

In this example, because the sessions are instantiated overtly, it is necessary to also do the authentication overtly.

After session instantiation and authentication, a `Hello` object is instantiated in each session, the `helloWorld()` method is invoked on each, and the returned `String` is printed on the console.

The printed output is:

```
Hello from Session1
Hello from Session2
```

```
helloServer/HelloImpl.java
```

```
-----
```

This source file implements the two methods specified in the `hello.idl` file: `setMessage()` to set the instance variable `message`, and `helloWorld()` to return the value set in `message`.

Compiling and Running the Example

```
=====
```

UNIX

```
----
```

Enter the command `'make all'` or simply `'make'` in the shell to compile, load, and deploy the objects, and run the client program. Other targets are `'run'` and `'clean'`.

Make sure that a shell environment variable `ORACLE_HOME` is set to point to the home location of the Oracle installation. This is operating system dependent, so see the Installation documentation that came with your system for the location. Also, review the `README` file for the Oracle database, and the `README` file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

Windows NT

On Windows NT, run the batch file `makeit.bat` from a DOS command prompt to compile, load, and deploy the objects. Run the batch file `runit.bat` to run the client program, and see the results.

Make sure that the environment variables `%ORACLE_HOME%`, `%CLASSPATH%`, and `%SERVICE%` are set appropriately for the DOS command window. You can set these as either user or system environment variables from the Control Panel. Double click on System in the Control Panel then on the Environment tab to set these variables. Start a new DOS window after setting environment variable values.

See the Installation documentation that came with your Oracle8i system for the values of these variables. Also, review the `README` file for the Oracle database, and the `README` file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

You can also set an environment variable `%JAVA_HOME%` to point to the root of your Java JDK. For example, `SET JAVA_HOME=C:\JDK1.1.6`.

Hello.IDL

```
module hello {
    interface Hello {
        wstring helloWorld ();
        void setMessage (in wstring message);
    };
};
```

Client.java

```
import hello.Hello;
```

```
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jndi.sess_iiop.SessionCtx;
import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.client.Login;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class Client
{
    public static void main (String[] args) throws Exception {
        if (args.length != 4) {
            System.out.println ("usage: Client serviceURL objectName user password");
            System.exit (1);
        }
        String serviceURL = args [0];
        String objectName = args [1];
        String user = args [2];
        String password = args [3];

        // Prepare a simplified Initial Context as we are going to do
        // everything by hand
        Hashtable env = new Hashtable ();
        env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        Context ic = new InitialContext (env);

        // Get a SessionCtx that represents a database instance
        ServiceCtx service = (ServiceCtx)ic.lookup (serviceURL);

        // Create and authenticate a first session in the instance.
        SessionCtx session1 = (SessionCtx)service.createSubcontext (":session1");
        LoginServer login_server1 = (LoginServer)session1.activate ("etc/login");
        Login login1 = new Login (login_server1);
        login1.authenticate (user, password, null);

        // Create and authenticate a second session in the instance.
        SessionCtx session2 = (SessionCtx)service.createSubcontext (":session2");
        LoginServer login_server2 = (LoginServer)session2.activate ("etc/login");
        Login login2 = new Login (login_server2);
        login2.authenticate (user, password, null);

        // Activate one Hello object in each session
        Hello hello1 = (Hello)session1.activate (objectName);
    }
}
```

```
        Hello hello2 = (Hello)session2.activate (objectName);

        // Verify that the objects are indeed different
        hello1.setMessage ("Hello from Session1");
        hello2.setMessage ("Hello from Session2");

        System.out.println (hello1.helloWorld ());
        System.out.println (hello2.helloWorld ());
    }
}
```

Server

```
package helloServer;

import hello.*;
import oracle.aurora.AuroraServices.ActivableObject;

public class HelloImpl extends _HelloImplBase implements ActivableObject
{
    String message;

    public String helloWorld () {
        return message;
    }

    public void setMessage (String message) {
        this.message = message;
    }

    public org.omg.CORBA.Object _initializeAuroraObject () {
        return this;
    }
}
```

Applet Example

JDK and Jinitiator Applets

README

For all oracle clients, you need to set the following in the

environment before creating InitialContext
 env.put(ServiceCtx.APPLET_CLASS, this);// This refers to the applet class

In the HTML file itself, you need to set
 ORBdisableLocator=true, for all versions. In addition, if you are using
 JDK 1.2 (i.e, plug in 1.2), then you also need to set the following in the
 HTML file.

```
<PARAM NAME="org.omg.CORBA.ORBClass" VALUE="com.visigenic.vbroker.orb.ORB">
<PARAM NAME="org.omg.CORBA.ORBSingletonClass" VALUE="com.visigenic.vbroker.orb.ORB">
```

The syntax slightly differs from plugin to plugin. Look at the example applets
 in this directory.

Note that you don't need to copy any jars on to the client machine. But if you
 do wish to copy the jar files to the client machine, also set "ClassLoader" to
 the appletClassLoader (this.getClass().getClassLoader()) and things will
 work fine. You also don't need to change any files (like java.security etc.) on
 the client.

