

ORACLE APPLICATION SERVER

DISTRIBUTED APPLICATIONS

INSIEL S.p.A.

Thursday, 26 November 1998

H.L.H. de Penning

SUMMARY

Oracle Application Server is a collection of middleware services and tools that provide a scalable, robust, secure, and extensible platform for distributed, object-oriented applications. This platform is particularly suited to the needs of the enterprise. Oracle Application Server supports access to applications from both web clients (browsers) using the Hypertext Transfer Protocol (HTTP), and Java applet clients, using the Common Object Request Broker Architecture (CORBA) and the Internet Inter-ORB Protocol (IIOP).

Oracle Application Server provides a more powerful and efficient kind of server-side application called a cartridge server. A cartridge is a shared library that either implements program logic or provides access to program logic stored elsewhere, such as in a database.

The JWeb cartridge contains a Java Virtual Machine and Java class libraries and provides a runtime environment for Java applications on the server side. A Java application is a Java class that starts executing from its main() method. Java applications are platform-independent and they can access all the functionality of Java, including database access and running legacy libraries (in C) through native method interface.

Oracle Application Server version 4.0 supports a component-based application model in the form of JCORBA. In a component-based application model, you build applications by integrating components using development tools often referred to as an integrated development environment (IDE).

The JCORBA model supports CORBA applications written in Java. Components in the JCORBA model are JCORBA objects (JCO), which are Java classes that can be packaged together to form an application running on Oracle Application Server.

Oracle Application Server 4.0 introduces a new management interface for administration, configuration, and management. The new interface uses both HTML forms and Java navigation applets to allow an administrator to maintain an entire Oracle Application Server site.

PREFACE

This report is a result of a literature study for an internship at INSIEL S.p.A., Trieste, Italy. It is part of a series of reports that we have written to transfer our knowledge about an environment for distributed applications to the developers of the company. On this environment we have build a data mining system using genetic algorithms and a document retrieval system for Intranets. The environment existed out of Oracle Application Server 4, Oracle Database server 8, ConText Cartridge, Netscape Communicator 4 and Suns Java 1.1.4.

This report describes the Oracle Application Server and the two cartridges, JWeb and JCORBA, we used in building our distributed applications.

Other reports in this serie are:

- Introducing CORBA; A short overview
- Oracle Application Server; Distributing applications
- ConText; Intelligent Oracle Text Databases
- IntraSearch; Document Retrieval for Intranets using Oracle
- Genetic Search Base; Using genetic algorithms

CONTENTS

SUMMARY.....	1
PREFACE.....	2
INTRODUCTION.....	6
ORACLE APPLICATION SERVER	7
Web Servers and Application Servers.....	7
<i>Web Servers</i>	7
<i>Application Servers</i>	7
Types of Web Applications	8
<i>Common Gateway Interface (CGI) Applications</i>	9
<i>Cartridge Servers</i>	9
<i>CORBA Objects in Java</i>	9
Application Deployment.....	10
Cartridges	10
<i>PL/SQL cartridge</i>	10
<i>JWeb cartridge</i>	10
<i>C cartridge</i>	11
<i>LiveHTML cartridge</i>	11
<i>Perl cartridge</i>	11
<i>Oracle Worlds cartridge(VRML)</i>	11
<i>JCORBA cartridge</i>	11
Resource Management	11
Security.....	11
Monitoring and Site Management.....	12
Intercartridge Exchange Service	12
Sessions Service.....	12
Application Development.....	12
Oracle Application Server Administration	13

Oracle Application Server Architecture	13
<i>Clients</i>	13
<i>Sites</i>	14
<i>Listeners</i>	14
<i>Dispatchers</i>	15
<i>Cartridges, Cartridge Servers, and Cartridge Instances</i>	15
<i>Resource Manager (RM)/Proxy</i>	15
<i>Request Handling</i>	15
JWEB	17
Use of the Term “Application”	17
Developing Java Applications for the JWeb Cartridge	17
<i>Environment Variables</i>	18
<i>Building Strategy</i>	18
<i>The main() Method</i>	18
<i>Java Application Models</i>	19
Adding and Running the Application	19
<i>Location of Java Classes</i>	19
<i>Cartridge Configuration</i>	19
<i>Name Spaces of Java Classes</i>	20
<i>Invoking JWeb Cartridges</i>	23
Control Flow	24
Versions of Java Supported	25
Changes for Version 4.0	25
Database Access	25
JCORBA	26
Features Provided by Oracle Application Server	26
Differences Between JCORBA Applications and Other Application Server Applications	26
<i>Transport Protocol</i>	26

<i>Clients</i>	27
<i>Characteristics of JCORBA Applications</i>	27
<i>Cartridges and JCORBA Objects</i>	27
Communication Path.....	27
Architecture Details.....	28
Processes.....	30
Tools and Development Process	30
Creating JCORBA Applications	30
<i>Overview of Steps</i>	31
<i>Properties of JCORBA Objects</i>	31
<i>ObjectManager and Logger Classes</i>	31
<i>Invoking Methods on Other JCORBA Objects</i>	32
<i>Using Java Finalizer Methods</i>	33
<i>The Remote Interface</i>	33
<i>The Deployment Information File</i>	33
<i>The JAR File for Installation</i>	35
<i>Installation</i>	36
<i>Configuration</i>	36
Developing Clients for JCORBA Applications	38
<i>Getting the Object Reference for an Object</i>	39
<i>Invoking Methods on the Object</i>	43
<i>Using the ObjectFactory Object</i>	43
<i>Using the WebBrowsers ORB</i>	44
CONCLUSIONS AND FINAL REMARKS.....	45
LITERATURE.....	46

INTRODUCTION

This report describes the Oracle Application Server and two of its cartridges in more detail. These cartridges are JWeb and JCORBA and are described in the third and fourth chapter. In the fifth chapter some conclusions and final remarks are given regarding the use of Oracle Application Server. In the last chapter we give the references of the literature that is used to make this report.

A lot of text is copied literally from the documentation that is delivered with Oracle Application Server 4.0. Some of this is altered and extended with our own knowledge about the Oracle Application Server. This report gives only an overview of the Oracle Application Server, the JWeb cartridge and the JCORBA cartridge and I suggest to read [ORA01], [ORA02] and [ORA03] for the complete documentation on the Oracle Application Server and all its features.

First I start with an overview on the Oracle Application Server.

ORACLE APPLICATION SERVER

Oracle Application Server is a collection of middleware services and tools that provide a scalable, robust, secure, and extensible platform for distributed, object-oriented applications. This platform is particularly suited to the needs of the enterprise. Oracle Application Server supports access to applications from both web clients (browsers) using the Hypertext Transfer Protocol (HTTP), and Java applet clients, using the Common Object Request Broker Architecture (CORBA) and the Internet Inter-ORB Protocol (IIOP).

WEB SERVERS AND APPLICATION SERVERS

Client/server computing architectures are commonly described as having two or more tiers according to how application logic is distributed between client and server. Minimally, a client/server architecture must have a client tier and a server tier. Oracle's Network Computing Architecture (NCA) can be described as a three-tier client/server-computing model in which Oracle Application Server functions as a middle tier, or application tier.

WEB SERVERS

The World-Wide Web has historically used a two-tier, fat-server model, in which application logic is stored almost entirely on the server side. In this model, web browsers form the client tier and HTTP servers with their static HTML pages and CGI programs form the server tier. Similarly, database applications have traditionally been separated into a relatively thin client tier, and a server tier (the database) containing both the application data and the application logic in the form of stored procedures.

APPLICATION SERVERS

Recently, a new three-tier model has been emerging that isolates much of the application and "glue" logic in a new middle tier between the client tier and the database. Oracle Application Server implements this middle tier using the Common Object Request Broker Architecture (CORBA) to go beyond the traditional two-tier models of the World-Wide Web and of database applications.

The Common Object Request Broker Architecture (CORBA)

CORBA (Common Object Request Broker Architecture) [INS01] is an industry-standard model for distributed object-oriented programming. It is defined and maintained by the Object Management Group (OMG), an industry consortium. CORBA specifies the behaviour of Object Request Brokers (ORBs). An ORB is responsible for routing requests from clients and other ORBs to CORBA objects. ORBs communicate with each other using the Internet Inter-ORB Protocol (IIOP). Oracle Application Server provides a CORBA 2.0-compliant ORB that Java applet clients can use to communicate with server-side applications implemented as CORBA objects.

Application Logic in the Thin-client Model

This diagram shows Oracle Application Server as the middle tier of Oracle's three-tier Network Computing Architecture (NCA):

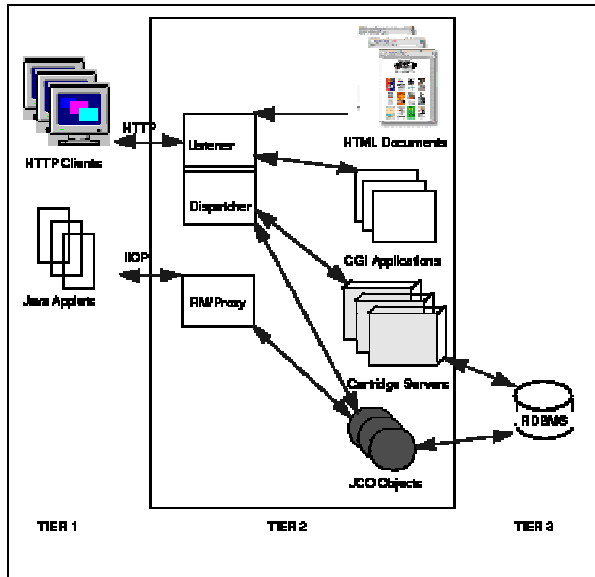


Fig. 1. Three-tier Oracle Application Server Architecture.

In this thin-client three-tier client/server architecture, client software (the client tier) is lightweight enough to be downloaded on demand, and does little but present the user interface for a server-side application. The bulk of the application logic is implemented either in the middle (application) tier, or is stored in the database, as in the case of PL/SQL procedures.

As the middle tier in a three-tier client/server architecture, Oracle Application Server allows application logic to be implemented in the form of cartridges and cartridge servers (described below in Types of Web Applications).

