Developer’s Guide to Clients

Sun™ ONE Application Server

Version 7

817-2173-10
March 2003

Contents

About This Document

This guide describes how to create and run Java(2) Platform, Enterprise Edition^{TM} (J2EE) clients that access Enterprise JavaBeans^{TM} (EJBs) on Sun^{TM} Open Net Environment (Sun ONE) Application Server 7. In addition to describing programming concepts and tasks, this guide offers sample code, implementation tips, reference material, and a glossary.

This preface contains information about the following topics:

Who Should Use This Guide

The intended audience for this guide is the person who develops, assembles, and deploys J2EE applications in a corporate enterprise.

This guide assumes you are familiar with the following topics:

- J2EE specification
- HTML
- Java^{TM} and XML programming
- Java APIs as defined in specifications for EJBs, JSPs, and JDBC

- Software development processes, including debugging and source code control

Using the Documentation

The Sun ONE Application Server manuals are available as online files in Portable Document Format (PDF) and Hypertext Markup Language (HTML) formats, at:

`http://docs.sun.com/`

The following table lists tasks and concepts described in the Sun ONE Application Server manuals. The left column lists the tasks and concepts, and the right column lists the corresponding manuals.

Table 1 Sun ONE Application Server Documentation Roadmap

For information about	See the following
Late-breaking information about the software and the documentation	Release Notes
Supported platforms and environments	Platform Summary
Introduction to the application server, including new features, general installation information, migration details, and architectural overview	*Getting Started Guide*
Installing Sun ONE Application Server and its various components (sample applications, Administration interface, Sun ONE Message Queue).	*Installation Guide*
Creating and implementing J2EE applications that follow the open Java standards model on the Sun ONE Application Server 7. Includes general information about application design, developer tools, security, assembly, deployment, debugging, and creating lifecycle modules.	*Developer's Guide*
Creating and implementing J2EE applications that follow the open Java standards model for web applications on the Sun ONE Application Server 7. Discusses web application programming concepts and tasks, and provides sample code, implementation tips, and reference material.	*Developer's Guide to Web Applications*

Table 1 Sun ONE Application Server Documentation Roadmap

For information about	See the following
Creating and implementing J2EE applications that follow the open Java standards model for EJBs on the Sun ONE Application Server 7. Discusses EJB programming concepts and tasks, and provides sample code, implementation tips, and reference material.	*Developer's Guide to Enterprise JavaBeans Technology*
Creating clients that access J2EE applications on the Sun ONE Application Server.	*Developer's Guide to Clients*
J2EE features such as JDBC, JNDI, JTS, JMS, and JavaMail.	*Developer's Guide to J2EE Features and Services*
Creating custom NSAPI plug-ins	*Developer's Guide to NSAPI*
Performing the following administration tasks: • Using the Administration interface and the command line interface • Configuring server preferences • Using server instances • Monitoring and logging server activity • Configuring the web server plug-in • Configuring the Java Messaging Service • Using J2EE features • Configuring support for CORBA-based clients • Configuring database connectivity • Configuring transaction management • Configuring the web container • Deploying applications • Managing virtual servers	*Administrator's Guide*
Editing server configuration files	*Administrator's Configuration File Reference*
Configuring and administering security for the Sun ONE Application Server 7 operational environment. Includes information on general security, certificates, and SSL/TLS encryption. Web-core-based security is also addressed.	*Administrator's Guide to Security*

Table 1 Sun ONE Application Server Documentation Roadmap

For information about	See the following
Configuring and administering service provider implementation for J2EE CA connectors for the Sun ONE Application Server 7. Includes information about the Administration Tool, DTDs and provides sample XML files.	J2EE CA Service Provider Implementation Administrator's Guide
Migrating your applications to the new Sun ONE Application Server 7 programming model from the Netscape Application Server version 2.1, including a sample migration of an Online Bank application provided with Sun ONE Application Server	*Migration Guide*
Using Sun ONE Message Queue.	The Sun ONE Message Queue documentation at http://docs.sun.com/db/prod/s1.s1msgqu#hic

How This Guide Is Organized

This guide provides instructions for the development, assemble, and the deployment of various types of J2EE clients to Sun ONE Application Server.

- Chapter 1, "Overview of Clients"

  This chapter introduces you to various types of clients that are supported by Sun ONE Application Server.

- Chapter 2, "Using the Application Client Container"

  This chapter describes how to use the Application Client Container to develop and package application clients.

- Chapter 3, "Java-based CORBA Clients"

  This chapter describes the procedure to develop, assemble, and deploy Java-based CORBA clients that do not use the ACC.

- Chapter 4, "C++ Clients"

  This chapter describes the procedure to develop C++ clients using a third-party ORB.

Finally, Glossary and Index are provided.

Reference Information

We recommend the following additional reading:

General J2EE Information:

Core J2EE Patterns: Best Practices and Design Strategies by Deepak Alur, John Crupi, & Dan Malks, Prentice Hall Publishing

Java Security, by Scott Oaks, O'Reilly Publishing

Programming with EJB components:

Enterprise JavaBeans, by Richard Monson-Haefel, O'Reilly Publishing

Java Remote Method Invocation Technology over Internet Inter-ORB Protocol:

`http://java.sun.com/j2se/1.4/docs/guide/rmi-iiop/`

Documentation Conventions

This section describes the types of conventions used throughout this guide:

- General Conventions
- Conventions Referring to Directories

General Conventions

The following general conventions are used in this guide:

- **File and directory paths** are given in UNIX® format (with forward slashes separating directory names). For Windows versions, the directory paths are the same, except that backslashes are used to separate directories.
- **URLs** are given in the format:

  http://*server.domain*/*path*/*file*.html

  In these URLs, *server* is the server name where applications are run; *domain* is your Internet domain name; *path* is the server's directory structure; and *file* is an individual filename. Italic items in URLs are placeholders.
- **Font conventions** include:

  - The `monospace` font is used for sample code and code listings, API and language elements (such as function names and class names), file names, pathnames, directory names, and HTML tags.
  - *Italic* type is used for code variables.
  - *Italic* type is also used for book titles, emphasis, variables and placeholders, and words used in the literal sense.
  - **Bold** type is used as either a paragraph lead-in or to indicate words used in the literal sense.

- **Installation root directories** for most platforms are indicated by *install_dir* in this document. Exceptions are noted in "Conventions Referring to Directories" on page 12.

  By default, the location of *install_dir* on **most** platforms is:

  - Solaris 8 non-package-based Evaluation installations:

    user's home directory / `sun/appserver7`

  - Solaris unbundled, non-evaluation installations:

    `/opt/SUNWappserver7`

  - Windows, all installations:

    `C:\Sun\AppServer7`

  For the platforms listed above, *default_config_dir* and *install_config_dir* are identical to *install_dir*. See "Conventions Referring to Directories" on page 12 for exceptions and additional information.

- **Instance root directories** are indicated by *instance_dir* in this document, which is an abbreviation for the following path:

  default_config_dir`/domains/`*domain*`/`*instance*

- **UNIX-specific descriptions** throughout this manual apply to the Linux operating system as well, except where Linux is specifically mentioned.

Conventions Referring to Directories

By default, when using the Solaris 8 and 9 package-based installation and the Solaris 9 bundled installation, the application server files are spread across several root directories. These directories are described in this section.

- **For Solaris 9, bundled installations**, this guide uses the following document conventions to correspond to the various default installation directories provided:
    - *install_dir* refers to `/usr/appserver/`, which contains the static portion of the installation image. All utilities, executables, and libraries that make up the application server reside in this location.
    - *default_config_dir* refers to `/var/appserver/domains`, which is the default location for any domains that are created.
    - *install_config_dir* refers to `/etc/appserver/config`, which contains installation-wide configuration information such as licenses and the master list of administrative domains configured for this installation.
- **For Solaris 8 and 9 package-based, non-evaluation, unbundled installations**, this guide uses the following document conventions to correspond to the various default installation directories provided:
    - *install_dir* refers to `/opt/SUNWappserver7`, which contains the static portion of the installation image. All utilities, executables, and libraries that make up the application server reside in this location.
    - *default_config_dir* refers to `/var/opt/SUNWappserver7/domains`which is the default location for any domains that are created.
    - *install_config_dir* refers to `/etc/opt/SUNWappserver7/config`, which contains installation-wide configuration information such as licenses and the master list of administrative domains configured for this installation.

Product Support

If you have problems with your system, contact customer support using one of the following mechanisms:

- The online support web site at:

  `http://www.sun.com/supportraining/`

- The telephone dispatch number associated with your maintenance contract

Please have the following information available prior to contacting support. This helps to ensure that our support staff can best assist you in resolving problems:

- Description of the problem, including the situation where the problem occurs and its impact on your operation

- Machine type, operating system version, and product version, including any patches and other software that might be affecting the problem
- Detailed steps on the methods you have used to reproduce the problem
- Any error logs or core dumps

Chapter 1

Overview of Clients

A client can be a simple web browser or an application that runs on the client system. Sun ONE Application Server 7 provides various types of clients, a framework to connect to a back end source, execute the application logic, and return the result to the client.

This chapter introduces different types of clients that Sun ONE Application Server supports. The following topics are discussed in this chapter:

- Introducing Clients
- Types of Clients

Introducing Clients

A client application can be written using Java, C, C++, Visual Basic, or any compatible programming language. A client application sends a request to an application server at a given URL. The server receives the request, processes it, and returns a response. These client programs execute remote procedures and functions in an application server instance.

Sun ONE Application Server is a Java application server and is fully compliant with the J2EE specifications. The important layers of J2EE platform are as follows:

- Client layer - The client layer is where the user accesses the application.
- Presentation layer - The presentation layer is where the user interface is dynamically generated. An application may require the following J2EE components in the presentation layer.
    - Servlets
    - JSPs

  - Static Content

  In addition, an application may require the following non-J2EE, HTTP server-based components in the presentation layer:

  - SHTML
  - CGI

  For more information about the components in the presentation layer, see the *Sun ONE Application Server Developer's Guide to Web Applications*.

- Business logic layer -The business logic layer contains deployed EJB components that encapsulate business rules and other functions in session beans, entity beans, and message-driven beans.

  For more information about components in business logic layer, see the *Sun ONE Application Server Developer's Guide to Enterprise JavaBeans Technology*.

- Data access layer - In the data access layer, JDBC (java database connectivity) is used to connect to databases, make queries, and return query results, and custom connectors work with Sun ONE Application Server to enable communication with legacy EIS systems, such as IBM's CICS.

  Developers are likely to integrate access to the following systems using J2EE CA (J2EE connection architecture):

  - Enterprise resource management system
  - Mainframe systems
  - Third-party security systems

  For more information about JDBC, see the *Sun ONE Application Server Developer's Guide to J2EE Features and Services*.

  For more information about connections, see the *J2EE CA Service Provider Implementation Administration Guide* and the corresponding release notes.

For more information on the J2EE Architecture, see *Sun ONE Application Server Developer's Guide*.

Types of Clients

This section introduces the following types of clients that are supported by Sun ONE Application Server:

- Web Clients

- Web Services Clients
- JMS Clients
- CORBA Clients
- Application Clients

Web Clients

A web client consists of two parts:

- Dynamic web pages containing various types of markup languages such as Hyper Text Markup Language (HTML), Extensible Markup Language (XML), etc, that are generated by web components running in the web server.
- A web browser, which renders the pages received from the server.

A web client is sometimes called a thin client. Thin clients do not query databases, execute complex business rules, or connect to legacy applications. When you use a thin client, heavyweight operations like these are off-loaded to enterprise beans executing on the J2EE server where they can leverage the security, speed, services, and reliability of J2EE server-side technologies.

Web Services Clients

Sun ONE Application Server supports Java-based client applications to send requests to the web service, and receive a response from the web service. To invoke a web service, these clients must construct and send SOAP messages over HTTP.

Sun ONE Application Server supports Apache SOAP version 2.2 and Java^{TM} API for XML-based RPC (JAX RPC) 1.1. Web services support is also built into Sun ONE Studio 4, which is bundled with Sun ONE Application Server.

For information on developing and deploying Web Services clients, see the *Sun ONE Application Server Developer's Guide to Web Services*.

JMS Clients

Java Message Service (JMS) clients are the Java language programs that send and receive messages using the JMS provider. JMS client can be any type of J2EE application component:a web application, an Application Client Container client, an EJB component, and so on. A client accesses a special kind of Enterprise JavaBeans called the message-driven beans (MDB), through JMS by sending messages to the JMS destination.

For more information on using the JMS API to develop JMS clients, see the *Sun ONE Application Server Developer's Guide to J2EE Features and Services.*

CORBA Clients

CORBA clients are the client applications written in any language supported by Common Object Request Broker Architecture (CORBA), including the Java programming language, C++, and C.

CORBA clients are used when a stand-alone program or another application server acts as a client to the EJBs deployed to Sun ONE Application Server. Sun ONE Application Server supports access to EJBs using the Internet Inter-ORB Protocol (IIOP) as specified in the Enterprise JavaBeans Specification, V2.0, and the Enterprise JavaBeans to CORBA Mapping Specification. These clients use Java Naming and Directory Interface (JNDI) to locate EJBs, and use Java^{TM} Remote Method Invocation / Internet Inter-ORB Protocol (RMI / IIOP) to access business methods of remote EJBs.

CORBA clients that do not use the Application Client Container (ACC) have the following limitations:

- JNDI is not supported. However, you can build name translations and do lookups using standard COSNaming binding.
- SSL over RMI / IIOP is not supported.
- Features that are configurable in the `sun-application-client.xml` and `sun-acc.xml` files are not available.