HTML for JDK 1.1

```
<pre>
<html>
<title> CORBA Applet talking to 8i</title>
<h1> CORBA applet talking to 8i using java plug in 1.1 </h1>
<hr>
The good old bank example
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
WIDTH = 500 HEIGHT = 50 codebase="http://java.sun.com/products/plugin/1.1/jinst
all-11-win32.cab#Version=1,1,0,0">
<PARAM NAME = CODE VALUE = OracleClientApplet.class >
<PARAM NAME = ARCHIVE VALUE = "oracleClient.jar,aurora_client.jar,vbjorb.jar,vbj
app.jar" >
<PARAM NAME="type" VALUE="application/x-java-applet;version=1.1">
<PARAM NAME="ORBdisableLocator" VALUE="true">
<COMMENT>
<EMBED type="application/x-java-applet;version=1.1"
ORBdisableLocator="true" java_CODE = OracleClientApplet.class java_ARCHIVE = "or
acleClient.jar,aurora_client.jar,vbjorb.jar,vbjapp.jar" WIDTH = 500 HEIGHT = 50
pluginpage="http://java.sun.com/products/plugin/1.1/plugin-install.html">
<NOEMBED></COMMENT>
</NOEMBED></EMBED>
</OBJECT>
```

```

</center>
<hr>
</pre>

```

HTML for JDK 1.2

```

<pre>
<html>
<title> CORBA applet talking to 8i</title>
<h1> CORBA applet talking to 8i using Java plug in 1.2 </h1>
<hr>
The good old bank example
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
WIDTH = 500 HEIGHT = 50 codebase="http://java.sun.com/products/plugin/1.2/jinst
all-11-win32.cab#Version=1,1,0,0">
<PARAM NAME = CODE VALUE = OracleClientApplet.class >
<PARAM NAME = ARCHIVE VALUE = "oracleClient.jar,aurora_client.jar,vbjorb.jar,vbj
app.jar" >
<PARAM NAME="type" VALUE="application/x-java-applet;version=1.1.2">
<PARAM NAME="ORBdisableLocator" VALUE="true">
<PARAM NAME="org.omg.CORBA.ORBClass" VALUE="com.visigenic.vbroker.orb.ORB">
<PARAM NAME="org.omg.CORBA.ORBSingletonClass" VALUE="com.visigenic.vbroker.orb.O
RB">
<COMMENT>
<EMBED type="application/x-java-applet;version=1.1.2"
ORBdisableLocator="true"
org.omg.CORBA.ORBClass="com.visigenic.vbroker.orb.ORB"
org.omg.CORBA.ORBSingletonClass="com.visigenic.vbroker.orb.ORB" java_CODE = Orac
leClientApplet.class java_ARCHIVE = "oracleClient.jar,aurora_client.jar,vbjorb.j
ar,vbjapp.jar" WIDTH = 500 HEIGHT = 50 pluginspage="http://java.sun.com/produ
cts/plugin/1.2/plugin-install.html">
<NOEMBED></COMMENT>
</NOEMBED></EMBED>
</OBJECT>

</center>
<hr>
</pre>

```

HTML for Oracle JInitiator

```

<h1>CORBA applet talking to 8i using JInitiator 1.1.7.18</h1>
<COMMENT>
<EMBED type="application/x-jinit-applet;version=1.1.7.18"

```

```
java_CODE="OracleClientApplet"  
java_CODEBASE="http://mysun:8080/applets/bank"  
java_ARCHIVE="oracleClient.jar,aurora_client.jar,vbjobr.jar,vbjapp.jar"  
WIDTH=400  
HEIGHT=100  
ORBdisableLocator="true"  
org.omg.CORBA.ORBClass="com.visigenic.vbroker.orb.ORB"  
org.omg.CORBA.ORBSingletonClass="com.visigenic.vbroker.orb.ORB"  
serverHost="mysun"  
serverPort=8080  
<NOEMBED>  
</COMMENT>  
</NOEMBED>  
</EMBED>
```

Applet Client

```
// ClientApplet.java  
  
import java.awt.*;  
  
import oracle.aurora.jndi.sess_iiop.ServiceCtx;  
  
import javax.naming.Context;  
import javax.naming.InitialContext;  
import java.util.Hashtable;  
import Bank.*;  
  
public class OracleClientApplet extends java.applet.Applet {  
  
    private TextField _nameField, _balanceField;  
    private Button _checkBalance;  
    private Bank.AccountManager _manager;  
  
    public void init() {  
        // This GUI uses a 2 by 2 grid of widgets.  
        setLayout(new GridLayout(2, 2, 5, 5));  
        // Add the four widgets.  
        add(new Label("Account Name"));  
        add(_nameField = new TextField());  
        add(_checkBalance = new Button("Check Balance"));  
        add(_balanceField = new TextField());  
        // make the balance text field non-editable.  
        _balanceField.setEditable(false);  
        try {
```



```

String serviceURL = "sess_iiop://mysun:2222";
String objectName = "/test/myBank";

// Initialize the ORB (using the Applet).
Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(Context.SECURITY_PRINCIPAL, "scott");
env.put(Context.SECURITY_CREDENTIALS, "tiger");
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
env.put(ServiceCtx.APPLET_CLASS, this);

Context ic = new InitialContext(env);
_manager = (AccountManager)ic.lookup (serviceURL + objectName);
} catch (Exception e) {
    System.out.println(e.getMessage());
    e.printStackTrace();
    throw new RuntimeException();
}
}

public boolean action(Event ev, Object arg) {
    if(ev.target == _checkBalance) {
        // Request the account manager to open a named account.
        // Get the account name from the name text widget.
        Bank.Account account = _manager.open(_nameField.getText());
        // Set the balance text widget to the account's balance.
        _balanceField.setText(Float.toString(account.balance()));
        return true;
    }
    return false;
}
}
}

```

Visigenic Applet

README

To run VisiClient applet, you need to do the following.

```

Start osagent and gatekeeper (with port 16000)
(for gate keeper, create a file called gatekeeper.properties and just
put this entry in there : exterior_port=16000)

```

Then start the Bank server (vbj Server &).