TYPES OF WEB APPLICATIONS

Oracle Application Server supports the following types of web applications:

- Common Gateway Interface (CGI) Applications
- Cartridge Servers
- CORBA Objects in Java

COMMON GATEWAY INTERFACE (CGI) APPLICATIONS

When an HTTP server receives a request specifying a CGI application, the server launches the application and uses the Common Gateway Interface to pass to the application any accompanying query data. This type of application is usually easy to write and CGI is a simple, well-established standard. Oracle Application Server fully supports this kind of application.

CGI applications are most useful for processing simple forms on a small scale. This type of application, however, has many disadvantages. Every time the HTTP server receives a request for a CGI application, it must start a new process to run the application. Each CGI application is also restricted to running on a single machine. While this approach is adequate for small-scale deployment, it does not accommodate a large number of clients without seriously degrading performance.

CARTRIDGE SERVERS

Oracle Application Server provides a more powerful and efficient kind of server-side application, called a cartridge server. A cartridge is a shared library that either implements program logic or provides access to program logic stored elsewhere, such as in a database. A cartridge server is a process in which one or more cartridge instances run. A cartridge instance is a CORBA object within the cartridge server process that executes cartridge code. Unlike CGI programs, cartridge server processes do not have to be restarted for each request. Oracle Application Server listener and dispatcher components route incoming requests to running cartridge servers based on the current server load. The cartridge server that handles the request does not have to run on the same machine that initially received the request, allowing the dispatcher to distribute the request load across multiple machines. These features allow applications implemented as cartridge servers to scale easily and automatically to meet heavy demand.

Applications implemented as cartridge servers also take advantage of the cartridges bundled with Oracle Application Server, such as the PL/SQL cartridge, which provides access to PL/SQL procedures stored in a database. This approach makes sense for sophisticated, large-scale applications that can benefit from the built-in functionality of Oracle Application Server.

CORBA OBJECTS IN JAVA

Oracle Application Server also supports applications implemented as CORBA objects in Java (JCO objects) by providing a CORBA 2.0-compliant ORB. Java applet clients running inside web browsers can use this ORB to obtain object references for these applications, allowing them to call methods on the JCO objects directly. Currently, CORBA objects must be implemented in Java. You would generally deploy this kind of application with a client applet. In this model, the client uses a web browser to download and run the applet. The applet then uses a client ORB built into the browser to communicate with the server-side application using IIOP. It's also possible to use a normal Java application as a client, using the weblistener of OAS to obtain references for the CORBA Objects. How this is done is described later on.

APPLICATION DEPLOYMENT

Oracle Application Server offers the following features relevant to application deployment:

- **Robustness and fault tolerance**-By distributing request handling among many machines, and among many processes running on those machines, Oracle Application Server avoids single points of failure; sophisticated error detection and automatic recovery features help to minimise the effects of failures when they do occur.
- **Scalability**-The distributed architecture and load-balancing capabilities of Oracle Application Server allow you to deploy applications on a large scale, using multiple machines to handle client requests. **Security**-Oracle Application Server supports security standards such as the Secure Sockets Layer (SSL) 3.0 and X.509, and provides a flexible authentication service for authenticating clients. Oracle Application Server 4.0 Enterprise Edition also supports using a Lightweight Directory Access Protocol (LDAP) directory for certificate-based authentication.
- **Extensibility**-You can extend Oracle Application Server's capabilities by developing applications in any of several languages using the C-Web, Java, PL/SQL, Perl, or LiveHTML cartridge.

CARTRIDGES

The following cartridges are bundled with Oracle Application Server 4.0:

PL/SQL CARTRIDGE

Runs PL/SQL stored procedures in Oracle databases. It uses a Database Access Descriptor to locate the database to which to connect. The cartridge can also run files that contain PL/SQL source code; it loads the contents of the file into the database and executes the code. The PL/SQL cartridge comes with the PL/SQL Web Toolkit, which enables you to get information about the request, specify values for HTTP headers such as the content type and cookies, and generate HTML tags.

JWEB CARTRIDGE

Runs Java applications. The JWeb cartridge contains a Java Virtual Machine that interprets the bytecodes for the Java application. When you configure the cartridge, you specify where the class files for the applications are located.

You can also use the JWeb cartridge to connect to and access data from databases in two different ways:

- (1) You can use the pl2java utility, which generates Java methods for procedures and functions in the database. You can then invoke the methods from your Java application just like regular Java methods;
- (2) You can use the JDBC interface, which enables you to execute SQL statements.

The JWeb cartridge comes with the JWeb Toolkit, which enables you to get information about the request, connect to the database, specify values for HTTP headers such as the content type and cookies, and generate HTML tags.

C CARTRIDGE

Runs C applications. To use this cartridge, you implement call-back functions that are invoked by the application server. The C cartridge comes with the WRB API, which contains functions and data structures that you can use to get information about the request, specify values for HTTP headers such as the content type and cookies, and send requests to other cartridges.

LIVEHTML CARTRIDGE

Interprets Server-Side Includes (SSI) documents. SSI is a standard that allows you to embed dynamic data in an otherwise static HTML document. You use special tags in your document to mark the places where the cartridge will substitute dynamic data. You can also embed Perl scripts in documents for the LiveHTML cartridge to generate dynamic data. This feature lets you generate more complex dynamic data than what is supported by SSI.

The LiveHTML cartridge supports Web Application Objects. This feature enables you to group pages into an “application” object and share data across requests or even between users.

PERL CARTRIDGE

Runs Perl scripts. The Perl cartridge also comes with the oraperl module, which enables you to access Oracle Databases.

ORACLE WORLDS CARTRIDGE(VRML)

Interprets VRML data. The data can be stored as files in the operating system, or it can be generated dynamically from data stored in a database.

JCORBA CARTRIDGE

Runs JCORBA objects. This cartridge differs from the other cartridges in that it not enables the creation of applications but objects. These objects are CORBA objects and are grouped together in an Oracle application. The client can use these objects directly using IIOP and not HTTP like with the other cartridges. This means that the clients of the JCORBA applications can not be webbrowsers but Java applets, applications and other CORBA objects.

RESOURCE MANAGEMENT

Oracle Application Server resource management services make applications scalable by providing automatic load-balancing and failure recovery.

SECURITY

To address the security concerns of clients, Oracle Application Server supports the Secure Sockets Layer (SSL) 3.0 standard both for HTTP and IIOP requests. (The International version of Oracle Application Server uses 40-bit encryption.)

To protect the security of the server, Oracle Application Server provides an authentication service that supports both database- and directory-based authentication of clients. Transparent client authentication is also supported through the use of X.509v3 certificates.

Oracle Application Server also supports access control list (ACL)-based and role-based authorisation for client requests. Role information is stored in a Lightweight Directory Access Protocol (LDAP) directory. This allows Oracle Application Server to regulate server access with more precision than is possible with authentication alone.

MONITORING AND SITE MANAGEMENT

Oracle Application Server provides tools and built-in support for monitoring your site, listeners, and applications. Applications can use the Logging service API to record information in log files. Oracle Application Server also provides support for Common Logfile Format (CLF) and Extended Logfile Format (XLF) system message formats. The Oracle Application Server Manager provides tools for analysing log files and for tracking and viewing statistics for specific sites, listeners, and applications. These tools also allow you to generate reports on these statistics.

INTERCARTRIDGE EXCHANGE SERVICE

The Intercartridge Exchange service provides an API that cartridges can use to issue requests and receive responses from other cartridges using a transport-independent, stateless protocol that mimics HTTP.

SESSIONS SERVICE

The Sessions service allows cartridges to be set up so that successive requests from a particular client are handled by the same cartridge instance. This allows you to code your cartridge to maintain state across requests submitted by a specific client (browser). For example, if you are creating a shopping cart cartridge, you can enable sessions in the cartridge so that it can remember the items that the client has in his shopping cart. Each client would have its own cartridge instance, and cartridge instance data would not be shared across clients. You can enable and manage sessions in two ways:

1. You can enable sessions for your cartridge using the Oracle Application Server Manager. This is called “configurable sessions”.
2. You can use the session API to manage sessions. This is called “programmable sessions”. Programmable sessions give you more control than configurable sessions.

Sessions work with all listeners supported by Oracle Application Server. For more information about the Sessions service, see [ORA03].

APPLICATION DEVELOPMENT

Oracle Application Server provides several options for developing applications. Most of the bundled cartridges described in the chapter Cartridges provide deployment platforms for cartridges written in a particular language and with a particular emphasis.

- If you have a large investment in PL/SQL procedures stored in your Oracle database, you can use the PL/SQL cartridge to provide clients easy access to these procedures.
- If you prefer to develop in Java, you can use the JWeb cartridge to develop web applications, taking advantage of the JWeb Toolkit to respond to HTTP requests, and the pl2java utility to access PL/SQL procedures stored in the database.

- If you need to support CORBA/IIOP clients, you can use Java to develop JCO objects and deploy them on Oracle Application Server.
- If you prefer to write Perl scripts, both the Perl and LiveHTML cartridges allow web clients to access these scripts using HTTP.
- If you need to extend the capabilities of your Oracle Application Server installation, the C cartridge provides a powerful general-purpose API for developing web applications in the C language.

For detailed information about developing cartridges for Oracle Application Server, see [ORA03].

ORACLE APPLICATION SERVER ADMINISTRATION

Oracle Application Server 4.0 introduces a new management interface for administration, configuration, and management. The new interface uses both HTML forms and Java navigation applets to allow an administrator to maintain an entire Oracle Application Server site. The new administration tools use a Java-based navigation tree to provide a view of the entire site configuration at a glance. To view the navigation applet, your web browser must be configured to run Java (JDK 1.0.2-compliant) applets.

ORACLE APPLICATION SERVER ARCHITECTURE

This chapter describes the major components of Oracle Application Server 4.0 and how they interact to provide a middle tier for server-side applications.

CLIENTS

There are three kinds of clients that can use Oracle Application Server:

Web browsers-these clients use HTTP to communicate with Oracle Application Server.

Java applets-these clients run within web browsers equipped with client ORBs that communicate with the Oracle Application Server ORB using IIOP.

Java applications-these clients also use HTTP to communicate with Oracle Application Server. But this communication is limited to JCORBA instantiation and destruction only. To use the JCORBA objects these clients communicate with the Oracle Application Server ORB directly using IIOP.

SITES

A site is a collection of Oracle Application Server machines that together form a distributed platform for server-side applications. For each site, a single machine, called the primary node, hosts the broker and resource manager(RM)/proxy and stores configuration data for the entire site. The other machines that comprise the site are called remote nodes.