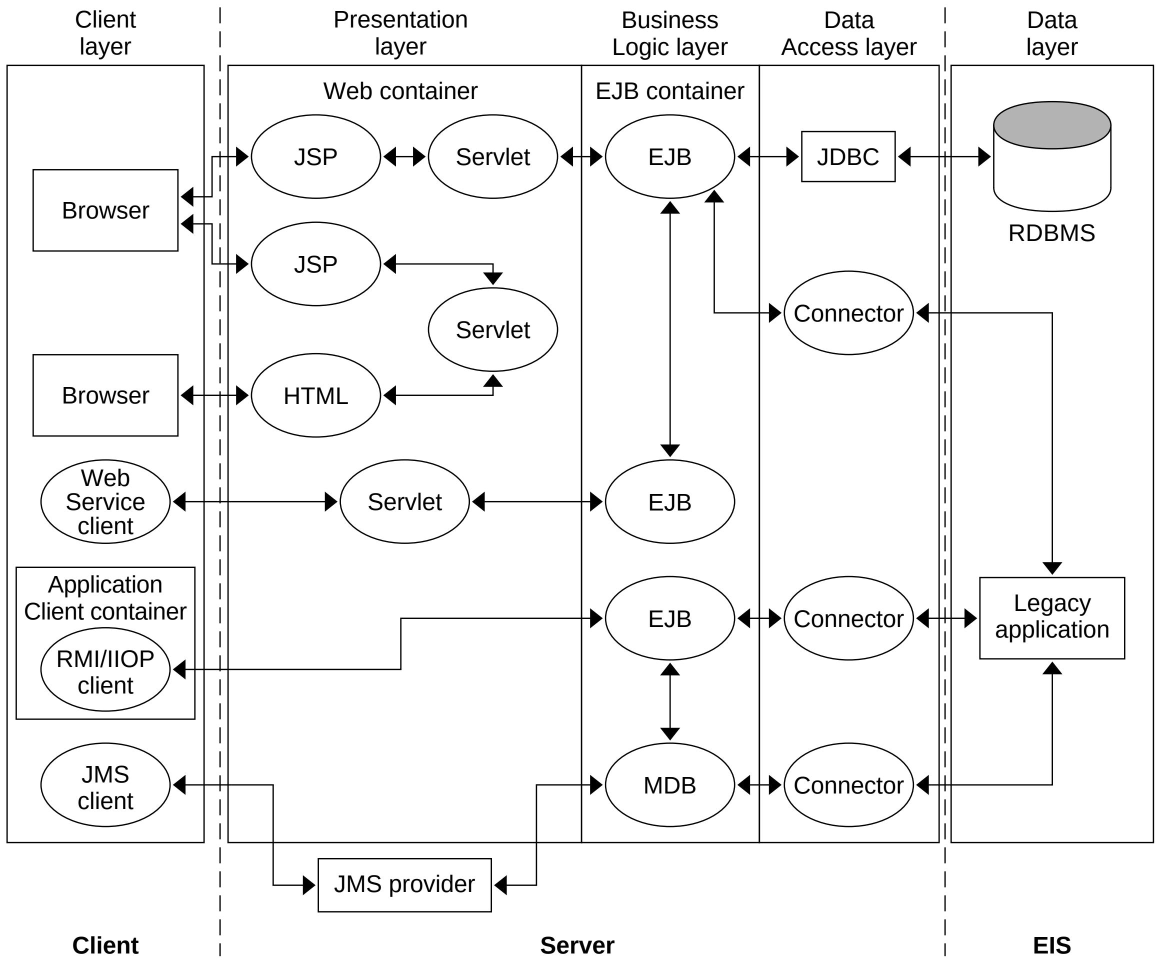
Application Clients

A J2EE application client runs on a client machine and provides a way to handle tasks that require a richer user interface than can be provided by a markup language. Typically, an application client has a GUI created from Swing or Abstract Window Toolkit (AWT) APIs. Alternatively, you can use the command-line interface.

Application clients directly access the EJB components residing in Sun ONE Application Server. However, if application requirements warrant it, a J2EE application client can open an HTTP connection to establish communication with a servlet running in the web server.

The figure, "Client and Sun ONE Application Server Architecture" illustrates client machines running the web browser, web service clients, RMI-IIOP clients, or JMS clients; J2EE server machines running the Sun ONE Application Server; and EIS server machines running databases and legacy applications. JSPs and servlets provide the interface to the client tier, EJBs reside in the business tier, and connectors provide the interface to legacy applications.

Figure 1-1 Client and Sun ONE Application Server Architecture


Client layer
Presentation layer
Business Logic layer
Data Access layer
Data layer
Web container
EJB container
Browser
JSP
Servlet
EJB
JDBC
RDBMS
JSP
Connector
Servlet
Browser
HTML
Web Service client
Servlet
EJB
Application Client container
RMI/IIOP client
EJB
Connector
Legacy application
JMS client
MDB
Connector
JMS provider
Client
Server
EIS

Chapter 2

Using the Application Client Container

This chapter describes how to access the application server using RMI/IIOP protocol, and how to use the Application Client Container (ACC) to develop and package application clients.

This chapter contains the following sections:

Introducing the Application Client Container

The Application Client Container (ACC) includes a set of Java classes, libraries, and other files that are required and distributed along with Java client programs that execute on their own Java Virtual Machine. It manages the execution of the application client components. The ACC provides system services that enable a Java client program to execute. It communicates with Sun ONE Application Server using RMI/IIOP and manages the details of RMI/IIOP communication using the client ORB that is bundled with it. The ACC is specific to the EJB container and is often provided by the same vendor. Compared to other J2EE containers that reside on the server, this container is lightweight.

Application Client Container Features

Security

The ACC is responsible for collecting authentication data such as the username and password from the user. Sends the collected data over RMI/IIOP to the server. The server then processes the authentication data using the configured Java^{TM} Authentication and Authorization Service (JAAS) module. See "Authenticating an Application Client Using the JAAS Module" on page 28.

Authentication techniques are provided by the client container, and are not under the control of the application client. The container integrates with the platform's authentication system. When you execute a client application, it displays a login window and collects authentication data from the user. It also support SSL (Secure Socket Layer)/IIOP if configured and when it is necessary.

Naming

The client container enables the application clients to use Java Naming and Directory Interface (JNDI) to look up EJB components and to reference configurable parameters set at the time of deployment.

Developing Applications Using the ACC

This section describes the procedure to develop, assemble, and deploy client applications using the ACC. This section describes the following topics:

- Creating an Application Client
- Using an Application Client to Invoke an EJB Module
- Invoking an RMI/IIOP-based Client Without Using the ACC
- Authenticating an Application Client Using the JAAS Module
- Packaging an Application Client Using the ACC
- Running an Application Client Using the ACC

Creating an Application Client

A J2EE application client is a program written in the Java programming language. At runtime, the client program executes in a different virtual machine than the J2EE server.

Code examples from the `Converter` sample application illustrate the following steps involved in the development of an application client:

- Locating the Home Interface
- Creating an Enterprise Bean Instance
- Invoking a Business Method

Locating the Home Interface

Use Java Naming and Directory Interface^{TM} (JNDI) to lookup and locate an EJB component's home interface. The following steps describe the procedure to locate an EJB component's home interface.

1. Create an initial naming context.

   ```
   Context initial = new InitialContext();
   Context myEnv = (Context)initial.lookup("java:comp/env");
   Object objref = myEnv.lookup("ejb/RMIConverter");
   ```

   The context interface is part of JNDI. An initial context object, which implements the `Context` interface, provides the starting point for the resolution of names. All naming operations are relative to a context.

2. Retrieve the object bound to the name `rmiConverter`.

   ```
   Object objref = initial.lookup("rmiConverter");
   ```

   The `rmiConverter` name is bound to an enterprise bean reference, a logical name for the home of an enterprise bean component. In this case, the `rmiConverter` name refers to the `ConverterHome` object. The names of enterprise bean components should reside in the `java:com/env/ejb` subcontext.

3. Narrow the reference to a `ConverterHome` object.

   ```
   ConverterHome home =(ConverterHome)
   PortableRemoteObject.narrow(objref, ConverterHome.class);
   ```

Creating an Enterprise Bean Instance

To create the bean instance, the client invokes the `create` method on the `ConverterHome` object. The `create` method returns an object whose type is `Converter`. The remote converter interface defines the business methods of the bean that the client may call and the EJB container instantiates the bean and then invokes the `ConverterBean.ejbCreate` method.

```
Converter currencyConverter = home.create();
```

Invoking a Business Method

To invoke a business method, you first need to invoke a method on the `Converter` object. The EJB container will invoke the corresponding method on the `ConverterEJB` instance that is running on the server. The client invokes the `dollarToYen` business method in the following lines of code:

```
BigDecimal param = new BigDecimal ("100.00");

BigDecimal amount = currencyConverter.dollarToYen(param);
```

Using an Application Client to Invoke an EJB Module

This section describes how an application client can be used to call a stand-alone EJB module, or an EJB module residing in another J2EE application client.

To call an EJB module from an application client, perform the following steps:

1. Define the element `<ejb-ref>` in the `sun-application-client.xml` file.

   For more information on the `sun-application-client.xml` file, see "Sun ONE Application Client Deployment Descriptor" on page 45.

2. Make sure that the JNDI name matches with the JNDI name defined in the EJB module.

3. Deploy the EJB module using the Administration interface. For more information on deploying an EJB module using the Administration interface, see the *Sun ONE Application Server Administrator's Guide*.

   The client JAR file is created at the following location:
   `/application/j2ee-modules/`*ejbmodulename/*`appclient.jar`

4. Distribute your `appclient.jar` file to the location that the client JVM can access.

5. Ensure that the `appclient.jar` file includes the following files:

   - a Java class to access the bean
   - `application-client.xml`
   - `sun-application-client-.xml`
   - `The MANIFEST.MF` file. This file contains the main class, which states the complete package prefix and classname of the Java client.

6. Run the application client to access the EJB component. The following line of code illustrates how to invoke an EJB component using the ACC:

`appclient -client` *jarpath* `-mainclass` *client application main class*`|-name` *name* `-xml` *config_xml_file app-args*

- ❍ `-client` is required and specifies the name and location of the application client jar file.
- ❍ `-mainclass` is optional and specifies the class name, that is located within the `appclient.jar` file whose `main()` method is to be invoked. By default, the class specified in the client jars `Main-class` attribute of the `MANIFEST` file is used.
- ❍ `-name` is optional and specifies the display name that is located within the `appclient.jar`. By default, the display name is specified in the client jar `application-client.xml` file as `display-name` attribute.
- ❍ `-xml`, which specifies the name and location of the ACC configuration xml file, is required if you are not using the default domain and instance. By default, the ACC uses *instance_dir/*`config/sun-acc.xml` for clients running on the application server, or *install_dir//*`lib/appclient/sun-acc.xml` for clients that are packaged using the `package-applclient` script.
- ❍ `app-args` are optional and they represent the arguments passed to the client's `main()` method.

7. To deploy the application client, assemble the application client to create a standard J2EE .ear file and then deploy the application client to Sun ONE Application Server.

Making a Remote Call on the EJB

If you need to access the EJB components that are residing in a remote system other than the system where the application client is being developed, make the following changes into the `sun-acc.xml` fie.

- Define the `<target-server> address` attribute to reference the remote server machine.
- Define the `<target-server> port` attribute to reference the ORB port on the remote server.

This information can be obtained from the `server.xml` file on the remote system.

For more information on `server.xml` file, see the *Sun ONE Application Server Administrator's Configuration File Reference.*

Invoking an RMI/IIOP-based Client Without Using the ACC

You can invoke a J2EE client without using the ACC. When you are creating an application client that does not use the ACC, you need to setup your development environment as follows:

1. Include the following non-java libraries in the client's classpath.

   Windows:

   The following libraries can be found at *install_dir*/`bin`:

   - `cis.dll`
   - `libnspr4.dll`
   - `libplc4.dll`
   - `nss3.dll`
   - `ssl3.dll`

   Solaris:

   The following libraries can be found at *install_dir*/`lib`:

   - `libcis.so`
   - `libnspr4.so`
   - `libplc4.so`
   - `libnss3.so`
   - `libssl3.so`

2. In addition to the non-java libraries, copy the following jar files to the client system and add them to the classpath:

   - `appserv-ext.jar`
   - `appserv-rt.jar`
   - `fscontext.jar`
   - `imq.jar`
   - `imqadmin.jar`
   - `imqutil.jar`

The following steps describe the procedure to create a client:

1. Define the main class as shown in the code illustration below:

```
public static(String[] args) {

            String url = null;

            String jndiname = null;

            boolean acc = true;

}
```

2. If the code sees the `url` and `jndiname` passed in, the `acc` flag is set to false and does the EJB lookup differently than it does if this client code is called by the application client utility without any arguments.

```
if (args.length == 2 ) {

    url = args[0];
    jndiname = args[1];
    acc = false;
    System.out.println("url = "+url);

}
```

3. Obtain the naming initial context and perform the JNDI look up.

```
Properties env = new Properties();

env.put("java.naming.factory.initial",
"com.sun.jndi.cosnaming.CNCtxFactory");

env.put("java.naming.provider.url", url);

initial = new InitialContext(env);

objref = initial.lookup(jndiname);
```

4. Run the client from the command line.

   `java -classpath` *CP ClientApp URL JNDIName*

   where,

   - *CP* is the CLASSPATH which includes the application client jar file and the `appserv-ext.jar`.
   - *ClientApp* refers to the client program.
   - *URL* refers to the application server running on a machine with host name and with an ORB-port.
   - *JNDIName* matches the *JNDIName* specified in the deployment file.

Authenticating an Application Client Using the JAAS Module

Using the JAAS module, you can provide security in your application client code. Create a `LoginModule` that describes the interface implemented by authentication technology providers. `LoginModule`s are plugged in under applications to provide a particular type of authentication.The following steps are involved in creating a `LoginModule`:

1. Write the `LoginModule` interface.

```
public class ClientPasswordLoginModule implements LoginModule{

private static Logger _logger=null;

    static{

       _logger=LogDomains.getLogger(LogDomains.SECURITY_LOGGER);

        }

}

private Subject subject;
private CallbackHandler callbackHandler;
private Map sharedState;
private Map options;
```

   The standard JAAS package required by this class is `javax.security`. The code line below illustrates how you can import the package in your client application:

```
import javax.security.*;
```

2. Initialize the `LoginModule` interface that you just created.

```
public void initialize(Subject subject, CallbackHandler
callbackHandler, Map sharedState, Map options) {

this.subject = subject;
this.callbackHandler = callbackHandler;
this.sharedState = sharedState;
this.options = options;

}
```

   - The parameter `subject`, is the subject to be authenticated.
   - `callbackHandler`, for communicating with the end user which prompts for the username and password.
   - `sharedState`, is the shared `LoginModule` state.

- `options`, the options specified in the configuration file of the `LoginModule`.

3. Use `login()` method to fetch the login information from the client application and authenticate the user.

```
public boolean login() throws LoginException {

if (uname != null) {
   username = new String (uname);
   pswd = System.getProperty (LOGIN_PASSWORD);

}
```

The login information is fetched using the `CallBackHandler`.

```
Callback[] callbacks = new Callback[2];

callbacks[0] = new
NameCallback(localStrings.getLocalString("login.username",
"ClientPasswordModule username: "));

callbacks[1] = new
PasswordCallback(localStrings.getLocalString("login.password",
"ClientPasswordModule password: "), false);

username = ((NameCallback)callbacks[0]).getName();

char[] tmpPassword =
((PasswordCallback)callbacks[1]).getPassword();
```

The `login()` method tries to connect to the server using the login information that is fetched. If the connection is established, the method returns the value true.

4. Use `commit()` method to set the subject in the session to the username that is verified by the login method. If the commit method returns a value true, then this method associates `PrincipalImpl` with the `subject` located in the `LoginModule`. If this LoginModule's own authentication attempt is failed, then this method removes any state that was originally saved.

```
public boolean commit() throws LoginException {
if (succeeded == false) {
return false;
} else {
// add a Principal (authenticated identity)to the Subject
// assume the user we authenticated is the PrincipalImpl
userPrincipal = new PrincipalImpl(username);
```

5. Use `logout()` method to remove the privilege settings associated with the roles of the subject.

```
public boolean logout() throws LoginException {

subject.getPrincipals().remove(userPrincipal);
succeeded = false;
succeeded = commitSucceeded;
username = null;
if (password != null) {
   for (int i = 0; i < password.length; i++)
      password[i] = ' ';
      password = null;
}
userPrincipal = null;
return true;
}
```

6. Edit the `sun-acc.xml` deployment descriptor to configure JAAS authentication for the client. See "auth-realm" on page 57.