Your browser should have Jinitiator installed (use `ojdk-pc.us.oracle.com` for getting Jinitiator, `jdk 1.1.7.18`)
(Browser security doesn't have to be off, i.e, you may set it to `AppletHost` in Jinitiator)

Then simply connect to `mysun:8080/applets/bank/VisiClient.html`

HTML for Visigenic Client Applet

```
<h1>Visigenic Client applet</h1>
<COMMENT>
<EMBED type="application/x-jinit-applet;version=1.1.7.18"
  java_CODE="VisiClientApplet"
  java_CODEBASE="http://mysun:8080/applets/bank"
  java_ARCHIVE="visiClient.jar,vbjorb.jar,vbjapp.jar"
  WIDTH=400
  HEIGHT=100
  ORBgatekeeperIOR="http://mysun:16000/gatekeeper.ior"
  USE_ORB_LOCATOR="true"
  ORBbackCompat="true"
  serverHost="mysun"
  serverPort=8080
<NOEMBED>
</COMMENT>
</NOEMBED>
</EMBED>
```

Visigenic Client Applet

```
// ClientApplet.java

import java.awt.*;

public class VisiClientApplet extends java.applet.Applet {

    private TextField _nameField, _balanceField;
    private Button _checkBalance;
    private Bank.AccountManager _manager;

    public void init() {
        // This GUI uses a 2 by 2 grid of widgets.
        setLayout(new GridLayout(2, 2, 5, 5));
    }
}
```

```
// Add the four widgets.
add(new Label("Account Name"));
add(_nameField = new TextField());
add(_checkBalance = new Button("Check Balance"));
add(_balanceField = new TextField());
// make the balance text field non-editable.
_balanceField.setEditable(false);
// Initialize the ORB (using the Applet).
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(this, null);
// Locate an account manager.
_manager = Bank.AccountManagerHelper.bind(orb, "BankManager");
}

public boolean action(Event ev, Object arg) {
    if(ev.target == _checkBalance) {
        // Request the account manager to open a named account.
        // Get the account name from the name text widget.
        Bank.Account account = _manager.open(_nameField.getText());
        // Set the balance text widget to the account's balance.
        _balanceField.setText(Float.toString(account.balance()));
        return true;
    }
    return false;
}
}
```

Comparing the Oracle8*i* JServer and VisiBroker VBJ ORBs

This appendix, which is for developers who are familiar with the VisiBroker VBJ ORB, summarizes the main differences between that ORB and the current version of the Oracle8*i* JServer ORB. Each ORB supports multiple styles of usage, but this appendix compares only the most commonly used styles. In particular, it assumes that VBJ clients use the helper `bind()` method to find objects by name, whereas Oracle8*i* clients use the JNDI `lookup()` method for the same purpose. It also assumes that Oracle8*i* clients use Oracle's session IOP to communicate with server objects, though the JServer ORB also supports the standard IOP used by the VBJ ORB.

The differences in the ORBs are summarized in these sections:

- [Object References Have Session Lifetimes](#)
- [The Database Server is the Implementation Mainline](#)
- [Server Object Implementations are Deployed by Loading and Publishing](#)
- [Implementation by Inheritance is Nearly Identical](#)
- [Implementation by Delegation is Different](#)
- [Clients Look Up Object Names with JNDI](#)
- [No Interface or Implementation Repository](#)

At the end of the appendix, equivalent client and server implementations of the same IDL for the VBJ and Aurora ORBs are provided for comparison.

Object References Have Session Lifetimes

The Aurora ORB creates object instances in database *sessions*. When a session disappears, references to objects created in that session become invalid; attempts to use them incur the “object does not exist” exception. A session disappears when the last client connection to the session is closed or the session’s timeout value is reached. An object in a session can set the session timeout value with `oracle.aurora.net.Presentation.sessionTimeout()` optionally providing a client interface to this method, which a client can call if it wants an object to persist after client connections to the session are closed.

The life of a typical Oracle8i CORBA object proceeds as follows:

- A client looks up an object implementation’s name with JNDI specifying the database where the implementation has been published.
- The Oracle ORB responds by instantiating an object of the type, and returning a reference to the client.
- The client calls methods on the object, and may pass the reference to other clients who may then call methods on the object.
- The object ceases to exist when its session is destroyed.

The Database Server is the Implementation Mainline

An Oracle8i server object implementation consists of a single class. Developers do not write a mainline server because the database server is the mainline. If the database is running, all implementations published in that database are available to clients. The database server dynamically assigns MTS threads to implementations. An implementation may multithread its own execution with Java threads.