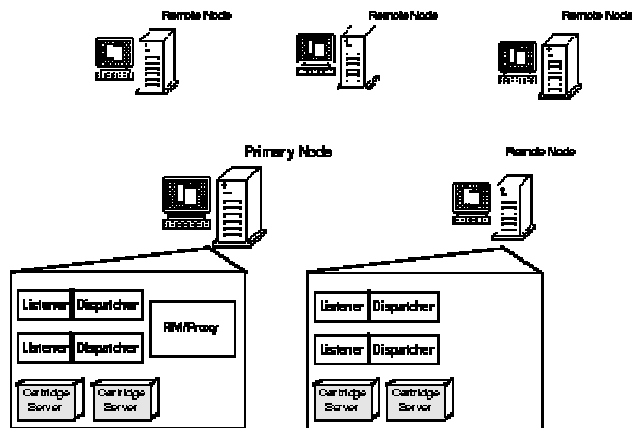


Fig. 2. Oracle Application Server site.

For more information about sites, see [ORA02].

LISTENERS

A listener is an HTTP server; Oracle Application Server supports the following listeners:

- Oracle Web Listener (included with Oracle Application Server)
- Netscape FastTrack Server
- Netscape Enterprise Server
- Microsoft IIS
- Apache

Oracle Application Server provides adapters for each type of listener that allow listeners to work together with the dispatchers in a tightly integrated way, optimising performance. For more information about listeners, see [ORA02].

DISPATCHERS

A dispatcher manages a pool of cartridge server instances running on one or more nodes; each cartridge server instance runs one or more cartridge instances. There is one dispatcher associated with each listener on each node of a web site. When a listener receives an HTTP request that does not identify a static HTML page or CGI program, it passes the request to its dispatcher, which assigns a request to a cartridge instance of the appropriate type. This process is described in more detail in Request Handling below. For more information about dispatchers, see [ORA02].

CARTRIDGES, CARTRIDGE SERVERS, AND CARTRIDGE INSTANCES

A cartridge is a shared library that either implements program logic or provides access to program logic stored elsewhere, such as in a database. A cartridge server is a process in which one or more cartridge instances runs. A cartridge instance is a CORBA object running within the cartridge server process that executes cartridge code.

Most of the cartridges bundled with Oracle Application Server, such as the JWeb cartridge and the PL/SQL cartridge, provide an API in a particular language and a platform for custom applications. The PL/SQL cartridge, for example, provides the PL/SQL Web Toolkit, a collection of PL/SQL procedures that custom applications can use to respond to HTTP requests with database content. The JWeb cartridge provides a similar JWeb Toolkit. See “Cartridges” in the second chapter for descriptions of the bundled cartridges. For more information about cartridges and cartridge servers, see [ORA03].

RESOURCE MANAGER (RM)/PROXY

A resource manager is a component of the Oracle Application Server ORB. It is responsible for obtaining JCO object references on behalf of external clients. There is one resource manager per site.

REQUEST HANDLING

Oracle Application Server can handle both HTTP requests from web browser clients, and CORBA/IIOP requests from Java applet and application clients by way of a client ORB.

Life Cycle of an HTTP Request

This diagram shows the sequence of events when a web browser client issues an HTTP request to a server-side application:

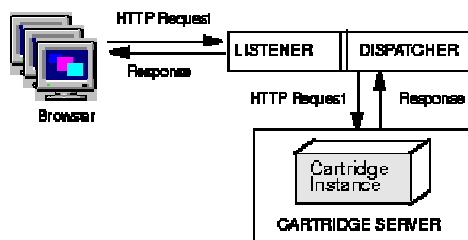


Fig. 3. HTTP request life cycle.

The dispatcher maintains a cache of available cartridge instances for each cartridge. When a request arrives for a particular cartridge, the dispatcher routes the request to one of its cached cartridge instances of the appropriate type.

If the dispatcher has no such cartridge instance available, it requests that the Oracle Application Server runtime creates a new cartridge server and creates the appropriate cartridge instance within that cartridge server. The dispatcher then adds this cartridge instance to its cache and assigns the pending request to it. When the request is complete, the new cartridge instance is available to handle a new request.

Life Cycle of a CORBA/IIOP Request

This diagram shows the sequence of events when a Java applet issues a CORBA/IIOP request to a server-side JCORBA (JCO) object

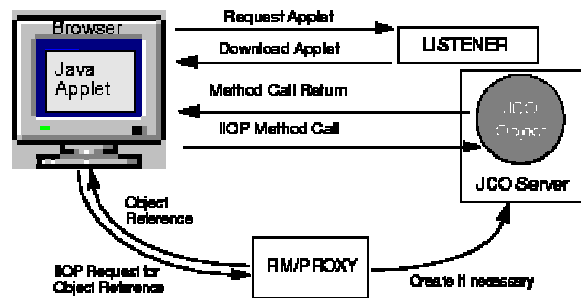


Fig. 4. CORBA/IIOP request life cycle.

When a client requests an object reference from the Resource Manager(RM)/Proxy, the Resource Manager requests the object reference from the Oracle Application Server runtime. If necessary, the Oracle Application Server runtime instantiates the object within a JCORBA cartridge server. The Oracle Application Server runtime then passes back the object reference to the Resource Manager, which in turn passes it to the client. The client then uses the object reference to call methods on the object directly.

The JWeb cartridge contains a Java Virtual Machine and Java class libraries and provides a runtime environment for Java applications on the server side. A Java application is a Java class that starts executing from its main() method. Java applications are platform-independent and they can access all the functionality of Java, including database access and running legacy libraries (in C) through native method interface.

Although you can run Java applications using CGI from Oracle Application Server, using the JWeb cartridge provides the following advantages:

- The JWeb cartridge provides better performance than CGI because there is no start-up and shutdown of the Java Virtual Machine required for each request.
- The JWeb cartridge enables you to access load-balancing, scalability, monitoring, logging, sessions, and other features of the Web Request Broker (WRB).
- The JWeb cartridge minimises use of resources by running multiple Java applications on the same virtual machine and/or handling multiple requests for the same application using the same instance of the application.
- The JWeb cartridge comes with the JWeb Toolkit, which is a set of classes that you can use to generate HTML, access databases, retrieve and set HTTP headers, and access WRB services.
- The JWeb cartridge runs Java applications on the server side. It is not involved with running applets, which are downloaded and run on the client side.

To invoke a Java application through the JWeb cartridge, you provide the cartridge with configuration information such as virtual path and authentication information. You configure the cartridge using the Oracle Application Server Manager.

USE OF THE TERM “APPLICATION”

The term “application” is used by Java and Oracle Application Server to mean different things. In Java, an application is a Java class that has a main() method. The class starts executing from the main() method, and it can invoke other Java classes. In Oracle Application Server, an application is a container of cartridges. A JWeb application is an Oracle Application Server application that contains JWeb cartridges, which invoke Java

applications. This guide describes how to create Java applications that JWeb cartridges can run.

An instance of a JWeb application runs in a cartridge server process. You create and configure JWeb applications using the Oracle Application Server Manager.

DEVELOPING JAVA APPLICATIONS FOR THE JWEB CARTRIDGE

The JWeb cartridge is a runtime environment, and as such, it does not come with any development tools. To develop Java applications for the JWeb cartridge, you need a Java compiler, debugger, and other tools needed for Java application development. Examples of such tools include Oracle AppBuilder for Java, JavaSoft JDK, SunSoft Java Workshop, Microsoft Visual J++, and Symantec Cafe. With the JWeb cartridge comes the JWeb Toolkit, which contains several Java packages that can be used for developing JWeb Applications. For more information on these packages see [ORA03].

ENVIRONMENT VARIABLES

To compile your Java classes with the JWeb Toolkit, you must include the Toolkit's library in your development environment. Make sure that the following environment variables contain the following paths:

- **JAVA_HOME:**
\$ORAWEB_HOME/jdk.
- **CLASSPATH:**
\$JAVA_HOME/lib/classes.zip:
\$ORAWEB_HOME/classes/services.jar:
\$ORACLE_HOME/ows/cartx/jweb/classes/jweb.jar:
\$ORAWEB_HOME/classes/wrbjidl.jar:
\$ORAWEB_HOME/classes/cosnam.jar
- **LD_LIBRARY_PATH:**
\$JAVA_HOME/lib/sparc/native_threads:
\$ORACLE_HOME/ows/cartx/jweb/lib
- **THREADS_FLAG**
native

BUILDING STRATEGY

Because the JWeb cartridge is a runtime environment, it does not have built-in debugging facilities, other than using print statements to generate messages to standard output or to a log file. You should build and debug as much of your application outside of the application server as possible using the standard Java APIs, then finish the application using the JWeb Toolkit APIs. This allows you to use the full debugging capabilities of your development tool for the standard API parts of your program. You add the JWeb Toolkit APIs when the standard APIs are working properly.

When you debug your applications, some of the functionalities in the JWeb Toolkit may be disabled because your application is not running in the application server environment. You may need to insert dummy code in your application to substitute for JWeb Toolkit calls so that your application can function as if it is running in the JWeb cartridge.

THE MAIN() METHOD

When a JWeb cartridge invokes a Java application, it looks for the application's entry point, which is the main() method. The prototype of main() is consistent with that of Java applications invocable by the Java interpreter: it has one parameter (an array of Strings) and it does not return any value. It must also be public and static. For example, a class invocable by the JWeb cartridge should have a main() method similar to:

```
class HelloWorld {  
  
    // The main method must be public and static  
    public static void main (String args[]) { ... }  
  
}
```

JAVA APPLICATION MODELS

There are two different models of Java applications for the JWeb cartridge:

1. The Java application can generate all of the HTML content.
2. You can use the JWeb cartridge's "dynamic HTML generation" feature, which enables you to store the static portions of the HTML content in files and develop applications that generate only the dynamic portions of the HTML content. The static HTML pages are skeleton HTML pages that contain placeholders that will be replaced with HTML generated by your application. When serving a request, the application merges the static and the dynamic portions. This approach has the following advantages:

Better performance because you construct only the portions of your HTML page that change in each request.

Greater modularity because you can change the HTML layout of your application without touching your Java code.

ADDING AND RUNNING THE APPLICATION

Once your Java classes are compiled, add them to a JAR (Java archive)! and copy this JAR to a directory listed in the CLASSPATH of your Oracle Application Server application or to the physical path mapped to the virtual path for your JWeb cartridge. For example, if the \$ORAWEB_HOME/test/java directory is in the Oracle Application Server application's CLASSPATH, you can copy your JAR to that location.