7. Integrate the `LoginModule` with the application server.

   Edit the deployment descriptor to make the following changes:

   - Configure the server with a realm that uses a specific `LoginModule` for security authentication.

   - Map the application realm and roles to the realm and roles defined by the `LoginModule`.

8. Assemble the application client. See "Packaging an Application Client Using the ACC" on page 37.

Sample Code

The sample code of `ClinetLoginPasswordModule` is given below:

```
package com.sun.enterprise.security.auth.login;

import java.util.*;
import java.io.IOException;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import javax.security.auth.spi.*;
import com.sun.enterprise.security.auth.login.PasswordCredential;
import com.sun.enterprise.security.PrincipalImpl;
import com.sun.enterprise.security.auth.LoginContextDriver;
import com.sun.enterprise.util.LocalStringManagerImpl;
import java.util.logging.*;
import com.sun.logging.*;
```

```
public class ClientPasswordLoginModule implements LoginModule {

private static Logger _logger=null;
   static{
      _logger=LogDomains.getLogger(LogDomains.SECURITY_LOGGER);
   }

private static final String DEFAULT_REALMNAME = "default";
private static LocalStringManagerImpl localStrings =
   new LocalStringManagerImpl(ClientPasswordLoginModule.class);

// initial state

private Subject subject;
private CallbackHandler callbackHandler;
private Map sharedState;
private Map options;

private boolean debug = com.iplanet.ias.util.logging.Debug.enabled;

// the authentication status

private boolean succeeded = false;
private boolean commitSucceeded = false;

// username and password

private String username;
private char[] password;

private final  PasswordCredential passwordCredential=null;

// testUser's PrincipalImpl

private PrincipalImpl userPrincipal;
public static String LOGIN_NAME = "j2eelogin.name";
public static String LOGIN_PASSWORD = "j2eelogin.password";

public void initialize(Subject subject, CallbackHandler
callbackHandler, Map sharedState, Map options) {

      this.subject = subject;
      this.callbackHandler = callbackHandler;
      this.sharedState = sharedState;
      this.options = options;

// initialize any configured options

      debug =
"true".equalsIgnoreCase((String)options.get("debug"));

}
```

```
/* Authenticate the user by prompting for a username and password.
@return true in all cases since this <code>LoginModule</code> should
not be ignored.*/

/* @exception FailedLoginException if the authentication fails.
@exception LoginException if this <code>LoginModule</code> is unable
to perform the authentication.*/

public boolean login() throws LoginException {

// prompt for a username and password

if (callbackHandler == null){

   String failure =
localStrings.getLocalString("login.nocallback","Error: no
CallbackHandler available to garner authentication information from
the user");

   throw new LoginException(failure);
}

String uname = System.getProperty (LOGIN_NAME);
String pswd;

if (uname != null) {

   username = new String (uname);
   pswd = System.getProperty (LOGIN_PASSWORD);
   char[] dest;
   if (pswd == null){
      dest = new char[0];
      password = new char[0];
   } else {
      int length = pswd.length();
      dest = new char[length];
      pswd.getChars(0, length, dest, 0 );
      password = new char[length];
   }
   System.arraycopy (dest, 0, password, 0, dest.length);
   } else{
      Callback[] callbacks = new Callback[2];
      callbacks[0] = new
NameCallback(localStrings.getLocalString("login.username",
"ClientPasswordModule username: "));
      callbacks[1] = new
PasswordCallback(localStrings.getLocalString("login.password",
"ClientPasswordModule password: "), false);
```

```
    try {
        callbackHandler.handle(callbacks);
        username = ((NameCallback)callbacks[0]).getName();
        if(username == null){
            String fail =
localStrings.getLocalString("login.nousername", "No user
specified");
            throw new LoginException(fail);
        }

        char[] tmpPassword =
((PasswordCallback)callbacks[1]).getPassword();

        if (tmpPassword == null) {
    // treat a NULL password as an empty password
        tmpPassword = new char[0];
        }
        password = new char[tmpPassword.length];
        System.arraycopy(tmpPassword, 0,
        password, 0, tmpPassword.length);
        ((PasswordCallback)callbacks[1]).clearPassword();
        } catch (java.io.IOException ioe) {
        throw new LoginException(ioe.toString());
        } catch (UnsupportedCallbackException uce) {
String nocallback =
localStrings.getLocalString("login.callback","Error: Callback not
available to garner authentication information from
user(CallbackName):" );
throw new LoginException(nocallback + uce.getCallback().toString());

}

}

// print debugging information
if (debug) {

for (int i = 0; i < password.length; i++){
//System.out.print(password[i]);
}
//System.out.println();
}

// by default -  the client side login module will always say
// that the login successful. The actual login will take place
// on the server side.
if (debug)
```

```
{
_logger.log(Level.FINE,"[ClientPasswordLoginModule] "
+"authentication succeeded");
succeeded = true;
return true;

}

public boolean commit() throws LoginException {
if (succeeded == false) {
   return false;
   } else {
   // add a Principal (authenticated identity)to the Subject
   // assume the user we authenticated is the PrincipalImpl
      userPrincipal = new PrincipalImpl(username);
      if (!subject.getPrincipals().contains(userPrincipal))
         subject.getPrincipals().add(userPrincipal);
         if (debug) {
         _logger.log(Level.FINE,"[ClientPasswordLoginModule] "
+"added PrincipalImpl to Subject");
   }

PasswordCredential pc = new PasswordCredential(username, new
String(password), realm);
if(!subject.getPrivateCredentials().contains(pc))subject.getPrivate
Credentials().add(pc);

username = null;
for (int i = 0; i < password.length; i++){
password[i] = ' ';
password = null;
commitSucceeded = true;
return true;
}
}

public boolean abort() throws LoginException {
if (succeeded == false) {
return false;
} else if (succeeded == true && commitSucceeded == false) {
// login succeeded but overall authentication failed
   succeeded = false;
   username = null;
   if (password != null) {
      for (int i = 0; i < password.length; i++)
         password[i] = ' ';
```

```
            password = en das ull;
        }
        userPrincipal = null;
        } else {

        // overall authentication succeeded and commit succeeded,
        // but someone else’s commit failed
        logout();
        }
        return true;
        }

public boolean logout() throws LoginException {

subject.getPrincipals().remove(userPrincipal);
succeeded = false;
succeeded = commitSucceeded;
username = null;
if (password != null) {
    for (int i = 0; i < password.length; i++)
        password[i] = ’ ’;
        password = null;
}
userPrincipal = null;
return true;
}

}
```

Authenticating an RMI/IIOP Client Without Using the ACC

This section describes the necessary steps and procedure to create an RMI/IIOP client that accesses secure EJBs from outside the ACC.

First, you must setup your client development environment using the following steps:

1. Include the following jar files in the classpath on the client side:
   - `appserv-rt.jar` - available at *install_dir*/`lib`
   - `appserv-ext.jar` - available at *install_dir*/`lib`
   - The client jar that is generated after you deploy your application

2. Set `org.omg.CORBA.ORBInitialHost` to the host on which the IIOP listener is running.

   ```
   env.setProperty("org.omg.CORBA.ORBInitialHost", “name service
   hostname");
   ```

3. Set `org.omg.CORBA.ORBInitialPort` to the port on which the IIOP listener is listening (usually 3700).

   ```
   env.setProperty("org.omg.CORBA.ORBInitialPort", "3700");
   ```

4. Set `java.security.auth.login.config` to *install_dir*`/lib/appclient/appclientlogin.conf`

NOTE Do not set `java.naming.factory.initial`. The default JNDI provider will by default be picked from the above set classpath.

Next step is to create the client application. The following steps describe the procedure:

1. Obtain a username and a password. To obtain a username and a password, you can either write your own JAAS login callback handler or use the standard one provided with Sun ONE Application Server (`com.sun.enterprise.security.auth.login.LoginCallbackHandler`).

   The following code line illustrates the use of standard handler using GUI-based authentication:

   ```
   LoginCallbackHandler handler = new LoginCallbackHandler(true);
   ```

   The following code line illustrates the use of standard handler using the text authentication:

   ```
   LoginCallbackHandler handler = new LoginCallbackHandler(false);
   ```

   The following code line is an example code for writing your own login callback handler:

   ```
   import javax.security.auth.callback.CallbackHandler;
   import
   javax.security.auth.callback.UnsupportedCallbackException;
   import javax.security.auth.callback.Callback;
   import javax.security.auth.callback.NameCallback;
   import javax.security.auth.callback.PasswordCallback;

   public class LoginCallbackHandler implements CallbackHandler {
   ```

```
private String username = "j2ee";
private String password = "j2ee";

public void handle(Callback[] callbacks) throws
UnsupportedCallbackException {
try {
   for (int i = 0; i <callbacks.length; i++) {
   if (callbacks[i] instanceof NameCallback) {
      NameCallback nc = (NameCallback)callbacks[i];
      nc.setName(username);
      } else if(callbacks[i] instanceof PasswordCallback) {
         PasswordCallback pc = (PasswordCallback)callbacks[i];
         pc.setPassword(password.toCharArray());
      }
   }

   } catch (Exception ex) {
      ex.printStackTrace();
   }
   }
}
```

2. Pass an instance of your handler to the security infrastructure using the following call:

   ```
   LoginContextDriver.doClientLogin(AppContainer.USERNAME_PASSWORD,
   handler);
   ```

   The following two imports are required for the above call:

   ```
   import com.sun.enterprise.appclient.AppContainer;
   import com.sun.enterprise.security.auth.LoginContextDriver;
   ```

Packaging an Application Client Using the ACC

After installing Sun ONE Application Server, the ACC can be run by executing the `appclient` script located in the *install_dir*/`bin` directory. The script `package-appclient` that is located in the same directory, is used to package a client application into a single `appclient.jar` file. Packaging an application client involves the following main steps:

- Editing the Configuration File
- Editing the appclient Script
- Editing the sun-acc.xml File
- Setting Security Options

- Using the package-appclient Script

Editing the Configuration File

Modify the environment variables in `asenv.conf` file located in the *default-config_dir* directory as shown below:

- `$AS_INSTALL` to reference the location where the package was un-jared plus `/appclient`. For example: `$AS_INSTALL=`*/mylocation/*`appclient`.
- `$AS_NSS` to reference the location of the nss libs.

  For example:

  UNIX:

  `$AS_NSS=`*/mylocation/*`appclient/lib`

  WINDOWS:

  `%AS_NSS%=`*\mylocation*`\appclient\bin`
- `$AS_JAVA` to reference the location where you have installed the JDK.
- `$AS_ACC_CONFIG` to reference the configuration xml (`sun-acc.xml`). The `sun-acc.xml` is located at *install_dir/*`config`.
- `$AS_IMQ_LIB` to reference the imq home. It should be: *install_dir/*`imq/lib`.

Editing the appclient Script

Modify the `appclient` script file as follows:

UNIX:

Change `$CONFIG_HOME/asenv.conf` to *your_ACC_dir/*`config/asenv.conf`.

Windows:

Change `%CONFIG_HOME%\config\asenv.bat` to *your_ACC_dir*`\config\asenv.bat`

Editing the sun-acc.xml File

Modify `sun-acc.xml` file to set the following attributes:

- Ensure that the `DOCTYPE` references `%%%SERVER_ROOT%%%/lib/dtds` to *your_ACC_dir/*`lib/dtds`.
- Ensure that the `<target-server> address` attribute references the remote server machine.

- Ensure that the `<target-server> port` attribute references the ORB port on the remote server.
- If you want to log the messages in a file, specify a file name for the `<log-service> file` attribute. You can also set the log level.

  For example,

  ```
  <?xml version="1.0" encoding="UTF-8"?>

  <!DOCTYPE client-container SYSTEM "file:{Your installed server
  root}/lib/dtds/sun-application-client-container_1_0.dtd ">

  <client-container>

     <target-server name="qasol-e1" address="qasol-e1"
  port="3700">

     <log-service file=" " level="WARNING"/>

  </client-container>
  ```

For more information on the `sun-acc.xml` file, see "Application Client Container Configuration File" on page 50.

Setting Security Options

You can run the application client using SSL with certificate authentication. In order to set the security options, modify the `sun-acc.xml` file as shown in the code illustration below. For more information on the `sun-acc.xml` file, see the "Application Client Container Configuration File" on page 50.

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE client-container SYSTEM

"file:////export3/sun/appserver7/appserv/lib/dtds/sun-applicatio
n-client-container_1_0.dtd">

<client-container>

<target-server name="qasol-e1" address="qasol-e1" port="3700">

<security>

<ssl cert-nickname="cts" ssl2-enabled="false"
ssl2-ciphers="-rc4,-rc4export,-rc2,-rc2export,-des,-desede3"

ssl3-enabled="true"

ssl3-tls-ciphers="+rsa_rc4_128_md5,-rsa_rc4_40_md5,+rsa3_des_sha
,+rsa_des_sha,-rsa_rc2_40_md5,-rsa_null_md5,-rsa_des_56_sha,-rsa
_rc4_56_sha"

tls-enabled="true" tls-rollback-enabled="true"/>
```

```
<cert-db path="/export3/ctsdata/ctscertdb" password="changeit"/>
</security>
</target-server>
<client-credential user-name="j2ee" password="j2ee"/>
<log-service file="" level="WARNING"/>
</client-container>
```

Using the package-appclient Script

The following steps describe the procedure to use the `package-appclient` script that is bundled with Sun ONE Application Server:

1. Under *install_dir*/`bin` directory, run the `package-appclient` script. This creates an `appclient.jar` file and stores it under *install_dir*/`lib/appclient/` directory.

NOTE The `appclient.jar` file provides an application client container package targeted at remote hosts and does not contain a server installation. You can run this file from a remote machine with the same operating system as where it is created. That is, `appclient.jar` created on a Solaris platform will not function on Windows.

2. Copy the *install_dir*/`lib/appclient/appclient.jar` file to the desired location. The `appclient.jar` file contains the following files:
   - `appclient/bin` - contains the `appclient` script which you use to launch the ACC.
   - `appclient/lib` - contains the JAR and runtime shared library files.
   - `appclient/lib/appclient` - contains the following files:
     - `sun-acc.xml` - the ACC configuration file.
     - `client.policy` file- the security manager policy file for the ACC.
     - `appclientlogin.conf` file - the login configuration file.
     - `client.jar` file - is created during the deployment of the client application.

- ❍ `appclient/lib/dtds` - contains `sun-application_client-contianer_1_3-0.dtd` which is the DTD corresponding to `sun-acc.xml`.

client.policy

`client.policy` file is the J2SE policy file used by the application client. Each application client has a `client.policy` file. The default policy file limits the permissions of J2EE deployed application clients to the minimal set of permissions required for these applications to operate correctly. If you develop an application client that requires more than this default set of permissions, you can edit the `client.policy` file to add the custom permissions that your applications need. You can use the J2SE standard policy tool or any text editor to edit this file. For more information on using the J2SE policy tool, visit the following URL:

```
http://java.sun.com/docs/books/tutorial/security1.2/tour2/
index.html
```

For more information about the permissions you can set in the `client.policy` file, visit the following URL:

`http://java.sun.com/j2se/1.4/docs/guide/security/permissions.html`

Running an Application Client Using the ACC

To run a client application that is packaged in an application jar file, you first need to launch the ACC. You can launch the application client container using `appclient` script.

`appclient -client` *client_application_jar* `[-mainclass` *client_application_main_class_name*`|-name` *display_name*`][-xml sun-acc.xml]` `[-textauth] [-user` *user_name*`] [-password` *password*`]`

- `-client`: Specifies the name and location of the client application jar file. This is a required parameter.
- `-mainclass`: Specifies the class name that is located within the client jar whose `main()` method is to be invoked. By default, uses the class specified in the `client jar`. This is optional.

NOTE The class name must be the full name. For example, `com.sun.test.AppClient`

- `-name`: Specifies the display name that is located in the application client jar file. By default, the display name is specified in the client jar `application-client.xml` file which is identified by the `display-name` attribute. This is optional.

NOTE `-mainclass`, `-name` are optional for a single client application. For multiple client applications use either the `-classname` option or the `-name` option.

- `-xml`: is used to specify the name and location of the client configuration xml file. If you do not specify this option, ACC will use the default one from `appclient` script identified by `$AS_ACC_CONFIG` that references to the default instance. For Solaris bundle, this option is required.
- `-textauth`: is optional for user to specify authentication using the text format.

The following example shows how to run the sample application client, `rmiConverter`:

```
appclient -client rmi-simpleClient.jar
```

Sample Client Application

You can find the sample client application that demonstrates the working of an RMI/IIOP client that uses an application client container at the following location:

install_dir/`samples/rmi-iiop/simple`

Application Client Deployment Descriptors

Deployment descriptors are the XML files used to configure the runtime properties of a module or application. The J2EE Specification defines the format of these descriptors. You can view and edit the deployment descriptors using a text editor at any time during the development process.

Sun ONE Application Server application clients require three deployment descriptors files:

- A J2EE standard file (`application.client.xml`), described in the J2EE Specification.

- An optional Sun ONE Application Server specific client deployment descriptor file (`sun-application-client.xml`), described in this section.
- An optional Sun ONE Application Server specific Application Client Container Configuration file (`sun-acc.xml`), described in this section.

This section presents the following topics:

- Format of Deployment Descriptors
- J2EE Application Client Deployment Descriptor
- Sun ONE Application Client Deployment Descriptor
- Application Client Container Configuration File

Format of Deployment Descriptors

A deployment descriptor file defines the elements that an XML file can contain and the subelements and attributes these elements can have. The `sun-application-client-1_3-0.dtd` file defines the format of the `sun-application-client.xml` file. The `sun-application-client-container-1_0.dtd` file defines the format of `sun-acc.xml` file. These DTD files are located in the *install_dir*`/lib/dtds` directory.

NOTE	Do not edit the DTD files. Their contents change only with new versions of Sun ONE Application Server.

For general information about DTD files and XML, see the XML specification at:

`http://www.w3.org/TR/REC-xml`

Each element defined in a DTD file (which may be present in the corresponding XML file) can contain the following:

- Subelements
- Data
- Attributes

Subelements

An element can contain other elements. For example, the following code defines the `client-container` element.

```
<!ELEMENT
client-container(target-server,auth-realm?,client-credential?,
log-service?,property*))>
```

The `ELEMENT` tag specifies that a `client-container` element can contain `target-server, auth-realm, client-credential, log-service, property` subelements.

The following table shows how optional suffix characters of subelements determine the requirement rules, or number of allowed occurrences, for the subelements. The left column lists the subelement ending character, and the right column lists the corresponding requirement rule:

Table 2-1 requirement rules for subelement suffixes

Subelement Ending Character	Requirement
*	Can contain *zero or more* of this subelement.
?	Can contain *zero or one* of this subelement.
+	Must contain *one or more* of this subelement.
(none)	Must contain *only one* of this subelement.

If an element cannot contain other elements, you see `EMPTY` or `(#PCDATA)` instead of a list of element names in parentheses.

Data

Some elements contain data instead of subelements. These elements have definitions of the following format:

`<!ELEMENT` *element-name* `(#PCDATA)>`

For example:

```
<!ELEMENT credential (#PCDATA)>
```

Attributes

Elements that have `ATTLIST` tags contain attributes (name-value pairs). Attributes have definitions of the following format:

`<!ATTLIST` *element attribute type default attribute type default ...>*

For example:

`<!ATTLIST client-container` *user-name* `CDATA #REQUIRED`

password `CDATA #REQUIRED`

realm `CDATA #IMPLIED>`

A `client-container` element can contain `user-name`, `password`, and `realm` attributes.

The `#REQUIRED` label means that a value must be supplied.

The `#IMPLIED` label means that the attribute is optional, and that Sun ONE Application Server generates a default value. Wherever possible, explicit defaults for optional attributes (such as `"true"`) are listed.

Attribute declarations specify the type of the attribute. For example, `CDATA` means character data, and `%boolean` is a predefined enumeration.

J2EE Application Client Deployment Descriptor

Application clients are packaged in JAR format files with a .jar extension and include a deployment descriptor similar to other J2EE application components. The deployment descriptor describes the enterprise beans and external resources referenced by the application. As with other J2EE application components, you need to configure access to resources at the time of deployment, assign names for enterprise beans and resources, etc.The deployment descriptor is standardized by the J2EE 1.3 specification.

Sun ONE Application Client Deployment Descriptor

The `sun-application-client.xml` is the deployment descriptor for the application clients. The easiest way to create a `sun-application-client.xml` file is to deploy the application client. For more information on deploying a client using the Administration interface, see the *Sun ONE Application Server Developer's Guide*.

Elements in sun-application-client.xml file

Elements in the `sun-application-client.xml` file are as follows:

- `sun-application-client`
- `resource-ref`
- `ejb-ref`

- `resource-env-ref`
- `res-ref-name`
- `resource-env-ref-name`
- `default-resource-principal`
- `name`
- `password`
- `ejb-ref-name`
- `jndi-name`

NOTE Subelements must be defined in the order in which they are listed under each **Subelements** heading unless otherwise noted.

Attributes

Elements can contain attributes (name, value pairs). Attributes are defined in attributes lists using the ATTLIST tag.

None of the elements in the `sun-application-client.xml` file contain attributes.

sun-application-client

This is the root element describing all the runtime bindings of a single application client.

Subelements

The following table describes subelements for the `sun-application-client` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

Table 2-2 `sun-application-client` subelements

Element	Required	Description
`resource-ref`	zero or more	Maps the absolute JNDI name to the `resource-ref` in the corresponding J2EE XML file.
`ejb-ref`	zero or more	Maps the absolute JNDI name to the `ejb-ref` in the corresponding J2EE XML file.

Table 2-2 `sun-application-client` subelements *(Continued)*

Element	Required	Description
resource-env-ref	zero or more	Maps the absolute JNDI name to the `resource-env-ref` in the corresponding J2EE XML file.

resource-ref

Maps the absolute JNDI name to the `resource-ref` element in the corresponding J2EE XML file.

Subelements

The following table describes subelements for the `resource-ref` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

Table 2-3 `resource-ref` subelements

Element	Required	Description
`res-ref-name`	only one	Specifies the `res-ref-name` in the corresponding J2EE `application-client.xml` file.
`jndi-name`	only one	Specifies the absolute jndi name of a resource.
default-resource-principal	zero or more	Specifies the default principal (user) that the container uses to access a resource.

res-ref-name

Specifies the `res-ref-name` in the corresponding J2EE `application-client.xml` file `resource-ref` entry.

Subelements

none

default-resource-principal

Specifies the default principal (user) that the container uses to access a resource.

If this element is used in conjunction with a JMS Connection Factory resource, the `name` and `password` subelements must be valid entries in Sun ONE Message Queue's broker user repository. See the "Security Management" chapter in the *Sun ONE Message Queue Administrator's Guide* for details.

Subelements

The following table describes subelements for the `default-resource-principal` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

Table 2-4 `default-resource-principal` subelements

Element	Required	Description
`name`	only one	Specifies the name of the principal.
`password`	only one	Specifies the password for the principal.

name

Contains data that specifies the name of the principal.

Subelement

none

password

Contains data that specifies the password for the principal.

Subelement

none

ejb-ref

Maps the `ejb-ref-name` in the corresponding J2EE `ejb-jar.xml` file `ejb-ref` entry to the absolute `jndi-name` of a resource.

Subelements

The following table describes subelements for the `ejb-ref` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

Table 2-5 `ejb-ref` subelements

Element	Required	Description
`ejb-ref-name`	only one	Specifies the name of a ejb reference in the corresponding J2EE appclient.xml file.
`jndi-name`	only one	Specifies the absolute jndi name of a resource.

ejb-ref-name

Specifies the `ejb-ref-name` in the corresponding J2EE `ejb-ref.xml` file `ejb-ref` entry. This element locates the name of the ejb reference in the application.

Subelement

none

resource-env-ref

Specifies the name of a resource env reference.

Subelements

The following table describes subelements for the `resource-env-ref` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

Table 2-6 `resource-env-ref` subelements

Element	Required	Description
`resource-env-ref-name`	only one	Specifies the `res-ref-name` in the corresponding J2EE `application-client.xml` file `resource-env-ref` entry.
`default-resource-principal`	only one	Specifies the default principal (user) that the container uses to access a resource.
jndi-name	only one	Specifies the jndi-name of the associated entity.

resource-env-ref-name

Specifies the `res-ref-name` in the corresponding J2EE `application-client.xml` file `resource-env-ref` entry.

Subelements

none

jndi-name

Contains data that specifies the absolute `jndi-name` of a URL resource or a resource in the `application-client.xml` file.

Subelement

none

Application Client Container Configuration File

The `sun-acc.xml` file tracks changes in Sun ONE Application Client Container configuration.

Elements in the sun-acc.xml File

Elements in the `sun-acc.xml` file are as follows:

- `client-container`
- target-server
- description
- client-credential
- log-service
- security
- ssl
- cert-db
- auth-realm
- property

client-container

Defines Sun ONE Application Server specific configuration for the ACC. This is the root element; there can only be one `client-container` element in a `sun-acc.xml` file.

Subelements

The following table describes subelements for the `client-container` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

Table 2-7 `client-container` subelements

Element	Required	Description
`target-server`	zero or more	Specifies the IIOP listener configuration of the target server.
`auth-realm`	only one	Specifies the optional configuration for JAAS authentication realm.
client-credential	only one	Specifies the default client credential that will be sent to the server.
log-service	only one	Specifies the default log file and the severity level of the message.
property	zero or more	Specifies a property which has a name and a value.

Attributes

The following table describes attributes for the `client-container` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

Table 2-8 `client-container` attributes

Attribute	Default Value	Description
`sendPassword`	none	Specifies whether client authentication credentials should be sent to the server. Without authentication credential all access to protected EJBs will result in exceptions.

target-server

Defines the IIOP listener configuration of the target server.

Subelements

The following table describes subelements for the `target-server` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

Table 2-9 `target-server` subelements

Element	Required	Description
`description`	zero or more	Specifies the description of the target server.
`security`	zero or more	Specifies the security configuration for the IIOP/SSL communication with the target server.

Attributes

The following table describes attributes for the `target-server` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

Table 2-10 `target-server` attributes

Attribute	Default Value	Description
`name`	none	Specifies the name of the application server instance accessed by the client container.
`address`	none	Specifies the host name or IP address (resolvable by DNS) of the ORB.
`port`	3700	Specifies port number of the ORB. For the new server instance, you need to assign a different port number other than 3700. You can change the port number in the Administration Interface. See the *Sun ONE Application Server Administrator's Guide* for more information.

description

Contains data that specifies a text description of the containing element.

Subelement

none

Attributes

none

client-credential

Default client credentials that will be sent to the server. If this element is present, then it will be automatically sent to the server, without prompting the user for username and password on the client side.

Subelements

The following table describes subelements for the `client-credential` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

Table 2-11 `client-credential` subelement

Element	Required	Description
`property`	zero or more	Specifies a property which has a name and a value.

Attributes

The following table describes attributes for the `client-credential` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

Table 2-12 `client-credential` attributes

Attribute	Default Value	Description
`user-name`	none	The user name used to authenticate the Application client container.
`password`	none	The password used to authenticate the Application client container.
`realm`	none	The realm (specified by name) where credentials are to be resolved.

log-service

Specifies configuration settings for the log file.

Subelements

The following table describes subelements for the `log-service` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

Table 2-13 `log-service` subelement

Element	Required	Description
`property`	zero or more	Specifies a property which has a name and a value.

Attributes

The following table describes attributes for the `log-service` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

Table 2-14 `log-service` attributes

Attribute	Default Value	Description
`log-file`	client.log	Specifies the name of the file where the application client container logging information will be stored. By default, the log file will be located at *your_Acc_dir*/`logs/client.log.`
`level`	none	Sets the base level of severity. Messages at or above this setting get logged into the log file.

security

Defines SSL security configuration for IIOP / SSL communication with the target server.

Subelements

The following table describes subelements for the `security` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

Table 2-15 `security` subelement

Element	Required	Description
ssl	zero or more	Specifies the SSL processing parameters.
cert-db	zero or more	Specifies the location and authentication to read the certification database.

Attributes

none

ssl

Defines SSL processing parameters.

Subelements

none

Attributes

The following table describes attributes for the `SSL` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

Table 2-16 `ssl attributes`

Attribute	Default Value	Description
`cert-nickname`	none	The nickname of the server certificate in the certificate database or the PKCS#11 token.In the certificate, the name format is *tokenname:nickname*. Including the *tokenname*: part of the name in this attribute is optional.
`ssl2-enabled`	none	(Optional) Determines whether SSL2 is enabled.
`ssl3-enabled`	none	(Optional) Determines whether SSL3 is enabled.
`ssl2-ciphers`	none	(Optional) A space-separated list of the SSL2 ciphers used with the prefix + to enable or - to disable. For example, `+rc4`. Allowed values are `rc4, rc4export, rc2, rc2export, idea, des, desede3`.

Table 2-16 `ssl attributes`

Attribute	Default Value	Description
`ssl3-tls-ciphers`	none	(Optional) A space-separated list of the SSL3 ciphers used, with the prefix + to enable or - to disable, for example +`rsa_des_sha`. Allowed SSL3 values are `rsa_rc4_128_md5, rsa3_des_sha, rsa_des_sha, rsa_rc4_40_md5, rsa_rc2_40_md5, rsa_null_md5. Allowed TLS values are rsa_des_56_sha, rsa_rc4_56_sha`.
`tls-enabled`	none	Determines whether TLS is enabled.
`tls-rollback-enabled`	none	Determines whether TLS rollback is enabled.TLS rollback should be enabled for MicroSoft Internet Explorer 5.0 and 5.5.
`client-auth-enabled`	none	Determines whether SSL3 client authentication is performed on every request, independent of ACL-based access control.

If both SSL2 and SSL3 are enabled, the server tries SSL3 encryption first. If that fails, the server tries SSL2 encryption. If both SSL2 and SSL3 are enabled for a virtual server, the server tries SSL3 encryption first. If that fails, the server tries SSL2 encryption.

cert-db

Location and password to read the certificate database. SunONE Application Server provides utilities with which a certificate database can be created. `certutil`, distributed as part of NSS can also be used to create certificate database.

Subelement

none

Attributes

The following table describes attributes for the `cert-db` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

Table 2-17 `cert-db` attributes

Attribute	Default Value	Description
`cert-db-path`	none	Specifies the absolute path of the certificate database (cert7.db).
`cert-db-password`	none	Specifies the password to access the certificate database.

auth-realm

JAAS is available on the ACC. Defines the optional configuration for JAAS authentication realm.

Authentication realms require provider-specific properties, which vary depending on what a particular implementation needs.

For more information about how to define realms, see the *Sun ONE Application Server Developer's Guide*.

Here is an example of the default file realm:

```
<auth-realm name="file"

classname="com.iplanet.ias.security.auth.realm.file.FileRealm">

<property name="file" value="instance_dir/config/keyfile"/>

<property name="jaas-context" value="fileRealm"/>

</auth-realm>
```

Which properties an `auth-realm` element uses depends on the value of the `auth-realm` element's name attribute. The file `realm` uses `file` and `jaas-context` properties. Other realms use different properties.

Subelements

The following table describes subelements for the `auth-realm` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

Table 2-18 `auth-realm` subelement

Element	Required	Description
`property`	zero or more	Specifies a property which has a name and a value.

Attributes

The following table describes attributes for the `auth-realm` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

Table 2-19 `auth-realm` attributes

Attribute	Default Value	Description
`auth-realm-name`	none	Defines the name of this realm.
`classname`	none	Defines the Java class which implements this realm.

property

Specifies a property, which has a name and a value.

Subelement

none

Attributes

The following table describes attributes for the `property` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

Table 2-20 `property` attributes

Attribute	Default Value	Description
`name`	none	Specifies the name of the property.
`value`	none	Specifies the value of the property.

Chapter 3

Java-based CORBA Clients

This chapter describes how to develop and deploy CORBA clients that use RMI/IIOP protocol.

This chapter contains the following sections:

- CORBA Client Scenarios
- Developing Java-based CORBA Clients
- Third Party ORB Support

CORBA Client Scenarios

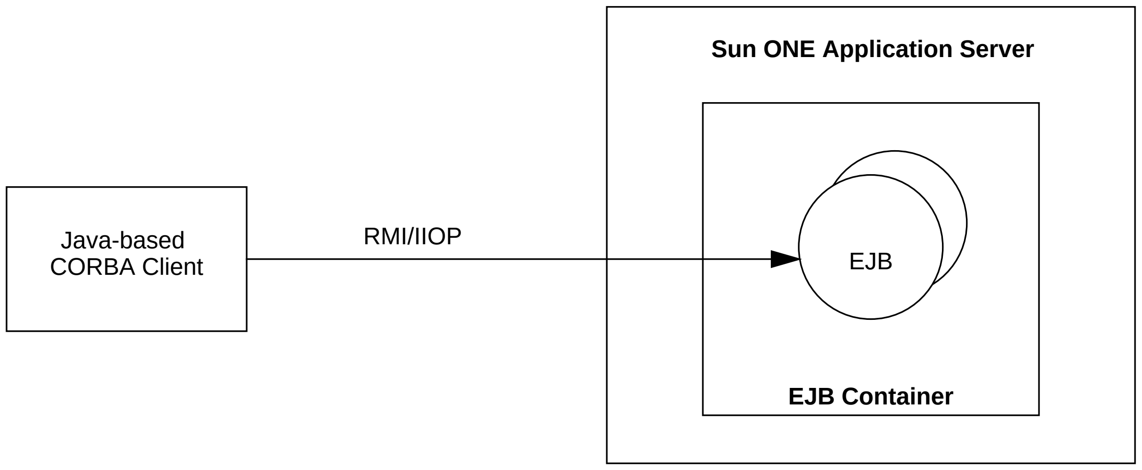
The most common scenarios in which CORBA clients are used are when either a stand-alone program or another application server acts as a client to EJBs deployed to Sun ONE Application Server. This section describes the following scenarios:

- Stand-alone Scenario
- Server to Server Scenario

Stand-alone Scenario

In the simplest case, a stand-alone program which does not use the ACC, running on a variety of operating systems uses IIOP to access business logic housed in backend EJB components, as shown in the figure "Stand-alone Client Accessing the EJB Components".

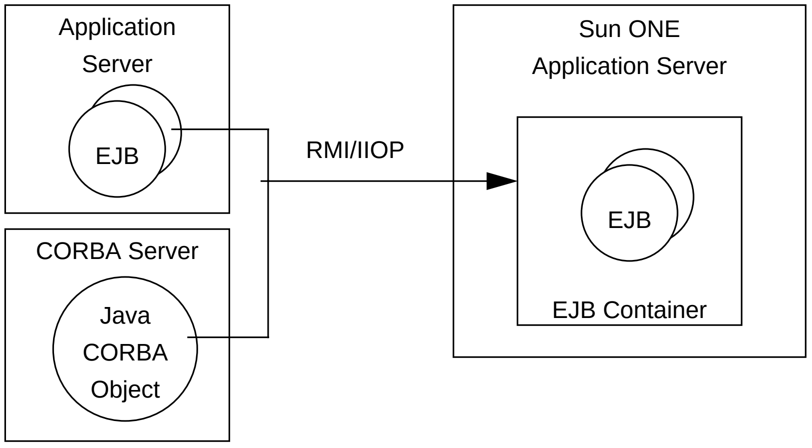
Figure 3-1 Stand-alone Client Accessing the EJB Components



Server to Server Scenario

CORBA objects, and other application servers can use IIOP to access EJB components housed in Sun ONE Application Server, as shown in the figure "Application Server and CORBA Objects Accessing EJB Components".

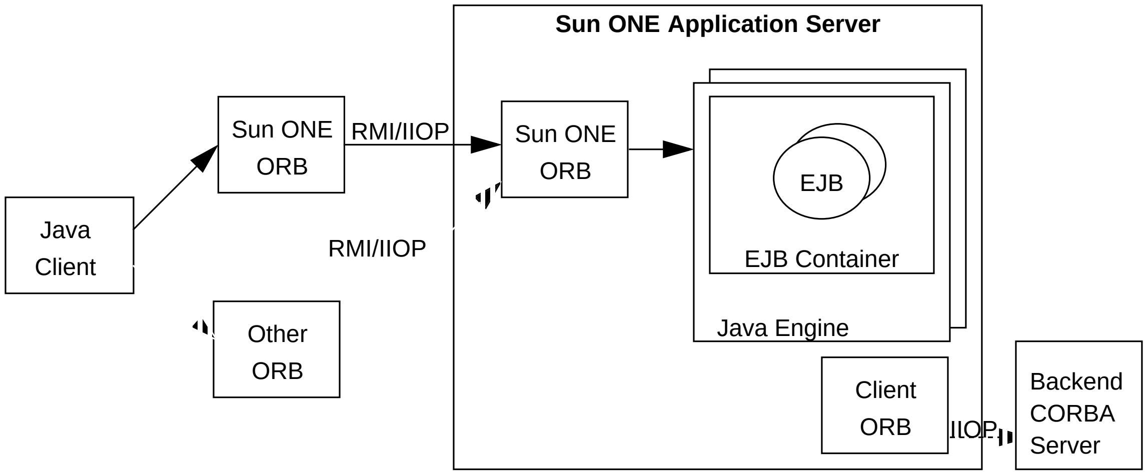
Figure 3-2 Application Server and CORBA Objects Accessing EJB Components



ORB Support Architecture

CORBA client support in Sun ONE Application Server involves the communication between the ORB on the client and the ORB on the server, as shown in the figure "ORB Support Architecture".

Figure 3-3 ORB Support Architecture



You can use the ORB that is bundled as part of the Sun ONE Application Server, or you can use a third-party ORB (ORBIX 2000 or ORBacus 4.1).

Developing Java-based CORBA Clients

This section describes the procedure to create, assemble, and deploy a Java-based CORBA client that is not packaged using the ACC. This section describes the following topics:

- Creating a Stand-alone CORBA Client
- Running a Stand-alone CORBA Client

Creating a Stand-alone CORBA Client

Clients do not directly access the EJB components. Instead, clients communicate with the EJB components using the JNDI to locate EJB components's home interface. Clients invoke a method on the EJB component's home interface to get a reference to the EJB components's home interface.

One of the first steps in coding a CORBA client using RMI/IIOP is, to perform a lookup of an EJB components's home interface. In preparation for performing a JNDI lookup of the home interface, you must first set several environment properties for the `InitialContext`. Then you provide a lookup name for the EJB component.

The steps and an example are summarized in the following sections.

- Specifying the Naming Factory Class
- Specifying the JNDI Name of an EJB

Specifying the Naming Factory Class

According to the RMI/IIOP specification, the client must specify `com.sun.jndi.cosnaming.CNCtxFactory` as the value of the `java.naming.factory.initial` entry in an instance of a `Properties` object. This object is then passed to the JNDI `InitialContext` constructor prior to looking up an EJB component's home interface. For example:

```
...

Properties env = new Properties();

env.put("java.naming.factory.initial","com.sun.jndi.cosnaming.CN
CtxFactory");

env.put("java.naming.provider.url", "iiop://" + host +":"+port);

Context initial = new InitialContext(env);
Object objref = initial.lookup("rmiconverter");

...
```

Specifying the JNDI Name of an EJB

After creating a new JNDI `InitialContext` object, your client calls the `lookup` method on the `InitialContext` to locate EJB component's home interface. The name of the EJB components is provided on the call to `lookup`. When using RMI/IIOP to access remote EJB components, the parameter is referred to as the "JNDI name" of the EJB component. The supported values of the JNDI name vary, depending on how your client application is packaged.

When the client application is not packaged as part of an Application Client Container (ACC), you must specify the absolute name of the EJB component in the JNDI lookup. You must use the prefix `java:comp/env/ejb/` when performing lookups using absolute references. For example, the lookup in the `rmiconverter` sample could be written as follows:

```
initial.lookup("java:comp/env/ejb/rmiconverter");
```

Or, with a module name, it could be written as follows:

```
initial.lookup("java:comp/env/ejb/rmiconverterEjb/
rmiconverter");
```

There is no mechanical difference between supplying this prefix and the first two approaches. You might find the `java:comp/env/ejb/` confusing when used in conjunction with absolute EJB references because this notation is typically used when you are using indirect EJB references.

NOTE Sun ONE Application Server does not support the authentication of Java-based stand-alone CORBA clients.

Sun ONE ORB Configuration

If you are using built-in Sun ONE ORB, you can configure client-side load balancing using the Round Robin DNS approach.

To implement a simple load balancing scheme without making source code changes to your client, you can leverage the round robin feature of DNS. In this approach, you define a single virtual host name representing multiple physical IP addresses on which server instance ORBs are listening. Assuming that you configure all of the ORBs to listen on a common IIOP port number, the client applications can use a single `host_name: IIOP port` during the JNDI lookup. The DNS server resolves the host name to a different IP address each time the client is executed.

You can also implement client-side load balancing using the Sun ONE Application Server-specific naming factory class `SIASCtxFactory`. You can use this class both on the client-side and on the server-side which maintains a pool of ORB instances in order to limit the number of ORB instances that are created in a given process.

The following code illustrates the use of `S1ASCtxFactory` class:

```
Properties env = new Properties();

env.setProperty("java.naming.factory.initial","com.sun.appserv.nami
ng.S1ASCtxFactory");
```

```
env.setProperty("org.omg.CORBA.ORBInitialHost", “name service
hostname");
```

```
env.setProperty("org.omg.CORBA.ORBInitialPort", "“name service port
number"");
```

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

If you set a single URL property for the host and port above, your code would look like this:

```
Properties env = new Properties();
```

```
env.setProperty("java.naming.factory.initial",
"com.sun.appserv.naming.S1ASCtxFactory");
```

```
env.setProperty("java.naming.provider.url", "iiop://“name service
hostname:name service port number");
```

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

If you prefer, you may set the host and port values and the URL value as Java System properties, instead of setting them in the environment as shown in the above code illustration. The values set in your code will, however, override any System property settings. Also, if you set both the URL and the host and port properties, the URL takes precedence.

Note that the `[name service hostname]` value mentioned above could be a name that maps to multiple IP addresses. The `S1ASCtxFactory` will appropriately round robin ORB instances across all the IP addresses everytime a user calls new `InitialContext()` method.

You can also use the following property of `S1ASCtxFactory` class to implement client-side load balancing:

`com.sun.appserv.iiop.loadbalancingpolicy=roundrobin,`*host1:port1,host2:port2,...,*

This property provides you with a list of *host:port* combinations to round robin the ORBs. These host names may also map to multiple IP addresses. If you use this property along with `org.omg.CORBA.ORBInitialHost` and `org.omg.CORBA.ORBInitialPort` as system properties, the round robin algorithm will round robin across all the values provided. If, however, you provide a host name and port number in your code, in the environment object, that value will override any such system property settings.

Running a Stand-alone CORBA Client

As long as the client environment is set appropriately and you are using a compatible JVM, you merely need to run the `main` class. Depending on whether you are passing the IIOP URL components (host and port number) on the command line or obtaining this information from a properties file, the exact manner in which you run the main program will vary. For example, the `rmiconverter` sample is run in the following manner:

```
java rmiconverter.ConverterClient host_name port
```

The *host_name* is the name of the host on which an ORB is listening on the specified *port*.

Third Party ORB Support

Sun ONE provides a built-in ORB to support IIOP access to the EJBs. You can also install and configure a third party ORB to use IIOP with Sun ONE Application Server.

For information on Configuring built-in ORB for supporting CORBA clients, see the *Sun ONE Application Server Administrator's Guide.*

This section discusses the following scenarios:

- Accessing EJBs in a Remote Application Server Instance From a Servlet/Enterprise JavaBean
- Configuring Back End Access Using Third Party Client ORBs Within Sun ONE Application Server

Accessing EJBs in a Remote Application Server Instance From a Servlet/Enterprise JavaBean

Sun ONE Application Server supports accessing the EJBs residing in another instance of the server via RMI/IIOP. This section describes the procedure to create a client application that accesses the EJB components residing in another instance of the application server.

Clients do not directly access the EJB components. Instead, clients communicate with the EJB components using the JNDI to locate EJB component's home interface. Clients invoke a method on the EJBs's home interface to get a reference to the EJB component's home interface.

One of the first steps in coding a client using RMI/IIOP is, to perform a lookup of an EJB component's home interface. In preparation for performing a JNDI lookup of the home interface, you must first set several environment properties for the `InitialContext`. Then you provide a lookup name for the EJB.

The steps and an example are summarized in the following sections.

- Specifying the Naming Factory Class
- Specifying the JNDI Name of an EJB

Specifying the Naming Factory Class

According to the RMI/IIOP specification, the client must specify `com.sun.jndi.cosnaming.CNCtxFactory` as the value of the `java.naming.factory.initial` entry in an instance of a `Properties` object. This object is then passed to the JNDI `InitialContext` constructor prior to looking up an EJB component's home interface. For example:

```
...