Server Object Implementations are Deployed by Loading and Publishing

Loading an object implementation into a database with the `loadjava` tool makes that implementation accessible to the ORB running in that database. Publishing an loaded implementation's name to a database's session name space with the `publish` tool makes the implementation accessible to clients by name. Every CORBA object implementation must be loaded but only those whose names will be looked up by clients need to be published.

Implementation by Inheritance is Nearly Identical

To implement the hypothetical interface `Alpha` in Oracle8i, you write a class called `AlphaImpl` which extends `AlphaImplBase` and defines the Java methods that implement the IDL operations. You *may* also provide instance initialization code in an `_initializeAuroraObject()` method which the Oracle ORB will call when it creates a new instance.

Implementation by Delegation is Different

For an Oracle8i implementation by delegation (tie), the class you write extends a class you have defined and implements two Oracle-defined interfaces. The first interface, whose name is the IDL interface name concatenated with `Operations`, defines the methods corresponding to the IDL operations. The second interface, called `ActivatableObject`, defines a single method called `_initializeAuroraObject()`. To implement this method, create and return an instance. Here is a minimal example:

```
// IDL
module hello {
    interface Hello {
        wstring helloWorld ();
    };
};
```

```
};

// Aurora tie implementation
package helloServer;

import hello.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class HelloImpl implements HelloOperations, ActivatableObject
//, extends <YourClass>
{
    public String helloWorld () {
        return "Hello World!";
    }

    public org.omg.CORBA.Object _initializeAuroraObject () {
        // create and initialize an instance and return it, for example ...
        return new _tie_Hello (this);
    }
}
```


Clients Look Up Object Names with JNDI

An Oracle8*i* client can look up a published object by name with CORBA COSNaming or with the simpler JNDI (Java Naming and Directory Interface) which interacts with COSNaming in the client's behalf.

A client creates an initial JNDI context for a particular database with a Java constructor, for example:

```
Context ic = new InitialContext(env);
```

The `env` parameter specifies user name and password under which the client is logging in. Because object implementations run in database servers, CORBA object users (via their clients) must identify and authenticate themselves to the database as they would for any database operation.

To obtain an instance of a published implementation, the client calls the JNDI context's `lookup()` method, passing a URL that names the target database and the published name of the desired object implementation. The `lookup()` call returns a reference to an instance in the target database. A client may pass the reference (perhaps in stringified form) to other clients, and the reference will remain valid as long as the session in which the associated object was created survives. Clients that use copies of the same object reference share the object's database session.

If a client executes `lookup()` twice in succession with the same parameters, the second object reference is identical to the first, that is, it refers to the instance created by the first `lookup()` call. However, if a client creates a second session and does the second `lookup()` in that session, a different instance is created and its reference returned.

No Interface or Implementation Repository

The current version of the Oracle8*i* ORB does not include an interface repository or an implementation repository.

The Bank Example in Aurora and VBJ

The following sections compare implementations of the bank example widely used in VBJ documentation. Both client and server are shown as they would be implemented in Oracle8*i* and VBJ. All implementations use inheritance.

The Bank IDL Module

```
// Bank.idl

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

Aurora Client

```
// Client.java

import bankServer.*;
import Bank.*;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class Client
{
    public static void main (String[] args) throws Exception {

        String serviceURL = "sess_iiop://localhost:2222";
        String objectName = "/test/myBank";
        String username = "scott";
        String password = "tiger";

        Hashtable env = new Hashtable();
        env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        env.put(Context.SECURITY_PRINCIPAL, username);
        env.put(Context.SECURITY_CREDENTIALS, password);
        env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
    }
}
```

```

Context ic = new InitialContext(env);

AccountManager manager =
    (AccountManager) ic.lookup(serviceURL + objectName);

// use args[0] as the account name, or a default.
String name = args.length == 1 ? args[0] : "Jack B. Quick";

// Request the account manager to open a named account.
Bank.Account account = manager.open(name);

// Get the balance of the account.
float balance = account.balance();

// Print out the balance.
System.out.println
    ("The balance in " + name + "'s account is $" + balance);
}
}

```

VBJ Client

```

// Client.java

public class Client {

    public static void main(String[] args) {
        // Initialize the ORB.
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
        // Locate an account manager.
        Bank.AccountManager manager =
            Bank.AccountManagerHelper.bind(orb, "BankManager");
        // use args[0] as the account name, or a default.
        String name = args.length > 0 ? args[0] : "Jack B. Quick";
        // Request the account manager to open a named account.
        Bank.Account account = manager.open(name);
        // Get the balance of the account.
        float balance = account.balance();
        // Print out the balance.
        System.out.println
            ("The balance in " + name + "'s account is $" + balance);
    }
}

```

```
}
```

Aurora Account Implementation

```
// AccountImpl.java
package bankServer;

public class AccountImpl extends Bank._AccountImplBase {
    public AccountImpl(float balance) {
        _balance = balance;
    }
    public float balance() {
        return _balance;
    }
    private float _balance;
}
```

VBJ Account Implementation