LOCATION OF JAVA CLASSES

The JWeb cartridge looks in the following places for Java classes, in the following order:

In the directories, zip files, and jar files listed in the CLASSPATH variable

In the physical path directories associated with the virtual paths of the cartridge

The CLASSPATH variable is set at the application level, while the virtual path/physical path mappings are done at the cartridge level. You can use this separation to install Java classes that are to be made available only for a cartridge in the cartridge's physical path. Each cartridge has one or more virtual path/physical path mappings.

The location of a class (whether it is read from CLASSPATH or from the physical path directory) determines the scope of the class, that is, it determines if other classes can access it and what other classes it can access. See the next section for details.

CARTRIDGE CONFIGURATION

For JWeb cartridges, the cartridge configuration section contains the Virtual Paths form. The Virtual Paths form enables you to specify a virtual path for a JWeb cartridge. Users can then specify this virtual path in URLs to invoke the JWeb cartridge. The virtual path is available from all listeners listed in the Web Configuration page for the application. For example, if you specify a virtual path of /myApp, users can invoke your Java applications by typing /myApp/class, where class is the name of a Java class that can be found in CLASSPATH or in the physical path associated with the virtual path.

The JWeb cartridge looks in the following places for Java classes, in the following order:

1. In the directories, zip files, and jar files listed in the CLASSPATH variable
2. In the physical path directories associated with the virtual paths of the cartridge

The CLASSPATH variable is set at the application level, while the virtual path/physical path mappings are done at the cartridge level. You can use this separation to install Java classes that are to be made available only for a cartridge in the cartridge's physical path. Each cartridge has one or more virtual path/physical path mappings.

The location of a class (whether it is read from CLASSPATH or from the physical path directory) determines the scope of the class, that is, it determines if other classes can access it and what other classes it can access. See the next section for details.

NAME SPACES OF JAVA CLASSES

The name space of your Java classes depends on the location of the class. The following figure shows the different name spaces:

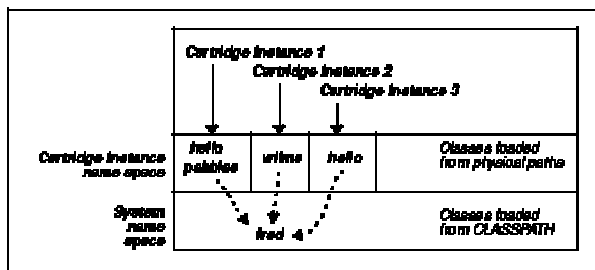


Fig. 5. Name spaces of Java classes.

When a JWeb cartridge loads a class from CLASSPATH, it loads it into the system name space. This means that all other classes, including those in the cartridge instance name space, can access it. In the figure above, the class fred was loaded from CLASSPATH. When a JWeb cartridge loads a class from the physical path associated with the cartridge's virtual path, it loads the class into the name space of the cartridge instance. Classes in the system name space and classes in other cartridge instances cannot access that class instance. In the figure above, the hello and pebbles classes can access each other, and access classes in the system name space (for example, fred), but they cannot access classes in the name spaces of other cartridge instances. For example, they cannot access wilma in instance 2 or hello in instance 3.

When designing your application, consider what other classes your classes access. If you load a class into the system name space (that is, the class is found in CLASSPATH), then it cannot access classes that are found in physical paths. You should place your commonly accessed classes in CLASSPATH, and place only those classes to which you want to limit access in physical paths.

Reflection APIs

Generally, classes in CLASSPATH cannot create new instances of or invoke methods on classes in physical paths. If you use the Java Reflection APIs, however, you can invoke methods on classes in physical paths from classes in CLASSPATH. You cannot create new instances of classes in physical paths from classes in CLASSPATH.

The general steps to invoke methods on physical path classes from CLASSPATH class are:

1. Get a reference to the class instance in physical path. The class instance must already exist.
2. Get the name of the method that you want to invoke and prepare the parameters for the method. This is an array of Objects passed to the invoke() method.
3. Invoke the method using `java.lang.reflect.Method.invoke()`.

In the following example, class Hello creates an instance of class fred, and the instance of fred invokes a method on Hello. Hello is loaded from a physical path, while fred is loaded from CLASSPATH.

```
public class Hello {

    public void foo() {
        // create a new instance of fred
        fred f = new Fred();
        // invoke a method on fred, and pass an instance of Hello
        f.doSomething(this);
    }
}

public class fred {

    public void doSomething(Object helloInstance) {
        Class[] params = null;
        Class helloClass = helloInstance.getClass();
        Method m1 = helloClass.getMethod("method_in_hello", params);

        // invoke the method "method_in_hello" which is in Hello
        m1.invoke(helloInstance, null);
    }
}
```

Static Variables

Static variables are variables whose values do not change from one class instance to the next; all instances see the same value. They are declared like this:

```
public class fred {

    static String fullname = null;
    // ...
}
```

If you use static class variables in your classes, and the classes are loaded in the system name space, then all cartridge instances access the same copy. In the example above, class fred defines a static variable called fullname, which all classes can access. Since each cartridge instance has its own thread, and your classes could create additional threads, you have to ensure that class fred is multi-thread safe.

On the other hand, if you use static variables in classes that are loaded into their own cartridge instance name space (for example, hello), its static variables are accessible only to other classes in the same cartridge instance name space. In the figure above, hello in instance 1 and hello in instance 3 are independent of each other.

Native Libraries

If a class is loaded from a physical path, it cannot make any native calls because the class is loaded into its own name space. You can invoke native calls only if the class is loaded from CLASSPATH. This is a restriction of JDK 1.1.x.

Threads and Location of Classes

When considering where to place your Java classes, you have to consider the number of threads configured for your Oracle Application Server application and whether your classes are multi-thread safe or not. The following table shows the relationship:

	MAX THREADS SET TO > 1	MAX THREADS SET TO 1
<i>Classes are multi-thread safe</i>	Classes can be placed in CLASSPATH or physical path.	Classes can be placed in CLASSPATH or physical path.
<i>Classes are not multi-thread safe</i>	Classes must be placed in physical path.	Classes can be placed in CLASSPATH or physical path.

Java classes loaded from the physical path do not have to be multi-thread safe (unless they create their own threads) because only one thread can execute in a single cartridge instance at any one time. However, in future releases, the JWeb cartridge may allow multiple threads to run in the same instance. To ensure your code can be used in the future release, you should design and code your Java classes to be multi-thread safe.

How to Use Packages with the JWeb Cartridge

It is recommended that you place all your Java class files accessed by a JWeb cartridge directly in the physical path directory associated with the cartridge's virtual path. Using a flat directory structure allows the JWeb cartridge to load the classes in the directory into a private name space. If you must use packages, you have these options:

1. Package Directory Different From the Physical Path Directory

For example, your physical path is `c:\apps\jweb\myApp\cart1`, and your package directory is `c:\myjava\packages`. This enables you to place the package directory in the JWeb application's CLASSPATH, which means that the cartridge loads the package classes into the system name space. Classes loaded from the physical path directory are still loaded in the cartridge instance's name space.

In the following figure, the cartridge loads the hello class from the physical path and places it in the cartridge instance name space, and loads a package class from the CLASSPATH and places it in the system name space.

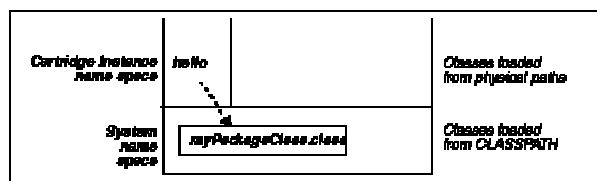


Fig. 6. Loading classes from a directory different from the physical path.

2. Package Directory in a Subdirectory Directly Under the Physical Path Directory

For example, your physical path is `c:\apps\jweb\myApp\cart1`, and your package directory is `c:\apps\jweb\myApp\cart1\myPackage`. In this case, to make the package classes accessible from your cartridge classes, you have to add `c:\apps\jweb\myApp\cart1` to the CLASSPATH. This path is same as the physical path, and this means that your cartridge classes and your package classes are loaded into the system name space.

In the following figure, the cartridge loads the hello class and a package class from the CLASSPATH and places them in the system name space.

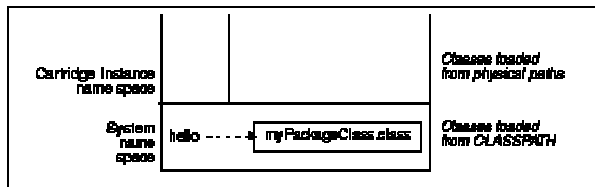


Fig. 7. Loading classes from a directory directly under the physical path.

3. Package Directory Not Directly Under the Physical Path Directory

For example, your physical path is `c:\apps\jweb\myApp\cart1`, and your package directory is `c:\apps\jweb\myApp\cart1\someDirectory\myPackage`. In this case, to make the package classes accessible from your cartridge classes, you have to add `c:\apps\jweb\myApp\cart1\someDirectory` to the CLASSPATH. This means that your cartridge classes are still loaded into the cartridge name space, but your package classes are loaded into the system name space.

In the following figure, the cartridge loads the hello class from the physical path and places it in the cartridge instance name space, and loads a package class from the CLASSPATH and places it in the system name space.

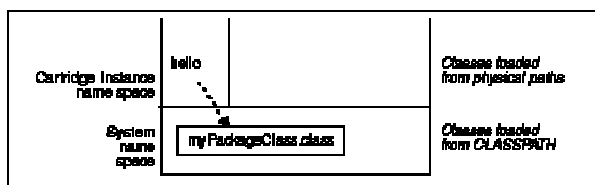


Fig. 8. Loading classes from a directory not directly under the physical path.

INVOKING JWEB CARTRIDGES

To invoke a JWeb cartridge, the URL must be in the following format:
[http://hostname\[:port\]/virtual_path/java_class_name\[?QUERY_STRING\]](http://hostname[:port]/virtual_path/java_class_name[?QUERY_STRING])

where:

- **hostname** specifies the machine where the application server is running.
- **port** specifies the port at which the application server is listening. If omitted, port 80 is assumed.
- **virtual_path** specifies a virtual path mapped to the JWeb cartridge. The virtual path can contain any number of components.
- **java_class_name** specifies the Java class to run. This class must contain the `main()` method.
- **QUERY_STRING** specifies parameters (if any) for the Java class. The string follows the format of the GET method. For example, multiple parameters are separated with the `&` character, and space characters in the values to be passed in are replaced with the `+` character. If you use HTML forms to generate the string (as opposed to generating the string yourself), the formatting will be done automatically for you.