Properties env = new Properties();

env.put("java.naming.factory.initial","com.sun.jndi.cosnaming.CN
CtxFactory");

env.put("java.naming.provider.url", "iiop://" + host +":"+port);

Context initial = new InitialContext(env);
System.out.println("Inside other host after initialcontext");
Object objref = initial.lookup("MyConverter");

...
```

The above code line is part of the EJB business method.

Specifying the JNDI Name of an EJB

After creating a new JNDI `InitialContext` object, your client calls the `lookup` method on the `InitialContext` to locate the EJB component's home interface. The name of the EJB is provided on the call to `lookup`. When using RMI/IIOP to access remote EJBs, the parameter is referred to as the "JNDI name" of the EJB.

`initial.lookup("ejb/`*ejb-name*`");`

`initial.lookup("ejb/`*module-name/ejb-name*`");`

The *ejb-name* is the name of the EJB as it appears in the `<ejb-name>` element of the EJB's deployment descriptor. For example, here is a lookup using the value `Myconverter`:

```
initial.lookup("MyConverter");

ConverterHome home =
(ConverterHome)PortableRemoteObject.narrow(objref,ConverterHome.cla
ss);

Converter currencyConverter = home.create();

System.out.println("Inside other host after Create");
```

Configuring Back End Access Using Third Party Client ORBs Within Sun ONE Application Server

J2EE components (such as Servlet and EJBs) deployed to Sun ONE Application Server can access backend CORBA objects through third party Object Request Brokers (ORBs). This support enables J2EE applications to leverage investments in the existing CORBA-based business components. In addition to supporting server side access to backend CORBA objects, you can also use the built-in Sun ONE ORB for RMI/IIOP-based access to EJB components from Java or C++ application clients as explained in the RMI/IIOP samples.

Configuring Orbix ORB with Sun ONE Application Server involves the following steps:

- Installing Orbix
- Configuring Sun ONE Application Server to Use Orbix
- Overriding the Built-in ORB

Installing Orbix

To install Orbix, perform the following steps:

- Ensure that you have the Orbix 2000 software available for installation.
- Install the software. For instructions to install Orbix 2000, read through the *Orbix Installation Guide.*
- Verify to ensure that the Orbix configuration is proper.

Configuring Sun ONE Application Server to Use Orbix

You must configure the runtime environment to enable the application server to load the Orbix ORB classes. Add the following to the CLASSPATH:

- Orbix classes

- OMG classes
- Directory containing Orbix license file

Go to `Application Server Instances` -> `server1` (or any other instance) then click on Java Options and append the following:

classpath to Class Path Suffix text field under Directory Paths option

```
/etc/opt/iona/:/opt/iona/orbix_art/1.2/classes/orbix2000.jar:/opt/i
ona/orbix_art/1.2/classes/omg.jar
```

After modifying Class Path Suffix click Save then click on server1 (server instance) and click on Apply Changes tab, restart the application server instance to update the changes.

Overriding the Built-in ORB

Sun ONE Application Server relies on a built-in ORB to support RMI/IIOP access to EJB components from Java application clients. When implementing servlets and EJB components that access backend CORBA-based applications residing outside of the application server, you may need to override the built-in ORB classes with the ORB classes from third party products such as Orbix 2000.

You can use any of the following approaches to override the built-in ORB classes with ORB classes from third party products:

- ORB.init() Properties Approach
- orb.properties Approach
- Providing JVM Start-up Arguments

ORB.init() Properties Approach

The code illustration given below overrides the built-in Sun ONE ORB classes with ORB classes from IONA's ORBIX 2000.

For example:

```
...

Properties orbProperties = new Properties();

orbProperties.put("org.omg.CORBA.ORBClass","com.iona.corba.art.arti
mpl.ORBImpl");

orbProperties.put("org.omg.CORBA.ORBSingletonClass","com.iona.corba
.art.artimpl.ORBSingleton");

orb = ORB.init(args, orbProperties);
```

```
...
```

The advantage of this approach is that RMI/IIOP access to EJB components housed in the application server will still be performed using the built-in Sun ONE ORB classes while only access from servlets and EJB components to backend CORBA-based applications will use the third party ORB classes. This is the efficient method of supporting simultaneous use of multiple ORBs in the application server environment.

orb.properties Approach

In Java 2 1.2.1 environment, the JVM's `orb.properties` file contains property settings to identify the ORB implementation classes that are used by default throughout the JVM. To override the use of the built-in Sun ONE ORB classes, you can simply modify the `orb.properties` file to specify third party ORB classes and restart the application server.

For example, to set the implementation classes to specify the Orbix 2000 classes, make the following modification to the `orb.properties` file that is located at: *install_dir*/`jdk/jre/lib/`

Before:

```
org.omg.CORBA.ORBClass=com.sun.corba.se.internal.Interceptors.PIOR
```

```
org.omg.CORBA.ORBSingletonClass=com.sun.corba.se.internal.corba.ORB
Singleton
```

After:

```
org.omg.CORBA.ORBClass=com.iona.corba.art.artimpl.ORBImpl
```

```
org.omg.CORBA.ORBSingletonClass=com.iona.corba.art.artimpl.
ORBSingleton
```

The `javax.rmi` classes are used to support RMI/IIOP client access to EJB components housed in the application server. Since these classes are not used to access backend CORBA servers, you do not need to override these settings.

The main advantage of this approach is that it involves only one time setting for all applications deployed to the application server. There is no need for each servlet and/or EJB component that is acting as a client to a backend CORBA application to specify the ORB implementation classes.

Providing JVM Start-up Arguments

You can also specify the ORB implementation classes as server's JVM_ARGS in the `server.xml` file.

Go to the

instances_dir/`config`

and edit the `server.xml` file and add these jvm options as a subelement under `<java-config>` tag.

```
<jvm-options>
      -Dorg.omg.CORBA.ORBClass=com.iona.corba.art.artimpl.ORBImpl
</jvm-options>
<jvm-options>
   -Dorg.omg.CORBA.ORBSingletonClass=com.iona.corba.art.artimpl.
ORBSingleton
</jvm-options>
```

This approach gives the benefit of specifying the ORB implementation classes only once, but the main advantage when compared to changing the `orb.properties` file is that, the changes made to server's configuration file are specific to server instance and are applicable to all the applications running on a particular instance only.

You can find a sample that demonstrates the third-party ORB support in Sun ONE Application Server at the following location:

install_dir`/samples/corba/`

Chapter 4

C++ Clients

This chapter describes how to develop and deploy C++ clients that uses third-party ORBs.

This chapter contains the following sections:

Introducing C++ Clients

Sun ONE Application Server relies on the Sun ONE built-in ORB to support access to EJBs via RMI/IIOP. Java programs and other components, such as servlets and applets can use the existing RMI/IIOP support to access EJB components housed in Sun ONE Application Server.

A C++ client can access EJB components via IIOP. However, this can not be achieved using the Sun ONE ORB due to the absence of a Sun ONE ORB for C++ clients. A C++ client requires an ORB implementation on its side; the Sun ONE ORB has only a Java version of the implementation. This forces the C++ client developers to use a third-party ORB on the client side.

Developing a C++ Client

This section describes the steps to develop a C++ client using ORBacus 4.1 runtime and development environment. This C++ client will call methods of an EJB that are deployed to Sun ONE Application Server.

This section describes the following topics:

- Configuring C++ Clients to Access Sun ONE Application Server
- Creating a C++ Client

Configuring C++ Clients to Access Sun ONE Application Server

This section describes how to configure C++ clients to access Sun ONE Application Server. In the code example here, C++ client accesses the third party ORB ORBacus 4.1.

This section presents the following topics:

- Software Requirements
- Preparing for C++ Client Development
- Assumptions and Limitations

Software Requirements

The following software are necessary for the development of a C++ client:

SOLARIS:

- Solaris 2.8
- ORBacus 4.1 for C++ on Solaris
- Sun Workshop 6 Update 2 (C++ 5.2)
- Sun ONE Application Server
- Java^{TM} 2 Platform, Standard Edition (J2SE^{TM} platform) 1.4

Windows:

- Windows 2000
- ORBacus 4.1 for C++ on WIN 2000
- VC++ Version 6.0
- Sun ONE Application Server
- J2SE 1.4 for WIN 2000

Preparing for C++ Client Development

You must perform the following tasks before you start developing a C++ client:

1. Make sure that all the required software are installed. For more information on the software required for C++ client development, see "Software Requirements" on page 72.

2. Install Java Development Kit (JDK) 1.4.

3. Install ORBacus 4.1.

   For instructions on installing ORBacus 4.1, see the *ORBacus* documentation.

4. SOLARIS:

   Set the PATH to CC (C++ compiler of Sun workshop 6.2), `rmic` (RMI compiler), idl compiler of ORBacus.

   ```
   export
   PATH=<SUNworkshoppath>/SUNWspro/WS6U2/bin:<JDK_HOME>bin::$PATH
   ```

   `export ORBACUS_LICENSE=`*path to ORBacus 4.1 license file directory*`/licenses.txt`
   `export LD_LIBRARY_PATH=`*path to ORBacus home*`/lib`

   Windows:

   Set the PATH to cl (VC++ compiler of MicroSoft visual studio), `rmic` of JDK1.4, idl compiler of ORBacus.

   These can be set at the command prompt as follows:

   ```
   set PATH=C:Programfiles\MicrosoftVisualStudio\VC98·in;
   C:\J2SDK_Forte\jdk1.4.0·in;C:\ORBacus_IDL;%PATH%
   ```

   set `ORBACUS_LICENSE=`*path*/licenses.txt.

   You can also set the PATH from the Environmental Settings dialog box.

NOTE

- If your client development machine is different from that of the machine where Sun ONE Application Server is installed, copy the following classes to your client system:
  - The `appserv-ext.jar` part of Sun ONE Application Server available in *install dir/*`lib`.
  - All the classes corresponding to the application including home interface, remote interface, helper classes, and third party classes used by the application.
- Java language mapping specification does not support the use of Java package names differing only in case, to simplify the mapping. Sun ONE Application Server also does not support the use of class or interface names within the same package that differ only in case. Both of these are treated as errors. Therefore the deployed beans should not have package name and class name differing only in case.
- The explanations in this document are with respect to the sample application `Cart` available at the following location: *install_dir/*`samples/rmi-iiop/cpp/`

---

5. Install Sun ONE Application Server and test for basic functionality.

6. Deploy the sample application `Cart - BookCartApp.ear`.

You can deploy this application using the Administration interface. It is not mandatory to deploy the application, but a recommended step. For detailed information on deploying this application, see the *Sun ONE Application Server Administrator's Guide*.

NOTE To develop a C++ client, all the corresponding classes of the application should be accessible. That is, the home and remote interfaces of all the EJB components, helper classes, and other classes that are part of the application must be accessible. After the deployment, these can be made either part of Sun ONE Application Server or independent of Sun ONE Application Server.

Assumptions and Limitations

For Java data types such as, HashTable or other custom Java classes that have to be passed by value, you have to provide native C++ implementation or provide a wrapper over existing C++ implementation of those classes (such as STL) that conforms to the IDL files generated for the Java classes.

Creating a C++ Client

This section describes the procedure to create a C++ client that uses a third party ORB. The developed C++ client application can then be deployed to Sun ONE Application Server. The following are the major steps involved in creating a C++ client:

- Generating the IDL Files
- Generating CPP Files from IDL Files

Generating the IDL Files

1. Create a directory for C++ client development. For example:

   ```
   mkdir cppclient
   cd cppclient
   ```

2. Generate IDL files corresponding to remote and home interfaces of the EJB components, helper classes, and other third party classes used by J2EE applications.

   Use the `rmic` tool, which is part of JDK^{TM} 1.4, for generating IDL files.

   a. Generate the IDL files corresponding to home and remote interface of all the EJB components.

      When the IDL files corresponding to home and remote references are generated, the IDL files corresponding to the classes mentioned as part of the method signature are also generated. Thus, the separate IDL generation of those classes are not required. Generate only the classes which do not figure as part of the method signature separately.

      For example:

      I. `rmic -classpath`

         ```
         instance_dir/applications/j2ee-apps/BookCartApp_1/BookCartApp
         Ejb_jar:install_dir/lib/appserv-ext.jar
         -idl samples.rmi_iiop.cpp.ejb.CartHome
         ```

II. `rmic -classpath`
instance_dir/`applications/j2ee-apps/BookCartApp_1/BookCartAppEjb_jar:`*install_dir/*`lib/appserv-ext.jar -idl samples.rmi_iiop.cpp.ejb.Cart`

III. `rmic -classpath`
instance_dir/`applications/j2ee-apps/BookCartApp_1/BookCartAppEjb_jar:`*install_dir/*`lib/appserv-ext.jar -idl samples.rmi_iiop.cpp.ejb.InterfaceTestClass`

`-classpath -` contains the path to all the classes against which IDL is being generated. If the classes appearing as arguments to the method are part of a different package, include those paths also. Include the path to `appserv-ext.jar` in the classes.

The generated IDL files will be stored under directories corresponding to the package of the classes.

For example, the `Cart.class` will be mapped to `Cart.idl` and will be under `/cppclient/samples/rmi_iiop/cpp/ejb/` directory.

Similarly, classes corresponding to JDK are generated under `java/lang`, `java/io`, `javax/rmi/ejb`, `org/omg/` and other similar directories.

3. Generate the valuetypes corresponding to the classes native to J2SDK.

   As mentioned in Step 2, when IDL specific to application classes such as, home interface, remote interface, and other classes part of the application are generated, it also generates the IDLs corresponding to the classes native to the JDK.

   The classes of JDK that are serializable get mapped as IDL value types. You have to provide the implementation for these valuetypes using the IDL-to-CPP compiler.

   This will create C++ classes corresponding to the classes native to JDK. However, these C++ files have only dummy methods apart from protected methods that have implementation of accessor and modifier methods. If you need to manipulate the C++ objects, you need to add new methods to the generated C++ files.

   If the Java class has any member variables, then the value type implementation of that class will have accessor and modifier methods and they are protected. You can add new public methods in the implementation class of valuetypes to access and modify those member variables by calling the corresponding protected methods.

   Subsequently, compile these classes to generate an object file or as a shared library. This is a one time effort and you do not require perform for every J2EE application that you develop. You may re-use these implementations.

4. Develop the library for the valuetype implementations.

   The following steps describe the procedure to develop your own library for the valuetype implementations. All these valuetype implementations can be grouped as a library. This library should contain object files (valuetype implementation), the header(.h) and the IDL (.idl) files.

   a. Modify the IDL files as required by following the guidelines given in the next step.

   b. Generate cpp files for all the IDL files corresponding to the Java classes using the IDL compiler supplied with ORBacus. For example,

      `idl --impl-all -I. -I`*classpath to IDL files* `-I`*orbacus_home/*`idl/`
      `-I`*orbacus_home/*`idl/OB *.idl`

   c. Implement the valuetype types, if required.

      This is required only if you need to manipulate the object. For example, collection classes like Vector, Hashtable, etc., proper implementation has to be provided as lists so that elements can be retrieved and added to the list.

d. Compile the cpp file to generate an object file or a shared library.

NOTE Generate the Java language classes before processing other IDL files. Implement all the IDL files corresponding to the JDK before proceeding with application specific IDL files.

---

5. Modify the generated IDL files such as the EJBs, helper classes, and third-party classes corresponding to the application.

   The generated IDL files do not compile directly. You need to manually modify the IDL files for generating a CPP file. The list below explains the situations when you need to modify the IDL files:

NOTE This is not a complete list and you may need to make suitable modification to IDL files for successful generation of IDL files to CPP files.

---

a. Delete the duplicate variables defined.

   For example, in `Employee.idl,` employee_ is defined twice as:
   `private::CORBA::WStringValue employee_;`
   `attribute::CORBA::WStringValue employee_;`

   Either of the duplicate entries can be deleted. Deleting the following attribute is recommended:
   `attribute::CORBA::WStringValue employee_;`

b. Change the custom valuetypes to non-custom valuetypes.

   For example, Valuetype Exception inherits from `Throwable`, which is a custom valuetype. Remove the tag custom from the `Throwable` valuetype definition.

c. There will be cases where the same IDL file will be included more than once. This will result in improper generation of the CPP files. Comment such multiple includes.

   - For example, `Exception.idl` under `java/lang` has `java/lang/Throwable.idl` included twice. Comment the second include.
   - The IDL file may compile even when multiple includes are present. However, the generated CPP file will be incorrect.

d. There will be cases where other IDL files are included circularly.

   Some of the abstract valuetypes would be inheriting from `java::io::Serializable`. Remove such inheritance.

   For example, in `InterfaceTest.idl`, `InterfaceTest` is an abstract valuetype and it inherits from `java::io::Serializable`. Remove this inheritance.

Generating CPP Files from IDL Files

To generate the .cpp files form the .idl files, perform the following steps:

1. Go to the path where the IDL files are generated. Include the following paths to the idl command:

   a. paths to all the application IDLs

   b. paths to all the JDK related IDLs

   c. *ORBacus_home*/`idl`

   d. *ORBacus_home*/`idl/OB`

   The paths are included by the `-I` option.

2. Execute the following command with the paths mentioned in Step 1, with `--impl-all` *options idl_file_name*.

   For example,

   ```
   idl --impl-all -Iclasspath_to_java_classes_IDL -I/cppclient
   -I/orbacus_home/idl/ -I/orbacus_home/idl/OB -I. ComplexObject.idl
   ```

   You must first include the *classpath to Java classes IDL* files.

3. Execute the above command for all the IDL files corresponding to the application in all the directories.

4. Modify the generated classes.

   Some of the cpp files should be manually modified. The situations under which modifications are required are given below:

a. There can be clashes in the namespaces that appear in the code generated from IDL to CPP using the IDL tool.

The following examples illustrate the scenarios:

Example 1

The class, `ClassDesc`, generated under `javax/rmi/CORBA` uses the classes such as, `CORBA::ValueBase`. The class, `CORBA::ValueBase`, is part of the ORB implementation and is defined under the namespace, CORBA.

`ClassDesc` is defined under the namespace, `javax::rmi::CORBA`. If a reference to ValueBase as `CORBA::ValueBase` is made inside this class, it looks for its definition under the `javax::rmi::CORBA` namespace.

This fails as it is defined under the namespace CORBA and not `javax::rmi::CORBA`. To force it to look in the namespace CORBA, change the syntax to `javax::rmi::CORBA::ValueBase`.

Example 2

In the class example generated under the `java/lang` directory, there are references to the Exception class.

There are two types of exceptions: `CORBA::Exception` and `java::lang::Exception`. Change to `java::lang::Exception` from `CORBA::Exception`. These kind of code changes are required for the classes to compile properly.

NOTE You need not compile the classes corresponding to the skeletons, as they will not be used to implement the valuetypes.

5. Implement the valuetypes.

The `--impl-all` option to the IDL command also generates the code for the valuetype implementation, including the factories for creating the value types. The valuetype implementation will have most of the methods as protected.

Therefore, they cannot be accessed directly and add new methods to the valuetype implementation that are public. These methods call the protected methods to achieve the desired functionality. The client programs will call these newly added methods depending on the functionality.

However, sometimes these public methods are also generated by the IDL. In such cases implementation can be provided in these methods by calling the protected methods without adding new methods.

This type of generation is dependent on whether the variables are defined as private or attribute in the IDL files. For example, `Employee.class` gets mapped as `Employee` valuetype. The implementation which is `Employee.cpp` generated for this valuetype as part of IDL command consists of the method, `employee_()` as protected. Since this cannot be accessed directly, we have to add `getEmployeeName()` as a public method in the `Employee_impl.cpp` and `Employee.h`. This method calls `employee_()` method to achieve the functionality of returning the `EmployeeName`.

NOTE You may have to add additional methods to achieve specific functionality and to change the state of the object. These are determined by your application design and the required functionality.

6. Compile the value type implementations and other generated cpp files. You need to write the makefile to generate a cpp file.

Windows:

Use the `/GR` option.

7. Develop the client program as required by design and functionality.

   Include the header files of all the valuetypes. The following code illustrates the steps:

   ```
   samples::rmi_iiop::cpp::ejb::ComplexObjectFactory_impl
   *complexObjectVf = new
   samples::rmi_iiop::cpp::ejb::ComplexObjectFactory_impl();
   ```

   ```
   // initializing the ORB
   ```

   ```
   CORBA::ORB_var orb = CORBA::ORB_init(argc,argv);
   ```

   ```
   // registering the value factories. This is required for
   //unmarshalling the valuetypes
   ```

   ```
   orb->register_value_factory(
   samples::rmi_iiop::cpp::ejb::ComplexObject::_OB_id(),complexObje
   ctVf);
   ```

   Register the valuefactories after `orbinit()`. The registration of the valuefactories are very essential. If they are not registered, it results in marshalling exceptions and the ORB fails to unmarshall valuetypes.

8. Compile and link the client program with the previously generated object files.

   Windows

   Use `/GR` option.

9. Run the client program.

   Provide the `NameService` URL to the program. You can pass this as the `-ORBconfig <config file>` property to the client. The configuration file contains the `NameService` URL as follows:

   ```
   ooc.orb.service.NameService=corbaloc::green.india.sun.com:1050/N
   ameService
   ```

   For other ways to pass the `NameService` URL, refer to the ORBacus documentation.

   For example, `c++client -ORBconfig =` *config_file_path/config_file_name*

Sample Applications

RMI/IIOP sample applications have been bundled with Sun ONE Application Server. These samples have been augmented with detailed setup instructions for deploying the application to Sun ONE Application Server. The setup documentation and source code are available at the following location:

install_dir/`samples/rmi-iiop/`

Glossary

This glossary provides definitions for common terms used to describe the Sun ONE Application Server deployment and development environment. For a glossary of standard J2EE terms, please see the J2EE glossary at:

```
http://java.sun.com/j2ee/glossary.html
```

access control The means of securing your Sun ONE Application Server by controlling who and what has access to it.

ACL Access Control List. ACLs are text files that contain lists identifying who can access the resources stored on your Sun ONE Application Server. *See also* general ACL.

activation The process of transferring an enterprise bean's state from secondary storage to memory.

Administration interface The set of browser based forms used to configure and administer the Sun ONE Application Server. S*ee also* CLI.

administration server An application server instance dedicated to providing the administrative functions of the Sun ONE Application Server, including deployment, browser-based administration, and access from the command-line interface (CLI) and Integrated Development Environment (IDE).

administrative domain Multiple administrative domains is a feature within the Sun ONE Application Server that allows different administrative users to create and manage their own domains. A domain is a set of instances, created using a common set of installed binaries in a single system.

API Applications Program Interface. A set of instructions that a computer program can use to communicate with other software or hardware that is designed to interpret that API.

applet A small application written in Java that runs in a web browser. Typically, applets are called by or embedded in web pages to provide special functionality. By contrast, a *servlet* is a small application that runs on a server.

application A group of components packaged into an `.ear` file with a J2EE application deployment descriptor. *See also* component, module.

application client container *See* container.

application server A reliable, secure, and scalable software platform in which business applications are run. Application servers typically provide high-level services to applications, such as component lifecycle, location, and distribution and transactional resource access,

application tier A conceptual division of a J2EE application:

client tier: The user interface (UI). End users interact with client software (such as a web browser) to use the application.

server tier: The business logic and presentation logic that make up your application, defined in the application's components.

data tier: The data access logic that enables your application to interact with a data source.

assembly The process of combining discrete components of an application into a single unit that can be deployed. S*ee also* deployment.

asynchronous communication A mode of communication in which the sender of a message need not wait for the sending method to return before it continues with other work.

attribute A name-value pair in a request object that can be set by a servlet. Also a name-value pair that modifies an element in an XML file. Contrast with *parameter*. More generally, an attribute is a unit of metadata.

auditing The method(s) by which significant events are recorded for subsequent examination, typically in error or security breach situations.

authentication The process by which an entity (such as a user) proves to another entity (such as an application) that it is acting on behalf of a specific identity (the user's security identity). Sun ONE Application Server supports basic, form-based, and SSL mutual authentication. *See also* client authentication, digest authentication, host-IP authentication, pluggable authentication.

authorization The process by which access to a method or resource is determined. Authorization in the J2EE platform depends upon whether the user associated with a request through authentication is in a given security role. For example, a human resources application may authorize managers to view personal employee information for all employees, but allow employees to only view their own personal information.

backup store A repository for data, typically a file system or database. A backup store can be monitored by a background thread (or sweeper thread) to remove unwanted entries.

bean-managed persistence Data transfer between an entity bean's variables and a data store. The data access logic is typically provided by a developer using Java Database Connectivity (JDBC) or other data access technologies. *See also* container-managed persistence.

bean-managed transaction Where transaction demarcation for an enterprise bean is controlled programmatically by the developer. *See also* container-managed transaction.

BLOB Binary Large OBject. A data type used to store and retrieve complex object fields. BLOBs are binary or serializable objects, such as pictures, that translate into large byte arrays, which are then serialized into container-managed persistence fields.

BMP *See* bean-managed persistence.

BMT *See* bean-managed transaction.

broker The Sun ONE Message Queue entity that manages JMS message routing, delivery, persistence, security, and logging, and which provides an interface that allows an administrator to monitor and tune performance and resource use.

business logic The code that implements the essential business rules of an application, rather than data integration or presentation logic.

CA See certificate authority or connector architecture.

cached rowset A `CachedRowSet` object permits you to retrieve data from a data source, then detach from the data source while you examine and modify the data. A cached row set keeps track both of the original data retrieved, and any changes made to the data by your application. If the application attempts to update the original data source, the row set is reconnected to the data source, and only those rows that have changed are merged back into the database.

Cache Control Directives Cache-control directives are a way for Sun ONE Application Server to control what information is cached by a proxy server. Using cache-control directives, you override the default caching of the proxy to protect sensitive information from being cached, and perhaps retrieved later. For these directives to work, the proxy server must comply with HTTP 1.1.

callable statement A class that encapsulates a database procedure or function call for databases that support returning result sets from stored procedures.

certificate Digital data that specifies the name of an individual, company, or other entity, and certifies that the public key included in the certificate belongs to that entity. Both clients and servers can have certificates.

certificate authority A company that sells certificates over the Internet, or a department responsible for issuing certificates for a company's intranet or extranet.

cipher A cryptographic algorithm (a mathematical function), used for encryption or decryption.

CKL Compromised Key List. A list, published by a certificate authority, that indicates any certificates that either client users or server users should no longer trust. In this case, the key has been compromised. *See also* CRL.

classloader A Java component responsible for loading Java classes according to specific rules. *See also* classpath.

classpath A path that identifies directories and JAR files where Java classes are stored. *See also* classloader.

CLI Command-line interface. An interface that enables you to type executable instructions at a user prompt. S*ee also* Administration interface.

client authentication The process of authenticating client certificates by cryptographically verifying the certificate signature and the certificate chain leading to the CA on the trust CA list. See also authentication, certificate authority.

client contract A contract that determines the communication rules between a client and the EJB container, establishes a uniform development model for applications that use enterprise beans, and guarantees greater reuse of beans by standardizing the relationship with the client.

CMP *See* container-managed persistence.

CMR *See* container-managed relationship.

CMT *See* container-managed transaction.

co-locate To position a component in the same memory space as a related component in order avoid remote procedure calls and improve performance.

column A field in a database table.

commit To complete a transaction by sending the required commands to the database. *See* rollback, transaction.

component A web application, enterprise bean, message-driven bean, application client, or connector. *See also* application, module.

component contract A contract that establishes the relationship between an enterprise bean and its container.

configuration The process of tuning the server or providing metadata for a component. Normally, the configuration for a specific component is kept in the component's deployment descriptor file. *See also* administration server, deployment descriptor.

connection factory An object that produces connection objects that enable a J2EE component to access a resource. Used to create JMS connections (TopicConnection or QueueConnection) which allow application code to make use of the provided JMS implementation. Application code uses the JNDI Service to locate connection factory objects using a JNDI Name.

Connection Pool allows highly efficient access to a database by caching and reusing physical connections, thus avoiding connection overhead and allowing a small number of connections to be shared between a large number of threads. *See also* JDBC connection pool

connector A standard extension mechanism for containers to provide connectivity to EISs. A connector is specific to an EIS and consists of a resource adapter and application development tools for EIS connectivity. The resource adapter is plugged in to a container through its support for system level contracts defined in the connector architecture.

connector architecture An architecture for the integration of J2EE applications with EISs. There are two parts to this architecture: a EIS vendor-provided resource adapter and a J2EE server that allows this resource adapter to plug in. This architecture defines a set of contracts that a resource adapter has to support to plug in to a J2EE server, for example, transactions, security and resource management.

container An entity that provides life cycle management, security, deployment, and runtime services to a specific type of J2EE component. Sun ONE Application Server provides web and EJB containers, and supports application client containers. *See also* component.

container-managed persistence Where the EJB container is responsible for entity bean persistence. Data transfer between an entity bean's variables and a data store, where the data access logic is provided by the Sun ONE Application Server. *See also* bean-managed persistence.

container-managed relationship A relationship between fields in a pair of classes where operations on one side of the relationship affect the other side.

container-managed transaction Where transaction demarcation for an enterprise bean is specified declaratively and automatically controlled by the EJB container *See also* bean-managed transaction.

control descriptor A set of enterprise bean configuration entries that enable you to specify optional individual property overrides for bean methods, plus enterprise bean transaction and security properties.

conversational state Where the state of an object changes as the result of repeated interactions with the same client. *See also* persistent state.

cookie A small collection of information that can be transmitted to a calling web browser, then retrieved on each subsequent call from that browser so the server can recognize calls from the same client. Cookies are domain-specific and can take advantage of the same web server security features as other data interchange between your application and the server.

CORBA Common Object Request Broker Architecture. A standard architecture definition for object-oriented distributed computing.

COSNaming Service An an IIOP-based naming service.

CosNaming provider To support a global JNDI name space (accessible to IIOP application clients), Sun ONE Application Server includes J2EE based CosNaming provider which supports binding of CORBA references (remote EJB references).

create method A method for customizing an enterprise bean at creation.

CRL Certificate Revocation List. A list, published by a certificate authority, that indicates any certificates that either client users or server users should no longer trust. In this case, the certificate has been revoked. *See also* CKL.

data access logic Business logic that involves interacting with a data source.

database A generic term for Relational Database Management System (RDBMS). A software package that enables the creation and manipulation of large amounts of related, organized data.

database connection A database connection is a communication link with a database or other data source. Components can create and manipulate several database connections simultaneously to access data.

data source A handle to a source of data, such as a database. Data sources are registered with the iPlanet Application Server and then retrieved programmatically in order to establish connections and interact with the data source. A data source definition specifies how to connect to the source of data.

DataSource Object A DataSource object has a set of properties that identify and describe the real world data source that it represents.

declarative security Declaring security properties in the component's configuration file and allowing the component's container (for instance, a bean's container or a servlet engine) to manage security implicitly. This type of security requires no programmatic control. Opposite of programmatic security. *See* container-managed persistence.

declarative transaction See container-managed transaction.

decryption The process of transforming encrypted information so that it is intelligible again.

delegation An object-oriented technique for using the composition of objects as an implementation strategy. One object, which is responsible for the result of an operation, delegates the implementation to another object, its delegatee. For example, a classloader often delegates the loading of some classes to its parent.

deployment The process of distributing the files required by an application to an application server to make the application available to run on the application server. *See also* assembly.

deployment descriptor An XML file provided with each module and application that describes how they should be deployed. The deployment descriptor directs a deployment tool to deploy a module or application with specific container options and describes specific configuration requirements that a deployer must resolve.

destination resource An objects that represents Topic or Queue destinations. Used by applications to read / write to Queues or publish / subscribe to Topics. Application code uses the JNDI Service to locate JMS resource objects using a JNDI Name.

digest authentication A for of authentication that allows the user to authenticate based on user name and password without sending the user name and password as cleartext.

digital signature an electronic security mechanism used to authenticate both a message and the signer.

directory server See Sun ONE Directory Server.

Distinguished Name *See* DN, DN attribute.

distributable session A user session that is distributable among all servers in a cluster.

distributed transaction A single transaction that can apply to multiple heterogeneous databases that may reside on separate servers.

Document Root The document root (sometimes called the primary document directory) is the central directory that contains all the virtual server's files you want to make available to remote clients.

Domain Registry The Domain Registry is a single data structure that contains domain-specific information, for all the domains created and configured on an installation of Sun ONE Application Server, such as domain name, domain location, domain port, domain host.

DTD Document Type Definition. A description of the structure and properties of a class of XML files.

DN Distinguished Name. The string representation for the name of an entry in a directory server.

DN attribute Distinguished Name attribute. A text string that contains identifying information for an associated user, group, or object.

dynamic redeployment The process of redeploying a component without restarting the server.