```
// AccountImpl.java

public class AccountImpl extends Bank._AccountImplBase {
    public AccountImpl(float balance) {
        _balance = balance;
    }
    public float balance() {
        return _balance;
    }
    private float _balance;
}
```

Aurora Account Manager Implementation

```
// AccountManagerImpl.java
package bankServer;
```

```
import java.util.*;

public class AccountManagerImpl extends Bank._AccountManagerImplBase {

    public AccountManagerImpl() {
        super();
    }

    public AccountManagerImpl(String name) {
        super(name);
    }

    public synchronized Bank.Account open(String name) {
        // Lookup the account in the account dictionary.
        Bank.Account account = (Bank.Account) _accounts.get(name);
        // If there was no account in the dictionary, create one.
        if(account == null) {

            // Make up the account's balance, between 0 and 1000 dollars.
            float balance = Math.abs(_random.nextInt()) % 100000 / 100f;

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

            _orb().connect (account);

            // Print out the new account.
            // This just goes to the system trace file for Aurora.
            System.out.println("Created " + name + "'s account: " + account);

            // Save the account in the account dictionary.
            _accounts.put(name, account);
        }
        // Return the account.
        return account;
    }

    private Dictionary _accounts = new Hashtable();
    private Random _random = new Random();
}
```

VBJ Account Manager Implementation

```
// AccountManagerImpl.java

import java.util.*;

public class AccountManagerImpl extends Bank._AccountManagerImplBase {
    public AccountManagerImpl(String name) {
        super(name);
    }
    public synchronized Bank.Account open(String name) {
        // Lookup the account in the account dictionary.
        Bank.Account account = (Bank.Account) _accounts.get(name);
        // If there was no account in the dictionary, create one.
        if(account == null) {
            // Make up the account's balance, between 0 and 1000 dollars.
            float balance = Math.abs(_random.nextInt()) % 100000 / 100f;
            // Create the account implementation, given the balance.
            account = new AccountImpl(balance);
            // Make the object available to the ORB.
            _boa().obj_is_ready(account);
            // Print out the new account.
            System.out.println("Created " + name + "'s account: " + account);
            // Save the account in the account dictionary.
            _accounts.put(name, account);
        }
        // Return the account.
        return account;
    }
    private Dictionary _accounts = new Hashtable();
    private Random _random = new Random();
}
```

VBJ Server Mainline