For example, if a browser sends the following URL:

http://www.acme.com:9000/javaApp/jcart/get_emp?fname='john'&lname='doe'&role='office+manager'

the application server running on www.acme.com and listening at port 9000 would handle the request. When the Listener receives the request, it passes the request to the WRB because it sees that the /javaApp/jcart virtual path is configured to call a JWeb cartridge. The WRB sends the request to a cartridge server that is running the cartridge.

The JWeb cartridge instance searches the directories, jar and zip files in its CLASSPATH and also the physical path associated with the virtual path for the get_emp class specified in the URL, and runs the class's main() method. The fname parameter of the procedure gets the value john, the lname parameter gets the value doe, and the role parameter gets the value "office manager". The space character between office and manager is put back in before the cartridge sees the value.

To get the values of the parameters in your Java class, use the getURLParameter() method in the HTTP class. The following example gets the values of the fname, lname, and role parameters:

```
// getRequest() is a static method that returns the current request
HTTP request = HTTP.getRequest();

// get the values of the parameters
String firstname = request.getURLParameter("fname");
String lastname  = request.getURLParameter("lname");
String role      = request.getURLParameter("role");
```

Once a Java application is invoked, the classes that implement the application are cached by the JWeb cartridge. This improves the performance of the application because the cartridge does not have to reload the classes in subsequent invocations. If you modify your Java classes, you need to restart the JWeb cartridge to reload the classes. You can do this from the Oracle Application Server Manager. For more information on adding and running JWeb applications see [ORA03].

CONTROL FLOW

The following figure shows how Oracle Application Server handles a request for a JWeb cartridge:

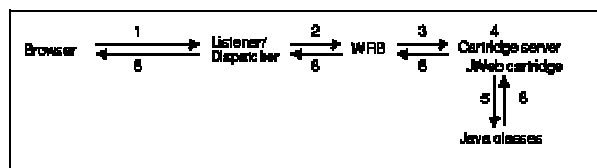


Fig. 9. Control flow for a JWeb cartridge.

1. The Listener component of Oracle Application Server receives a request for a JWeb cartridge from a client. For example, the request can look like: <http://machine/sample/java/HelloWorld>.
2. The Dispatcher sees that the request is for a cartridge and forwards it to the WRB.
3. The WRB examines the URL and sends the request to a JWeb cartridge because the virtual path is mapped to a JWeb cartridge.
4. The JWeb cartridge running in a cartridge server process receives the request, examines the URL, and finds the name of the Java application (class) to invoke. The name of the class is the tail part of the URL. For example, in /sample/java/HelloWorld, HelloWorld is the name of the class.

5. The JWeb cartridge loads the class and invokes its entry point, main().
6. The Java application generates a response, including both the HTTP response header and response body, and returns it through a special output stream (HtmlStream). The JWeb cartridge receives the response and returns it to the WRB. The WRB forwards the response to the browser that invoked the request.

VERSIONS OF JAVA SUPPORTED

The JWeb cartridge supports Java version 1.1.4 or later. You should use a JDK (Java Development Kit) or an IDE (integrated development environment) based on that version for developing Java applications for the JWeb cartridge.

CHANGES FOR VERSION 4.0

- The following features are new for the JWeb cartridge in Oracle Application Server version 4.0:
- The JWeb cartridge supports JDK 1.1.4 or later.
- Oracle Application Server 4.0 introduces the concept of applications, which contain values for environment variables such as JAVA_HOME, CLASSPATH, and LD_LIBRARY_PATH. This enables you to configure each application independently. In version 3.0, these values were set for the JWeb cartridge, and were used for all Java applications.
- Applications contain cartridges, and you can specify the number of instances and threads for each cartridge.
- The JWeb cartridge is multi-thread safe. This improves performance as multiple requests can be handled simultaneously.

DATABASE ACCESS

Java applications running in the context of the JWeb cartridge can access databases in different ways:

- To invoke PL/SQL procedures and functions from Java applications running in the context of a JWeb cartridge, you can use the pl2java utility to generate Java wrapper classes for subprograms in PL/SQL packages. You can then call the wrapper classes to invoke the PL/SQL subprograms. This allows you to implement database logic in PL/SQL to ensure proper control of data in your databases and to invoke existing PL/SQL code from Java applications. For more information see [ORA03].
- To invoke SQL statements directly or access non-Oracle databases, you can use the JDBC package or JdbcBeans. JDBC provides a standard interface to access databases from different vendors. For complete documentation on JDBC, see your vendor's documentation.
- JdbcBeans provides a higher level of abstraction of JDBC. You can drag-and-drop JdbcBeans classes in a graphical development environment that supports JavaBeans, or you can use JdbcBeans classes in a regular fashion (that is, write your code manually). JdbcBeans supports asynchronous database access mode, which is not supported by JDBC.

JCORBA

Oracle Application Server version 4.0 supports a component-based application model in the form of JCORBA. In a component-based application model, you build applications by integrating components using development tools often referred to as an integrated development environment (IDE). Examples of other component-based application technologies include COM/DCOM and OpenDoc.

The JCORBA model supports CORBA applications written in Java. Components in the JCORBA model are JCORBA objects (JCO), which are Java classes that can be packaged together to form an application running on Oracle Application Server. Objects of these classes can be instantiated and accessed from different types of clients such as Java applets, Java applications, CORBA, and DCOM. For more information on CORBA I suggest you read [INS01].

JCORBA objects in JCORBA applications typically provide the business logic. The objects define methods that clients can invoke to perform some operation. An object consists of one or more Java classes, and a JCORBA application consists of one or more objects.

FEATURES PROVIDED BY ORACLE APPLICATION SERVER

When you write JCORBA applications, you get the benefits of the Java language, the CORBA/IIOP scalability features, plus additional features provided by Oracle Application Server such as infrastructure, management, monitoring, and logging capabilities. For example: The management of JCORBA applications is integrated into the application server's management functions. Using the Oracle Application Server Manager, you can manage your JCORBA application and the objects within the application like any other Oracle Application Server application (for example, PL/SQL applications).

You can distribute your application and run it on multiple nodes on the network.

You can perform load-balancing to achieve maximum performance.

You do not have to know or understand CORBA infrastructure to create JCORBA objects because it is generated automatically for you.

You can incorporate JCORBA objects developed by third-parties in your application.

Note that JCORBA objects in JCORBA applications run on the server and therefore do not have a GUI or any other visual representation. They contain methods that perform operations and return values to the clients. The client can display them to the user, if appropriate.

DIFFERENCES BETWEEN JCORBA APPLICATIONS AND OTHER APPLICATION SERVER APPLICATIONS

There are significant differences between JCORBA applications and other applications in the application server environment.

TRANSPORT PROTOCOL

The transport protocol used between clients (for example, applets) and JCORBA objects, and between JCORBA objects running on different Java Virtual Machines is IIOP 2.0, not HTTP. IIOP (Internet Inter-ORB Protocol) is a CORBA transport protocol specified by OMG (Object Management Group, <http://www.omg.org>).

When you create your clients and applications, the application server generates for you infrastructure code (such as stubs and skeletons) to support the CORBA/IIOP architecture. CORBA/IIOP is used to allow clients to have a direct connection to the JCORBA objects. After the initialisation process, client requests and application responses do not go through the application server.

CLIENTS

Clients of JCORBA applications can be Java applets, the traditional client-side of distributed applications, JCORBA objects in the same or different application, but not HTTP-based browsers directly. You cannot enter a URL in a browser to access a JCORBA application. Instead, applets running within browsers are the direct clients of JCORBA applications. Also Java applications can be clients of JCORBA applications. Because they do not have an ORB from for example a browser they need a separate ORB environment installed on the machine where the Java Virtual Machine resides. The advantage of these clients is that they don't need any downloading time as applets do, because the application is already on the users machine, but a disadvantage is that the user needs a separate ORB to run the client application. An advantage of Java applets is that they don't have to be redistributed to the users, when they change, Java application do.

To access JCORBA objects, a client first gets a reference to an object factory that represents the client-side of a meta-factory (see Figure 11). Using the methods in this object factory, the client can get an object reference to a JCORBA object. Once it has an object reference to a JCORBA object, it can access the object directly and invoke methods on the object. When the client is done with the object, it uses the methods in the object factory to destroy the instance of the JCORBA object. See "Developing Clients for JCORBA Applications" for details.

CHARACTERISTICS OF JCORBA APPLICATIONS

When you invoke web-based applications from the browser, the application executes and returns an HTML page to the browser using HTTP. The browser then interprets and renders the HTML.

On the other hand, a JCORBA application consists of a set of components (objects) executing in different address spaces. A component invokes a method on another component and gets a return value. Communication between all components is IIOP-based. In addition to the flexibility of this model, there is no requirement for the client to be a browser.

CARTRIDGES AND JCORBA OBJECTS

When you configure JCORBA applications and objects in Oracle Application Server, you specify a set of attributes for the application and for each object in the application. This is similar to specifying parameters for applications and cartridges for web applications (such as PL/SQL applications). Examples of attributes include the number of instances and threads for the application, names of the Java class files for the objects, and how long an object can be idle before it is destroyed.

A web application contains one or more cartridges. Similarly, a JCORBA application contains one or more objects, and each object can be configured individually.

COMMUNICATION PATH

The following figure shows the communication path between an applet client and a JCORBA application.

The process starts when a browser requests an HTML page containing a Java applet that is the client of a JCORBA application. The application server downloads the page, the applet class files, and a JCORBA JAR file to the client.

1. The applet runs and makes a call to create a specific object in a JCORBA application.
2. The JCORBA object factory interacts with the ORB in the browser and the ORB in the application server (using IIOP) to get an object reference.
3. The application server's resource manager (RM) creates an instance of the specified object in a JCORBA server process.
4. The object reference of the object instance is returned to the applet in the reverse order (resource manager to server ORB to client ORB to JCORBA object factory to applet).
5. The applet uses the object reference to invoke methods on the object directly.