```
// Server.java

public class Server {

    public static void main(String[] args) {
```

```
// Initialize the ORB.
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
// Initialize the BOA.
org.omg.CORBA.BOA boa = orb.BOA_init();
// Create the account manager object.
Bank.AccountManager manager =
    new AccountManagerImpl("BankManager");
// Export the newly created object.
boa.obj_is_ready(manager);
System.out.println(manager + " is ready.");
// Wait for incoming requests
boa.impl_is_ready();
}
}
```

Abbreviations and Acronyms

This appendix lists some of the most common acronyms that you will find in the areas of networks, distributed object development, and Java. In cases where an acronym refers to a product or a concept that is associated with a specific group, company or product, the group, company, or product is indicated in brackets following the acronym expansion. For example: CORBA ... [OMG].

This acronym list is intended as a helpful guide only. There are no guarantees that it is complete or even completely accurate.

3GL	third generation language
4GL	fourth generation language
ACID	atomicity, consistency, isolation, durability
ACL	access control list
ADT	abstract datatype
AFC	application foundation classes [Microsoft]
ANSI	American National Standards Institute
API	application program interface
AQ	advanced queuing [Oracle8]
ASCII	American standard code for information interchange
ASP	active server pages [Microsoft] application service provider
AWT	abstract windowing toolkit [Java]
BDK	beans developer kit [Java]
BLOB	binary large object

BOA	basic object adapter [CORBA]
BSD	Berkeley system distribution [UNIX]
C/S	client/server
CGI	common gateway interface
CICS	customer information control system [IBM]
CLI	call level interface [SAG]
CLOB	character large object
COM	common object model [Microsoft]
COM+	common object model, extended [Microsoft]
CORBA	common object request broker architecture [OMG]
DB	database
DBA	database administrator, database administration
DBMS	database management system
DCE	distributed computing environment [OSF]
DCOM	distributed common object model [Microsoft]
DDCF	distributed document component facility
DDE	dynamic data exchange [Microsoft]
DDL	data definition language [SQL]
DLL	dynamic link library [Microsoft]
DLM	distributed lock manager [Oracle8]
DML	data manipulation language [SQL]
DOS	disk operating system
DSOM	distributed system object model [IBM]
DSS	decision support system
DTP	distributed transaction processing
EBCDIC	extended binary-coded decimal interchange code [IBM]
EJB	Enterprise JavaBean
ERP	enterprise resource planning
ESIOP	environment-specific inter-orb protocol

FTP	file transfer protocol
GB	gigabyte
GIF	graphics interchange format
GIOP	general inter-orb protocol
GUI	graphical user interface
GUID	globally-unique identifier
HTML	hypertext markup language
HTTP	hypertext transfer protocol
IDE	integrated development environment interactive development environment
IDL	interface definition language
IEEE	Institute of Electrical and Electronics Engineers
IIOP	internet inter-ORB protocol
IIS	Internet information server [Microsoft]
IP	internet protocol
IPC	interprocess communication
IS	information services
ISAM	indexed sequential access method
ISAPI	Internet server API [Microsoft]
ISO	international standards organization (translation)
ISP	internet service provider
ISQL	interactive SQL [Interbase]
ISV	independent software vendor
IT	information technology
J2EE	Java 2 Enterprise Edition [Sun]
JAR	Java archive (on analogy with tar, q.v.)
JCK	Java compatibility kit [Sun]
JDBC	"Java database connectivity"
JDK	Java developer kit

JFC	Java foundation classes
JIT	just in time
JLS	Java language specification
JMF	Java media framework
JMS	Java messaging service
JNDI	Java naming and directory interface
JNI	Java native interface
JOB	Java Objects for Business [Sun]
JPEG	joint photographic experts group
JRMP	Java remote ??message protocol
JSP	Java server pages [Sun] (sometimes used for Java Stored Procedure [Oracle])
JTA	Java transaction API
JTS	Java transaction service
JWS	Java Web Server [Sun]
KB	kilobyte
LAN	local area network
LDAP	lightweight directory access protocol
LDIF	LDPA data interchange format
LOB	large object
MB	megabyte
MIME	multipurpose Internet mail extensions
MIS	management information services
MOM	message-oriented middleware
MPEG	motion picture experts group
MTS	multi-threaded server [Oracle]
MTS	Microsoft Transaction Server [Microsoft]
NCLOB	national character large object
NIC	network information center [internet]

NNTP	net news transfer protocol
NSAPI	Netscape server application programming interface
NSP	network service provider
NT	New Technology [Microsoft]
OCI	Oracle call interface
OCX	OLE common control [Microsoft]
ODBC	open database connectivity [Microsoft]
ODBMS	object database management system
ODL	object definition language [Microsoft]
ODMG	Object Database Management Group
OEM	original equipment manufacturer
OID	object identifier
OLE	object linking and embedding
OLTP	on line transaction processing
OMA	object management architecture [OMG]
OMG	Object Management Group
OO	object-oriented, object orientation
OODBMS	object-oriented database management system
OQL	object query language
ORB	object request broker
ORDBMS	object-relational database management system
OS	operating system
OSF	Open System Foundation
OSI	open systems interconnect
OSQL	object SQL
OTM	object transaction monitor
OTS	object transaction service
OWS	Oracle Web Server
PB	petabyte

PDF	portable document format [Adobe]
PGP	pretty good privacy
PL/SQL	procedural language/SQL [Oracle]
POA	portable object adapter [CORBA]
RAM	random access memory
RAS	remote access service [Microsoft]
RCS	revision control system
RDBMS	relational database management system
RFC	request for comments
RFP	request for proposal
RMI	remote method invocation [Sun]
ROM	read only memory
RPC	remote procedure call
RTF	rich text file
SAF	server application function [Netscape]
SAG	SQL Access Group
SCSI	small computer system interface
SDK	software developer kit
SET	secure electronic transaction
SGML	standard generalized markup language
SID	system identifier [Oracle]
SLAPD	standalone LDAP daemon
SMP	symmetric multiprocessing
SMTP	simple mail transfer protocol
SPI	service provider interface
SQL	structured query language
SQLJ	SQL for Java
SRAM	static (or synchronous) random access memory
SSL	secure socket layer

TB	terabyte
TCPS	TCP for SSL
TCP/IP	transmission control protocol/internet protocol
TP	transaction processing
TPC	Transaction Processing Council
TPCW	TPC web benchmark
TPF	transaction processing facility
TPM	transaction processing monitor
UCS	universal character set [ISO 10646]
UDP	user datagram protocol
UI	user interface
UML	unified modeling language [Rational]
URI	uniform resource identifier
URL	universal resource locator
URN	universal resource name
VAR	value-added reseller
VB	Visual Basic [Microsoft]
VRML	virtual reality modeling language
WAI	web application interface [Netscape]
WAN	wide area network
WIPS	web interactions per second [TPCW]
WWW	world wide web
XA	extended architecture [X/Open]
XML	extended markup language
jdb	Java debugger [Sun]
tar	tape archive, tape archiver [UNIX]
tps	transactions per second

Symbols

`_get_interface_def` method, 5-6, 5-7
`_initializeAuroraObject` method, 2-25

A

ACID properties, 7-2
acronyms, C-1
ActivatableObject interface, 2-25
 `_initializeAuroraObject` method, 2-25
ADDRESS parameter, 3-11, 3-16
applet
 invoking server objects from, 5-16
 sandbox security restrictions, 5-16
APPLET_CLASS property, 5-17
aurora_client.jar file, 6-10
AuroraCertificateManager class, 6-24, 6-25
 `setCertificateChain` method, 6-24
 `setEncryptedPrivateKey` method, 6-24
AuroraCurrentManager class, 6-20
AuroraTransactionService class, 7-20
 `initialize` method, 7-20, 7-25
aurora.zip, 4-28
authenticate method, 4-20, 6-11
authentication
 defined, 6-5
 logout, 4-19, 6-11
 server-side, 6-20
 using SSL, 6-3

B

`begin` method, 7-9, 7-10, 7-12, 7-20, 7-22, 7-27

`bindds` command, 7-8, 7-17
`bindut` command, 7-7, 7-16
BOA
 `obj_is_ready` method, 2-25

C

callback, 5-3
 client-side authentication, 6-25
 server-side authentication, 6-22
 using SSL, 6-21
callout
 using SSL, 6-21
certificates, 6-20, 6-21, 6-24
 manager, 6-24
ClassLoader property, 5-17
client
 access existing bean, 4-22
 client-side authentication, 6-5
collections
 in IDL, 2-21
`commit` method, 7-9, 7-10, 7-13, 7-20, 7-22, 7-28
configuring, 3-1 to 3-19
 direct to dispatcher, 3-16
 IIOP clients, 3-1 to 3-19
 SSL over TCP/IP, 3-17
Contained object, 5-8
Container object, 5-8
Context
 JNDI object, 4-9
CORBA
 callbacks, 5-3
 Java 2 support, 5-11
 pure CORBA using Java 2, 5-15

- retrieving name service, 4-29
- skeletons, 2-4
- stubs, 2-4
- system exceptions, 2-23
- TIE mechanism, 5-10
- web sites for documentation, 1-9

CosNaming service, 4-1, 4-2, 4-29

Current class

- begin method, 7-20, 7-22
- commit method, 7-20, 7-22
- resume method, 7-20, 7-22
- rollback method, 7-20, 7-22
- rollback_only method, 7-20, 7-22
- suspend method, 7-20, 7-22

D

- data integrity, 6-3
- Database Configuration Assistant, 3-8
- DataSource object
 - binding in namespace, 7-8
 - create dynamically, 7-17
- DebugAgent class, 2-26
 - restart method, 2-26
 - stop method, 2-26
- debugging techniques, 2-25
- DESCRIPTION parameter, 3-10
- dispatchers
 - configuration, 3-11
 - connecting directly, 3-11
 - overview, 3-11

E

- endpoint, 3-13
 - registration, 3-15
- endSession method, 4-19
- example code, A-1
- exceptions
 - in IDL, 2-22

G

- General Inter-Orb Protocol, see GIOP
- get_status method, 7-29

- get_transaction_name method, 7-29
- getCurrent method, 7-20, 7-22
- getTS method, 7-20, 7-22, 7-26

GIOP

- dispatcher configuration, 3-11
- oracle.aurora.server.SGIopServer, 3-9
- presentation, 3-2

H

- hand off, 3-14
- Helper class
 - narrow method, 2-6
- Holder class, 2-6

I

- IDL, 1-4
 - IFR, 5-7
 - interface, 2-3
 - language mapping, 2-3
 - skeleton, 2-7
- idl2java tool, 2-4
- IFR, 5-6, 5-7, 5-8
 - object hierarchy, 5-8
 - overview, 5-6
 - Repository object, 5-6
- IIOP, 1-7, 3-2, 4-15
 - clients
 - connecting to dispatchers, 3-11
 - session-based, 3-9
 - configuring, 5-25
 - MTS_DISPATCHER, 3-3
 - profile, 4-13
 - SSL support, 3-17
- IIOP clients
 - configuring, 3-1 to 3-19
- implementation, 2-6
- init method, 2-24
- InitialContext object, 4-12
- initialize method, 7-20, 7-25