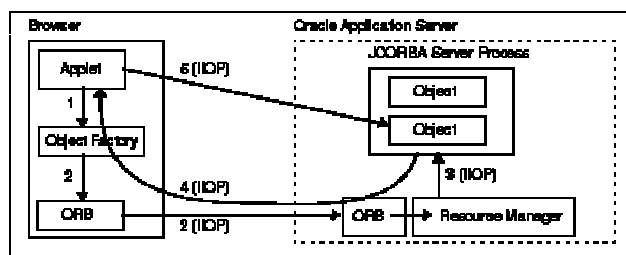


Fig. 10. Applet client talking to an object.

ARCHITECTURE DETAILS

Figure 11 shows the architecture of JCORBA. The rectangles in the figure indicate Java interfaces, and the ovals indicate classes. The shaded areas are Java Virtual Machine processes.

To integrate JCORBA objects into its environment, the application server generates wrappers for the objects. In the architecture diagram (Figure 11), the ServerStack object has a wrapper called StackWrapper.

Object wrappers enable JCORBA objects to work within the application server infrastructure. They manage your JCORBA object and allow other components of the application server to communicate with it. Object wrappers contain methods that your object can invoke. Object wrappers and their corresponding JCORBA objects are instantiated by factory objects.

- In the figure, the first column shows a client of JCORBA objects.
- The second column shows files and objects used by both client and JCORBA objects.
- The third column shows how a JCORBA object interfaces with the application server.
- The rows show the layers in the client and the object.
- The first row (CORBA) indicates that both client and object use the standard `org.omg.CORBA.Object`.
- The second row (application) is the code that you write for your client and JCORBA objects. Both client and object use the object's remote interface, shown in the middle column.

- The third row (generated CORBA infrastructure) is generated for you when you install the application. This layer contains the CORBA objects and interfaces.

- The fourth row (Oracle Application Server-dependent) contains objects that enable the application server to manage JCORBA objects. On the client side, the ObjectFactory communicates with the application server's resource manager (RM) to instantiate JCORBA objects on the server side. On the server side, the MetaFactory and the JavaServer instantiate the actual JCORBA objects.

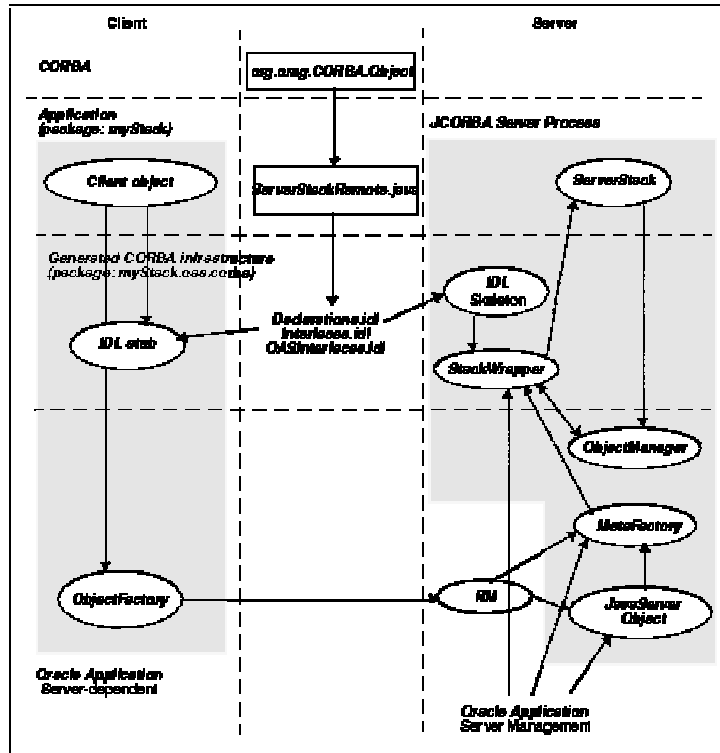


Fig. 11. JCORBA architecture.

PROCESSES

JCORBA objects run in JCORBA server processes (Figure 1-2). Each process is associated with one JCORBA application, and so all the objects running in a server process belong to the same application. You can have more than one JCORBA server process running the same JCORBA application. You might need to do this if you have many clients.

TOOLS AND DEVELOPMENT PROCESS

To develop the objects for JCORBA applications, you can use any Java development environment that supports Java 1.1.4 or later. For example, you can use Oracle AppBuilder for Java, or the JDK from Sun Microsystems. You also need the jar utility to package the application. jar comes with JDK 1.1.4 from Sun.

CREATING JCORBA APPLICATIONS

This chapter describes how to create JCORBA applications and the objects (JCO) in them. See “Developing Clients for JCORBA Applications” for information on how to create clients for JCORBA objects.

OVERVIEW OF STEPS

To create a JCORBA application, you perform the following steps:

1. To be able to compile your JCORBA object, add \$ORAWEB_HOME/jco/lib/jcort.jar to the CLASSPATH of your development environment.
2. Create a JCORBA object and compile it to get .class files.
3. Create a remote interface extending org.omg.CORBA.Object with the methods in the object that you want to be available to remote clients, and compile it to get a .class file.
4. Create a text file called META-INF/JCO.APP to contain deployment information for the application.
5. Create a JAR file that contains the object class files and the META-INF/JCO.APP deployment information file. You can use the jar utility provided with the JDK to do this.
6. Use the Oracle Application Server Manager to add the application to the application server environment.

To invoke methods in objects in a JCORBA application, you need a client. See “Developing Clients for JCORBA Applications”.

PROPERTIES OF JCORBA OBJECTS

A JCORBA object has the following properties:

- Its constructor does not take any parameters.
- Its public methods can be invoked by clients.
- It has no GUI or any other visible components.

The following example shows a simple one-class object called ServerStack:

```
public class ServerStack {  
    public ServerStack() { ... }           // Constructor with no parameters  
    public int getStackSize() { ... }  
    public void setStackSize() { ... }  
    public void push(String value) throws StackException { ... }  
    public String pop() throws StackException { ... }  
}
```

OBJECTMANAGER AND LOGGER CLASSES

In addition to the using standard Java classes, JCORBA objects can use the ObjectManager and Logger classes.

- **oracle.oas.jco.ObjectManager** represents an instance for every JCORBA object. JCORBA objects can use this object to get information about the environment or to access infrastructure services. This includes environment name-value pairs and logger reference.

- **oracle.oas.jco.Logger** enables JCORBA objects to log messages using the logger feature of Oracle Application Server. The Logger class extends java.io.PrintWriter, which contains methods to write formatted messages.

Getting the Reference for the Object Manager

For each object instance, there is an instance of an Object Manager, which is the oracle.oas.jco.ObjectManager class. The class has these methods:

getObjectManager() returns the object manager instance for the specified object.

getSelf() returns a reference to the CORBA object that hosts an instance of a JCORBA object. When a JCORBA object wants to pass itself to another JCORBA object or return itself to the client, it cannot pass its local reference but a reference returned by getSelf(). The receiving object or client should then cast the value to the appropriate remote reference type.

revokeSelf() destroys the corresponding object.

getEnvironment() returns the list of name-value pairs defined for the object.

getLogger() returns a Logger instance, so that the object can use the logging features of the application server.

For more detail I suggest you read [ORA03].

The following example gets the reference for the object manager, and uses it to get a value from the name-value pair list. The value is used to set the stack size.

```
import oracle.oas.jco.*;

public class ServerStack {

    ObjectManager om;

    private void init() {
        if (om != null) return;
        om = ObjectManager.getObjectManager(this);
        String size = om.getEnvironment().getProperty("stackSize");
        try {
            setStackSize(Integer.parseInt(size));
        } catch (Exception e) {
            Logger l = om.getLogger();
            l.setSeverity(LOG_SEVERITY_ERROR);
            l.println("Invalid stacksize" + e);
        }
    }
}
```

INVOKING METHODS ON OTHER JCORBA OBJECTS

One JCORBA object can invoke methods on another object. The object invoking the method is a client of the other object, and the process requires the client object to get an object reference for the other object before it can invoke the object's methods. See "Developing Clients for JCORBA Applications".

USING JAVA FINALIZER METHODS

If your JCORBA object uses finalizer methods (which are executed just before the Java Virtual Machine shuts down), make sure that the finalizer methods do not perform any ORB operations, such as calling any of the methods in the `oracle.oas.jco.ObjectManager` class. The reason is that the Java Virtual Machine does not guarantee the order in which the finalizers are run, and it could shut down the ORB before running your object's finalizer method.

THE REMOTE INTERFACE

The remote interface lists the methods in a JCORBA object that clients can invoke. The interface must extend the `org.omg.CORBA.Object` interface, and if a method in the object throws an exception, the method in the interface must also throw the same exception or a superclass of the exception in the object. For example, the remote interface for the `ServerStack` object looks like the following:

```
public interface ServerStackRemote extends org.omg.CORBA.Object {  
  
    public int getStackSize();  
    public void setStackSize();  
    public void push(String value) throws StackException;  
    public String pop() throws StackException;  
}
```

The name of the remote interface is specified in the deployment information file.

THE DEPLOYMENT INFORMATION FILE

The deployment information file (`META-INF/JCO.APP`) is a text file that contains deployment information for the application and its objects. You include this file in the JAR file that you create.

The deployment information file contains the following sections:

- The `[APPLICATION]` section provides information such as the application name and the default time-out duration.
- The `[APPLICATION.ENV]` section provides name-value pairs that the objects in your application can use.
- The `[<objectName>]` section provides information such as the Java class name for the object, the remote interface name for the object, and optionally the time-out duration.
- The `[<objectName>.ENV]` section provides name-value pairs that are only available for this object.

Each JCORBA object in the application has its own `[<objectName>]` and `[<objectName>.ENV]` sections. For example, if you have three objects in a JCORBA application, you would have one `[APPLICATION]` section, one `[APPLICATION.ENV]` section, three `[<objectName>]` sections, and three `[<objectName>.ENV]` sections.

Each section contains property name-value pairs of the form: `<proprname> = <value>` (`proprname` cannot contain space characters). Note that the contents of the file is case-sensitive. Lines starting with a semicolon character ("`;`") are comments, and the whole line is ignored.

The name of the deployment information file must be META-INF/JCO.APP.