- Inprise, 1-8
 - version supported, 5-11
 - VisiBroker for Java, B-1
- in-session activation, 4-24

- interceptors, 2-25
- interface
 - defined, 2-3
 - IFR, 5-7, 5-8
 - retrieving from IFR, 5-6
- Interface Description Language, see IDL
- Interface Repository, see IFR
- InterfaceDef class, 5-7
- Internet Inter-Orb Protocol, see IIOP

J

- Java 2
 - migrating from JDK 1.1, 5-11
- Java Naming and Directory Interface, see JNDI
- Java Transaction API, see JTA
- Java Transaction Service, see JTS
- java2idl tool, 2-24
- java2iiop tool, 2-24
- javax-ssl-1_1.jar, 4-11, 6-4
- javax-ssl-1_2.jar, 4-11, 6-4
- JDeveloper
 - debugging, 2-25
- JNDI
 - Context object, 4-9
 - initial context, 4-2
 - InitialContext constructor, 4-12
 - lookup method, 4-7, 4-12
- jssl-1_1.jar, 4-11, 6-4
- jssl-1_2.jar, 4-11, 6-4
- JTA
 - client-side demarcation, 7-9
 - enlisting resources, 7-5
 - limitations, 7-6
 - nested transactions, 7-6
 - overview, 7-1, 7-2
 - server-side demarcation, 7-7
 - specification web site, 7-1
 - timeout, 7-18
 - two-phase commit, 7-5, 7-15
- JTS, 7-1
 - begin method, 7-27
 - client-side demarcation, 7-20
 - commit method, 7-28
 - get_status method, 7-29

- get_transaction_name method, 7-29
- getTS method, 7-26
- initializing, 7-19
- invocation steps, 7-19
- limitations, 7-23
- overview, 7-19
- rollback method, 7-28
- rollback_only method, 7-29
- server-side demarcation, 7-22
- suspend method, 7-27

L

- listener, 3-11
 - configuration, 3-13
 - endpoint registration, 3-15
 - hand off, 3-14
 - overview, 3-11
 - redirection, 3-12, 3-13
- loadjava tool, 2-12
- login
 - non-JNDI login, 4-19, 6-11
- Login class, 4-5, 6-11
- LoginServer class, 6-11
 - authenticate method, 4-20, 6-11
- logout method, 4-19, 6-11
- LogoutServer class, 4-19, 6-11
- lookup method, 4-11, 4-12

M

- MTS_DISPATCHERS parameter
 - ADDRESS attribute, 3-16
 - configuration, 5-25
 - overview, 3-3
 - PRESENTATION attribute, 3-9, 3-10, 3-17
 - PROTOCOL attribute, 3-9

N

- NameService
 - retrieving, 4-29
- namespace, 4-3
- narrow method, 2-6
- Net8 Assistant

- configuring, 3-10
- configuring for IIOP clients, 3-6
- NON_SSL_LOGIN value, 4-2, 4-10

O

- obj_is_ready method, 2-25
- object activation, 2-24, 4-28
 - in-session, 4-24, 4-28
- Object class
 - _get_interface_def method, 5-7
- oracle.aurora.server.SGiopServer, 3-9
- OracleJTADatasource class, 7-18
- ORB
 - initialization, 2-24, 5-12, 6-24
- ORBClass property, 5-15, 5-18
- ORBDefaultInitRef, 4-33
- ORBdisableLocator property, 5-18
- ORBInitRef, 4-33
- ORBSingletonClass property, 5-15, 5-18
- OSS.SOURCE.MY_WALLET parameter, 3-19

P

- parameter passing
 - by value, 2-20
- presentation
 - GIOP, 3-2, 3-9
 - oracle.aurora.server.SGiopServer, 3-9
- PRESENTATION attribute, 3-9, 3-10, 3-11, 3-17
- property
 - ORBClass, 5-15
 - ORBSingletonClass, 5-15
- PROTOCOL attribute, 3-9
- PROTOCOL_STACK parameter, 3-11
- publish command
 - IFR, 5-7
- publish tool, 2-12, 5-7
- published object
 - permissions, 4-4

R

- RAW session layer, 3-11
- redirection, 3-12, 3-13, 3-16

- regep tool, 3-15, 3-16
- Repository object, 5-8
 - IFR, 5-6
- restart method, 2-26
- resume method, 7-20, 7-22, 7-27
- rollback method, 7-9, 7-10, 7-13, 7-20, 7-22, 7-28
- rollback_only method, 7-20, 7-22, 7-29

S

- Secure Socket Layer, see SSL
- SECURITY_AUTHENTICATION property, 4-10
- SECURITY_CREDENTIALS property, 4-10
- SECURITY_PRINCIPAL property, 4-10
- SECURITY_ROLE property, 4-10
- server-side authentication, 6-5
- service name, 2-9, 4-6, 4-11
- session
 - logout, 4-19, 6-11
 - routing, 4-14
 - terminating from server-side, 4-19
- SESSION attribute, 3-11
- setCertificateChain method, 6-24
- setEncryptedPrivateKey method, 6-24
- setTransactionTimeout method, 7-19
- SID, 4-6
- skeletons, 2-7
- SQLJ, 5-2
- SSL, 6-20
 - configuring, 3-17
 - defined, 6-3
 - JAR files, 4-11, 6-4
 - protocol version numbers, 6-4
- SSL_CLIENT_AUTHENTICATION
 - parameter, 3-19
- SSL_CLIENT_AUTH value, 4-11
- SSL_CREDENTIAL value, 4-10
- SSL_LOGIN value, 4-10
- SSL_VERSION parameter, 3-19
- SSL_VERSION property, 3-19
- start method, 2-26
- stop method, 2-26
- suspend method, 7-20, 7-22, 7-27
- system exceptions, 2-23
- system identifier, see SID

T

TIE mechanism, 5-10
trace files, 2-26
transaction
 client-side demarcation, 7-9
 context propagation, 7-4
 demarcation, 7-3
 enlisting resources, 7-5
 global, 7-3
 limitations, 7-6, 7-23
 overview, 7-2
 server-side demarcation, 7-7
 single-phase commit
 example, 7-8
 timeout, 7-18
 two-phase commit, 7-5, 7-15
Transaction class, 7-3
TransactionManager class, 7-3
TransactionService class, 7-19, 7-20, 7-24, 7-25
 getCurrent method, 7-20, 7-22
TRANSPORT_TYPE property, 5-25
TS class
 getTS method, 7-20, 7-22
TTC, 4-13
two-phase commit, 7-15
two-task common, see TTC

U

URL
 syntax for, 4-5
URL_PKG_PREFIXES property, 4-9
USE_SERVICE_NAME property, 4-11
UserTransaction interface, 7-24
UserTransaction object
 begin method, 7-9, 7-10, 7-12
 binding in namespace, 7-7
 commit method, 7-9, 7-10, 7-13
 retrieving, 7-7
 rollback method, 7-9, 7-10, 7-13
 setTransactionTimeout method, 7-19
useServiceName flag, 4-6
 deployejb option, 4-11

V

version
 Visibroker, 5-11
VisiBroker for Java, 1-8

W

wallet, 6-20
web sites
 CORBA, 1-9