This is a sample deployment information file for the ServerStack application:

```
[APPLICATION]
name = ServerStackApp
idleTimeOut = 2000

[ServerStack]
className = myStack.ServerStack
remoteInterface = myStack.ServerStackRemote

[Stack.ENV]
; property used by the ServerStack object
stackSize = 10
```

The Application Sections

The [APPLICATION] section contains the following properties:

name

The name of the application. The name cannot contain “.” or white space characters. This name is used by the client in the createInstance() method of the ObjectFactory class. This property is required.

usesApplications

A comma-separated list of names of other application installed in Oracle Application Server that are used by the application being installed. This property is optional.

idleTimeOut

How long in seconds an object can be idle before it is destroyed. Default: 0, which indicates no time-out

minInstances

The number of JCORBA object instances that the application server starts up in a JCORBA server process when the process starts up. This value can be over-ridden by the value in the object section. Default: 0

maxInstances

The maximum number of JCORBA object instances that a JCORBA server process can run. The number of instances increases as the process receives requests. This value can be over-ridden by the value in the object section. Default: 10

minThreads

The minimum number of threads that can access a JCORBA object. The number of threads determines the number of requests that an object can handle at the same time. Default: 1

maxThreads

The maximum number of threads that can access a JCORBA object. If there are more object instances than threads, then requests will have to wait until a thread is available. Default: 10

There are no pre-set system properties for the [APPLICATION.ENV] section. This section contains application-specific name-value pairs. For example, if you are creating a mortgage application, you could have a property that specifies the interest rate:

```
[APPLICATION.ENV]
```

InterestRate = 8.0

The Object Sections

The JCORBA object's name is specified between the square brackets. Clients use this name to identify the JCORBA object that they want to access. The object name cannot contain the "." character.

The object name does not have to match the object's Java class name, because the `className` property specifies the Java class associated with the object.

Note that the number of threads and the number of instances properties can be specified at the object level or at the application level. If specified in both places, the value specified at the object level takes precedence.

className

The full name of the Java class for the object. This property is required.

remoteInterface

The full name of the Java interface for the object. See "The Remote Interface" for details. This property is required.

minInstances

The number of object instances that the application server starts up in a JCORBA server process when the process starts up. The default is 0.

maxInstances

The maximum number of object instances that a JCORBA server process can hold. The number of instances increases as the process receives requests. The default is 10.

minThreads

The minimum number of threads in a JCORBA server process. Each object instance uses exactly one thread. The default is 1.

maxThreads

The maximum number of threads in a JCORBA server process. If there are more object instances than threads, then requests will have to wait until a thread is available. The default is 10.

Getting Values from the Deployment Information File

CORBA objects can retrieve values from the JCO.APP file using the `getEnvironment()` method in the `oracle.oas.jco.ObjectManager` class. See the example in the section "Getting the Reference for the Object Manager".

THE JAR FILE FOR INSTALLATION

After you have compiled the .java files of your JCORBA application into .class files and created the JCO.APP deployment information file, you need to create a Jar file to contain these files. The file is used by the Oracle Application Server Manager to install and register the JCORBA application in the application server environment.

To generate the JAR file, just create a regular JAR file containing all the files for your application. The javac compiler provides a -d option that lets you specify the destination directory for your class files.

For example, to place your class files in the /test/myApp/classes directory, you would do something like:

```
prompt> javac -d /test/myApp/classes *.java
```

In the directory that contains the class files, create a directory called META-INF to contain the JCO.APP file.

```
prompt> mkdir /test/myApp/classes/META-INF
prompt> cp JCO.APP /test/myApp/classes/META-INF
```

Generate the Jar file:

```
prompt> cd /test/myApp/classes
prompt> jar cf /test/myApp/myApp.jar *
```

The jar utility is provided by JDK. It can be found in the bin directory of your Java installation.

INSTALLATION

To install a JCORBA application in the application server, use the Oracle Application Server Manager to install the JAR file. You should see your application under Applications in the navigational tree. However, before you can run the application, you need to set some configuration parameters. See the next section for details.

CONFIGURATION

You configure JCORBA applications using the Oracle Application Server Manager. Configuration parameters for JCORBA applications can be divided into two levels: application parameters and object parameters.

Application-Level Parameters

Minimum and maximum number of servers

Each JCORBA server runs one JCORBA application, and each application can service one or more clients, depending on the number of object instances and threads. When the application server starts up, it starts up the minimum number of JCORBA servers. As it receives requests beyond what the minimum number of servers can handle, the application server starts up more servers, up to the maximum number.

User ID and Group ID

These parameters are available in the Enterprise Edition only. These parameters specify the user and group who own the JCORBA server processes.

JCORBA object timeout

When an instance of a JCORBA object has been idle for the specified duration (for example, the client has not made a request for this specified amount of time), the JCORBA Server can sever the connection between the client and the object instance. The timeout is specified in seconds. After severing the connection, the application server can use the object instance to service another client or it can shut down the instance, depending on how the Reusable flag is set. If the timeout is not set or if it is set to 0, then no timeout processing is performed.

This timeout feature is intended to free up instances when clients terminate abnormally and do not get to release the object instance. Your client should still call

oracle.oas.jco.ObjectFactory.destroyInstance() when it is done accessing an object. See “Destroying an Object” for details.

Minimum and maximum number of instances

When a JCORBA server process starts up, it starts up the minimum number of object instances for each object. As it receives requests beyond what the minimum number of instances can handle, it starts up more instances, up to the maximum number. Note that you should specify more than one instance only if multiple instances of your object can run safely on one Java Virtual Machine. When multiple instances are allowed, object instances of one type should not access other object instances of the same type due to possible deadlock situations.

Minimum and maximum number of threads

You can have multiple threads per object type, where each thread handles one client. Although JCORBA guarantees thread-safe access to object instances, when you write your objects, you have to ensure thread-safe access to the object’s static data.

Note that the number of clients that can be handled simultaneously by a JCORBA server depends on the number of threads in your object, not on the number of object instances.

Environment variables

This page defines the values of environment variables used by the application. You can append your own values (if necessary) to the following environment variables:

CLASSPATH

A list of directories or JAR files that contain class files for your objects. For example:

`%ORACLE_HOME%/ows/jco/apps/server/brokerage/serverbrokerage.jar.`

When you install your application, CLASSPATH is set to access all classes contained in the JAR file being deployed as well as other supporting classes. On Unix platforms, directories and JAR files in CLASSPATH are colon-separated (for example,

`%ORACLE_HOME%/ows/jco/apps/server/brokerage/serverbrokerage.jar:%ORACLE_HOME%/ows/jco/apps/server/brokerage/classes.)`

On NT, directories and JAR files in CLASSPATH are semicolon-separated (for example,

`%ORACLE_HOME%/ows/jco/apps/server/brokerage/serverbrokerage.jar;%ORACLE_HOME%/ows/jco/apps/server/brokerage/classes.)`

PATH

A list of directories (colon-separated on Unix, semicolon-separated on NT) that contain executables.

This should be set to contain `%ORAWEB_HOME%/jdk/bin.`

JAVA_HOME

The top-level directory where Java is installed. This should be set to `%ORAWEB_HOME%/jdk.`

THREADS_FLAG

Whether the Java Virtual Machine should use native threads or not. This is set to “native”. This value is required.

Logging parameters

You can enable or disable logging. If enabled, you have to specify the directory and file to which the logged messages are written. You also need to specify the severity levels (between 0 and 15), where low values indicate serious problems. Specifying a high value will cause the logger to log more messages because it writes all messages up to and including that severity level. For example, if you set the severity level at 3, the logger logs messages of severity levels 0, 1, 2, and 3.

Java environment

This page enables you to specify name-value pairs for the entire application. If you want name-value pairs to be visible only for an object, use the Java environment (object level) page. These name-value pairs are read from the JCO.APP file when you install your application.

*Object-Level Parameters***JCORBA object class name**

When you install your application, this value is read from the JCO.APP file. This value cannot be modified.

Global init and remove methods

When the JCORBA server starts up your object, it calls the specified init method. The init method does not take any parameters. When the JCORBA server shuts down your object, it calls the specified remove method, which does not take any parameters. You cannot modify these values.

Factory class name

You cannot modify this value.

Reusable

This parameter is available in the Enterprise Edition only. If set to true, the JCORBA server does not destroy the object instance when the connection between the client and the instance is broken. The instance can then be used to handle other clients. If set false, the JCORBA server calls the specified remove method, and then destroys the instance. The JCORBA server then instantiates a new instance.

Java environment

This page enables you to specify name-value pairs for the object. If you want name-value pairs to be visible to the entire application, use the Java environment (application level) page. These name-value pairs are read from the JCO.APP file when you install your application.

DEVELOPING CLIENTS FOR JCORBA APPLICATIONS

This chapter describes how to create clients for JCORBA applications. See “Creating JCORBA Applications” for information on how to create the objects on the server side.

Clients of JCORBA applications can be:

- Java applets running in browsers
- JCORBA objects in the same or other JCORBA applications
- Java applications
- JWeb cartridges

Clients of JCORBA applications do not include browsers, that is, you cannot specify a URL to invoke a method in an object.

JCORBA applications and their objects are CORBA objects, which are registered in the application server’s ORB. To access these objects, clients use Internet Inter-ORB Protocol (IIOP), instead of HTTP. You also need an ORB on the client side. If your client is a Java applet, you can use the ORB in the browser. See Figure 10.

The way clients invoke methods on remote objects is similar to Java's model of invoking methods on a local object. The difference is that instead of calling "new classType()" to create a new instance of classType on the local machine, you use the createInstance() method in the oracle.oas.jco.ObjectFactory class. The method returns a reference to an object instance. After you are done with the instance, you should call a method to destroy it. An object instance is available to clients until it is destroyed.

Object methods can return any value, and clients are free to decide how they want to display the values to the user. For example, one client can simply display the value in a text area, while another client can display it graphically in a chart.

GETTING THE OBJECT REFERENCE FOR AN OBJECT

Before a client can invoke a method on a JCORBA object, it must first get a reference to an instance of the object. To do this, the client needs an ObjectFactory instance, on which it can call createInstance() to get an instance of the desired object. createInstance()'s parameter specifies the object in the form "<application>/<objectName>", and it returns an object that can be cast to the remote interface associated with the object.

The way you instantiate an ObjectFactory depends on the client. The ObjectFactory class has different constructors for applet clients, application clients, and JCORBA object clients.

To be able to compile your client objects, add \$ORAWEB_HOME/jco/lib/jcort.jar to the CLASSPATH of your development environment.

Applet Clients

The following example shows a Java applet client getting an object reference for the TextFuncs object:

```
public class TextFuncsDemo extends java.applet.Applet {

    private myApp.TextFuncsRemote tfRemote;
    ObjectFactory f;

    public void getObjRef() {
        try {
            f = new ObjectFactory(this); // "this" refers to the applet itself
            tfRemote = (myApp.TextFuncsRemote)
                f.createInstance("myApp/TextFuncs");
        } catch (ObjectFactoryException e) {
            e.detail.printStackTrace();
        }
    }

    // other stuff for the applet
}
```

In the ObjectFactory constructor, the parameter value of "this" specifies the applet. The URL for this applet is used by the ORB in the browser to locate the Oracle Application Server that has the JCORBA application registered.

The <APPLET> tag in the HTML page looks like the following:

```
<APPLET code="myApp.TextFuncsDemo.class"
        codebase="." archive="demo.jar" width=100 height=100>
<PARAM name="org.omg.CORBA.ORBclass" value="com.visigenic.vbroker.org.ORB">
</APPLET>
```

Most browsers allow only one file to be specified in the archive attribute of the <APPLET> tag or, if multiple files are allowed, search only in the codebase directory for the files specified in the attribute. If your browser has these restrictions, you have to create a JAR file that contains all the required files. The required files are:

- The applet's class files
- The files in \$ORAWEB_HOME/../../jco/apps/client/<appName>/_client.jar. This file was created when you installed your application.
- The JCORBA API files in the \$ORAWEB_HOME/../../jco/api/jcoapi.jar
- The ORB files in \$ORAWEB_HOME/../../jco/api/vbjorb.jar

To create one JAR file that contains all the files:

- 1) Create a temporary directory (for example, /tmp/myJCOapp) and cd into it.

```
prompt> mkdir /tmp/myJCOapp
prompt> cd /tmp/myJCOapp
```

- 2) Copy your applet class files to the temporary directory.

- 3) Unjar the deployed client JAR file. This extracts the files from the JAR file and places them in the temporary directory.

```
prompt> jar xf $ORAWWEB_HOME/../../jco/apps/client/<appName>/_client.jar
```

- 4) Unjar the jcoapi.jar file.

```
prompt> jar xf $ORAWWEB_HOME/../../jco/api/jcoapi.jar
```

- 5) Unjar the vbjorb.jar file.

```
prompt> jar xf $ORAWWEB_HOME/../../jco/api/vbjorb.jar
```

- 6) Create a JAR file containing all the files in the temporary directory.

```
prompt> jar cf demo.jar *
```

Application Clients

If your client is a stand-alone Java application, it calls the ObjectFactory constructor in a slightly different syntax:

```
public class TextFuncsDemo {

    private myApp.TextFuncsRemote tfRemote;
    ObjectFactory f;

    public void getObjRef() {
        // This is the machine that is running Oracle Application Server.
        URL oas = new URL("http://mymachine/");
        try {
            f = new ObjectFactory(oas);
            tfRemote = (myApp.TextFuncsRemote)
                f.createInstance("myApp/TextFuncs");
        } catch (ObjectFactoryException e) {
            e.detail.printStackTrace();
        }
    }
}
```

```
}  
  // other stuff for the application  
}
```

The ObjectFactory constructor requires a URL that is running an Oracle Application Server that has the desired JCORBA application registered. The ObjectFactory connects to this URL to get an object reference to the desired object.

JCORBA Objects or JWeb Cartridges as Clients

If your client is an object in the same or different JCORBA application, or a JWeb cartridge, it calls the ObjectFactory constructor without any parameters:

```
public class TextFuncsDemo {

    private myApp.TextFuncsRemote tfRemote;
    ObjectFactory f;

    public void getObjRef() {
        try {
            f = new ObjectFactory();
            tfRemote = (myApp.TextFuncsRemote)
                f.createInstance("myApp/TextFuncs");
        } catch (ObjectFactoryException e) {
            e.detail.printStackTrace();
        }
    }

    // other stuff for the object
}
```

Without any parameters, the ObjectFactory constructor assumes the client (a JCORBA object or JWeb cartridge) and the server JCORBA object are executing within the same application server.

The JCORBA Object Name

The createInstance() method returns an org.omg.CORBA.object object, which you cast to the remote interface of the object. The parameter for the method indicates the full name of the object of interest. The format of the full name is: <application name>/<object name>

The application and object names are defined in the deployment information file. The application name is specified in the name field in the [APPLICATION] section of the file, and the object name is specified between the square brackets [<objectName>] in the object section.

Files Required by the Clients

Clients must have access to the JCORBA client APIs as well as the stubs generated during deployment. The APIs and the stub files are located at:

- \$ORAWEB_HOME/./jco/api/jcoapi.jar
- \$ORAWEB_HOME/./jco/apps/client/<appName>/_client.jar

You should add these files to the CLASSPATH variable.

If your client is using the Visibroker for Java ORB, you must also include the following files in CLASSPATH:

- \$ORAWEB_HOME/./jco/api/vbjorb.jar

INVOKING METHODS ON THE OBJECT

Once you have the reference to a JCORBA object, you can invoke methods on it. The following example uses the `tfRemote` variable from the previous section, and invokes the `encrypt()` method:

```
// get data from text area and encrypt it
private action() {
    if (target is button) {
        tfRemote.encrypt(getText from text area);
    }
}
```

USING THE OBJECTFACTORY OBJECT

The `oracle.oas.jco.ObjectFactory` class enables clients to access JCORBA applications. The class provides methods for creating an `ObjectFactory` object, getting an object reference for a JCORBA application, and destroying the JCORBA application. For more details see [ORA03].

Constructors:

public ObjectFactory() - use this constructor if the client is an object in a JCORBA application.

public ObjectFactory(java.applet.Applet applet) - use this constructor if the client is an applet.

public ObjectFactory(java.net.URL url) - use this constructor if the client is an application. The parameter specifies the URL of an Oracle Application Server listener.

Methods in oracle.oas.jco.ObjectFactory:

createInstance() returns an object reference to the specified object in a JCORBA application.

destroyInstance() destroys the specified object instance.

dispose() destroys the `ObjectFactory` object.

The following sections describe how to use these methods.

Destroying an Object

When a client no longer needs to refer to an object, it should call the `-destroyInstance()` method in the `oracle.oas.jco.ObjectFactory` class. The ORB then releases the object instance so that the Java Virtual Machine can clean it up during garbage collection. The method takes one parameter, indicating the object to destroy.

The following example shows an applet freeing an object in its destroy call-back:

```
public void destroy() {
    f.destroyInstance(tfRemote);
    // f is a private attribute for ObjectFactory
}
```

Releasing the Object Factory

When your client no longer needs objects in a JCORBA application, it should call the `dispose()` method in the `oracle.oas.jco.ObjectFactory` class. This method releases resources used by the Object Factory instance. After you have called this method, the `ObjectFactory` reference is no longer valid. Without an `ObjectFactory`, you cannot destroy existing objects; you should release the factory only after you have destroyed all the JCORBA objects used by the client.

The following example calls the `dispose()` method when the client is done:

```
public void done() {
    f.dispose();
}
```

Getting Information about an Exception

The `createInstance()` and `destroyInstance()` methods can throw `ObjectFactoryException`. If this exception is thrown, you can get more information about it from its detail exception attribute. See “Getting the Object Reference for an Object” for an example. For more details see [ORA03].

USING THE WEBBROWSERS ORB

If your JCORBA applications need to respond to Internet clients, like Java applets, you need to install an IIOP proxy on every listener host of your application server installation. The IIOP proxy allows Java applet clients running in browsers to communicate with machines other than the one from which the applet files were downloaded.

Oracle does not yet provide an IIOP proxy. In the interim, you need to use IIOP proxies from third-parties. Oracle Application Server has been tested with Gatekeeper, an IIOP proxy from Visigenic. You can download a trial version of the product from Visigenic’s web site: <http://www.visigenic.com>. Gatekeeper can be found under the “VisiBroker for Java” section.

Using Gatekeeper with Applet Clients

If you are using Gatekeeper and your clients are Java applets on the Internet, you need to add a `<PARAM>` tag to the `<APPLET>` tag to provide information about Gatekeeper.

```
<APPLET ... >
<PARAM name="ORBgatekeeperIOR" value="<URL for gatekeeper.ior file>">
</APPLET>
```

For example, if your `gatekeeper.ior` file can be accessed at <http://gateway:1999/gatekeeper.ior>, then your `<PARAM>` line would look like the following:

```
<PARAM name="ORBgatekeeperIOR" value="http://gateway:1999/gatekeeper.ior">
```

Gatekeeper requires you to use a full URL to specify the codebase for your applet. Otherwise, you will get a security exception. For example:

```
<APPLET ... codebase="http://gateway:1999/appletdirectory/classes">
```

For more information about Gatekeeper, see the Gatekeeper Guide from the Visigenic web site.

CONCLUSIONS AND FINAL REMARKS

Oracle Application Server provides an easy management and development environment for distributed applications.

For the management a lot of utilities are included that manage the security, monitoring and all other components in the Oracle Application Server.

For the development of distributed applications this environment is very easy to use. The easy management provides easy ways to add, delete and monitor your applications. It also provides an easy to use GUI to define several parameters that concern the applications. For the development of your applications Oracle Application Server contains several cartridges that contain a lot of functionality for executing the applications and packages for building them.

Although the overall experience on this environment is very good, there are still some bugs to solve. Also the documentation is not always very clear and that takes a lot of the developing time. In future releases Oracle promises a lot of these bugs will disappear and also the documentation shall be updated according to standard English language restrictions, but we can only hope.

LITERATURE

- [ORA01] Oracle Application Server Documentation: Overview of Oracle Application Server 4.0. No. A60115-01. Oracle Corporation, Redwood Shores, Ca 94065 USA.
- [ORA02] Oracle Application Server Documentation: Oracle Application Server Administration Guide. No. A60172-01. Oracle Corporation, Redwood Shores, Ca 94065 USA.
- [ORA03] Oracle Application Server Documentation: Oracle Application Server Building Applications. No. A60121-01. Oracle Corporation, Redwood Shores, Ca 94065 USA.
- [INS01] Reuzel J.G.H.; *Introducing CORBA; A short overview*; INSIEL stage olandesi 1998; ©1998 Trieste.