dynamic reloading The process of updating and reloading a component without restarting the server. By default, servlet, JavaServer Page (JSP), and enterprise bean components can be dynamically reloaded. Also known as versioning.

EAR file Enterprise ARchive file. An archive file that contains a J2EE application. EAR files have the `.ear` extension. *See also* JAR file.

e-commerce Electronic commerce. A term for business conducted over the Internet.

EIS Enterprise Information System. This can be interpreted as a packaged enterprise application, a transaction system, or a user application. Often referred to as an EIS. Examples of EISs include: R/3, PeopleSoft, Tuxedo, and CICS.

EJB container *See* container.

EJB QL EJB Query Language. A query language that provides for navigation across a network of entity beans defined by container-managed relationships.

EJB technology An enterprise bean is a server-side component that encapsulates the business logic of an application. The business logic is the code that fulfills the purpose of the application. In an inventory control application, for example, the enterprise beans might implement the business logic in methods called `checkInventoryLevel` and `orderProduct`. By invoking these methods, remote clients can access the inventory services provided by the application. *See also* container, entity bean, message-driven bean, and session bean.

ejbc utility The compiler for enterprise beans. It checks all EJB classes and interfaces for compliance with the EJB specification, and generates stubs and skeletons.

element A member of a larger set; for example, a data unit within an array, or a logic element. In an XML file, it is the basic structural unit. An XML element contains subelements or data, and may contain attributes.

encapsulate To localize knowledge within a module. Because objects encapsulate data and implementation, the user of an object can view the object as a black box that provides services. Instance variables and methods can be added, deleted, or changed, but if the services provided by the object remain the same, code that uses the object can continue to use it without being rewritten.

encryption The process of transforming information so it is unintelligible to anyone but the intended recipient.

entity bean An enterprise bean that relates to physical data, such as a row in a database. Entity beans are long lived, because they are tied to persistent data. Entity beans are always transactional and multi-user aware. *See* message-driven bean, read-only bean, session bean.

ERP Enterprise Resource Planning. A multi-module software system that supports enterprise resource planning. An ERP system typically includes a relational database and applications for managing purchasing, inventory, personnel, customer service, shipping, financial planning, and other important aspects of the business.

event A named action that triggers a response from a module or application.

external JDNI resource Allows the JNDI Service to act as a bridge to a remote JNDI server.

facade Where an application-specific stateful session bean is used to manage various Enterprise JavaBeans (EJBs).

factory class A class that creates persistence managers. *See also* connection factory.

failover A recovery process where a bean can transparently survive a server crash.

finder method Method which enables clients to look up a bean or a collection of beans in a globally available directory.

firewall an electronic boundary that allows a network administrator to restrict the flow of information across networks in order to enforce security.

File Cache The file cache contains information about files and static file content. The file cache is turned on by default.

form action handler A specially defined method in servlet or application logic that performs an action based on a named button on a form.

FQDN Fully Qualified Domain Name. The full name of a system, containing its hostname and its domain name.

general ACL A named list in the Sun ONE Directory Server that relates a user or group with one or more permissions. This list can be defined and accessed arbitrarily to record any set of permissions.

generic servlet A servlet that extends `javax.servlet.GenericServlet`. Generic servlets are protocol-independent, meaning that they contain no inherent support for HTTP or any other transport protocol. Contrast with HTTP servlet.

global database connection A database connection available to multiple components. Requires a resource manager.

global transaction A transaction that is managed and coordinated by a transaction manager and can span multiple databases and processes. The transaction manager typically uses the XA protocol to interact with the database backends. *See* local transaction.

granularity level The approach to dividing an application into pieces. A *high level of granularity* means that the application is divided into many smaller, more narrowly defined Enterprise JavaBeans (EJBs). A *low level of granularity* means the application is divided into fewer pieces, producing a larger program.

group A group of users that are related in some way. Group membership is usually maintained by a local system administrator. *See* user, role.

handle An object that identifies an enterprise bean. A client may serialize the handle, and then later deserialize it to obtain a reference to the bean.

Heuristic Decision The transactional mode used by a particular transaction. A transaction has to either Commit or Rollback.

home interface A mechanism that defines the methods that enable a client to create and remove an enterprise bean.

host-IP authentication A security mechanism used for of limiting access to the Administration Server, or the files and directories on a web site by making them available only to clients using specific computers.

HTML Hypertext Markup Language. A coding markup language used to create documents that can be displayed by web browsers. Each block of text is surrounded by codes that indicate the nature of the text.

HTML page A page coded in HTML and intended for display in a web browser.

HTTP Hypertext Transfer Protocol. The Internet protocol that fetches hypertext objects from remote hosts. It is based on TCP/IP.

HTTP servlet A servlet that extends `javax.servlet.HttpServlet`. These servlets have built-in support for the HTTP protocol. Contrast with generic servlet.

HTTPS HyperText Transmission Protocol, Secure. HTTP for secure transactions.

IDE Integrated Development Environment. Software that allows you to create, assemble, deploy, and debug code from a single, easy-to-use interface.

IIOP Internet Inter-ORB Protocol. Transport-level protocol used by both Remote Method Invocation (RMI) over IIOP and Common Object Request Broker Architecture (CORBA).

IIOP Listener The IIOP listener is a listen socket that listens on a specified port and accepts incoming connections from CORBA based client application

IP address A structured, numeric identifier for a computer or other device on a TCP/IP network. The format of an IP address is a 32-bit numeric address written as four numbers separated by periods. Each number can be zero to 255. For example, 123.231.32.2 could be an IP address.

IMAP Internet Message Access Protocol.

isolation level *See* transaction isolation level.

J2EE Java 2 Enterprise Edition. An environment for developing and deploying multi-tiered, web-based enterprise applications. The J2EE platform consists of a set of services, application programming interfaces (APIs), and protocols that provide the functionality for developing these applications.

JAF The JavaBeans Activation Framework (JAF) integrates support for MIME data types into the Java platform. See Mime Types.

JAR file Java ARchive file. A file used for aggregating many files into one file. JAR files have the.`jar` extension.

JAR file contract Java ARchive contract that specifies what information must be in the enterprise bean package.

JAR file format Java ARchive file format. A platform-independent file format that aggregates many files into one file. Multiple applets and their requisite components (class files, images, sounds, and other resource files) can be bundled in a JAR file and subsequently downloaded to a browser in a single HTTP transaction. The JAR files format also supports file compression and digital signatures.

JavaBean A portable, platform-independent reusable component model.

Java IDL Java Interface Definition Language. APIs written in the Java programming language that provide a standards-based compatibility and connectivity with Common Object Request Broker Architecture (CORBA).

JavaMail session An object used by an application to interact with a mail store. Application code uses the JNDI Service to locate JavaMail session resources objects using a JNDI name.

JAXM Java API for XML Messaging. Enables applications to send and receive document-oriented XML messages using the SOAP standard. These messages can be with or without attachments.

JAXP Java API for XML Processing. A Java API that supports processing of XML documents using DOM, SAX, and XSLT. Enables applications to parse and transform XML documents independent of a particular XML processing implementation.

JAXR Java API for XML Registry. Provides a uniform and standard Java API for accessing different kinds of XML registries. Enables users to build, deploy and discover web services.

JAX-RPC Java API for XML-based Remote Procedure Calls. Enables developers to build interoperable web applications and web services based on XML-based RPC protocols.

JDBC Java Database Connectivity. A standards-based set of classes and interfaces that enable developers to create data-aware components. JDBC implements methods for connecting to and interacting with data sources in a platform- and vendor-independent way.

JDBC connection pool A pool that combines the JDBC data source properties used to specify a connection to a database with the connection pool properties.

JDBC resource A resource used to connect an application running within the application server to a database using an existing JDBC connection pool. Consists of a JNDI name (which is used by the application) and the name of an existing JDBC connection pool.

JDK Java Development Kit. The software that includes the APIs and tools that developers need to build applications for those versions of the Java platform that preceded the Java 2 Platform. *See also* JDK.

JMS Java Message Service. A standard set of interfaces and semantics that define how a JMS client accesses the facilities of a JMS message service. These interfaces provide a standard way for Java programs to create, send, receive, and read messages.

JMS-administered object A pre-configured JMS object—a connection factory or a destination—created by an administrator for use by one or more JMS clients.

The use of administered objects allows JMS clients to be provider-independent; that is, it isolates them from the proprietary aspects of a provider. These objects are placed in a JNDI name space by an administrator and are accessed by JMS clients using JNDI lookups.

JMS client An application (or software component) that interacts with other JMS clients using a JMS message service to exchange messages.

JMS connection factory The JMS administered object a JMS client uses to create a connection to a JMS message service.

JMS destination The physical destination in a JMS message service to which produced messages are delivered for routing and subsequent delivery to consumers. This physical destination is identified and encapsulated by an JMS administered object that a JMS client uses to specify the destination for which it is producing messages and/or from which it is consuming messages.

JMS messages Asynchronous requests, reports, or events that are consumed by JMS clients. A message has a header (to which additional fields can be added) and a body. The message header specifies standard fields and optional properties. The message body contains the data that is being transmitted.

JMS provider A product that implements the JMS interfaces for a messaging system and adds the administrative and control functions needed for a complete product.

JMS Service Software that provides delivery services for a JMS messaging system, including connections to JMS clients, message routing and delivery, persistence, security, and logging. The message service maintains physical destinations to which JMS clients send messages, and from which the messages are delivered to consuming clients.

JNDI Java Naming and Directory Interface. This is a standard extension to the Java platform, providing Java technology-enabled applications with a unified interface to multiple naming and directory services in the enterprise. As part of the Java Enterprise API set, JNDI enables seamless connectivity to heterogeneous enterprise naming and directory services.

JNDI name A name used to access a resource that has been registered in the JNDI naming service.

JRE Java Runtime Environment. A subset of the Java Development Kit (JDK) consisting of the Java virtual machine, the Java core classes, and supporting files that provides runtime support for applications written in the Java programming language. *See also* JDK.

JSP JavaServer Page. A text page written using a combination of HTML or XML tags, JSP tags, and Java code. JSPs combine the layout capabilities of a standard browser page with the power of a programming language.

jspc utility The compiler for JSPs. It checks all JSPs for compliance with the JSP specification.

JTA Java Transaction API. An API that allows applications and J2EE servers to access transactions.

JTS Java Transaction Service. The Java service for processing transactions.

key-pair file *See* trust database.

LDAP Lightweight Directory Access Protocol. LDAP is an open directory access protocol that runs over TCP/IP. It is scalable to a global size and millions of entries. Using Sun ONE Directory Server, a provided LDAP server, you can store all of your enterprise's information in a single, centralized repository of directory information that any application server can access through the network.

LDIF LDAP Data Interchange Format. Format used to represent Sun ONE Directory Server entries in text form.

lifecycle event A stage in the server life cycle, such as startup or shutdown.

lifecycle module A module that listens for and performs its tasks in response to events in the server life cycle.

Listener A class, registered with a posting object, that says what to do when an event occurs.

local database connection The transaction context in a local connection is local to the current process and to the current data source, not distributed across processes or across data sources.

local interface An interface that provides a mechanism for a client that is located in the same Java Virtual Machine (JVM) with a session or entity bean to access that bean.

local session A user session that is only visible to one server.

local transaction A transaction that is native to one database and is restricted within a single process. Local transactions work only against a single backend. Local transactions are typically demarcated using JDBC APIs. *See also* global transaction.

mapping The ability to tie an object-oriented model to a relational model of data, usually the schema of a relational database. The process of converting a schema to a different structure. Also refers to the mapping of users to security roles.

MDB *See* message-driven bean.

message-driven bean An enterprise bean that is an asynchronous message consumer. A message-driven bean has no state for a specific client, but its instance variables may contain state across the handling of client messages, including an open database connection and an object reference to an EJB object. A client accesses a message-driven bean by sending messages to the destination for which the message-driven bean is a message listener.

messaging A system of asynchronous requests, reports, or events used by enterprise applications that allows loosely coupled applications to transfer information reliably and securely.

metadata Information about a component, such as its name, and specifications for its behavior.

MIME Data Type MIME (Multi-purpose Internet Mail Extension) types control what types of multimedia files your system supports.

module A web application, enterprise bean, message-driven bean, application client, or connector that has been deployed individually, outside an application. *See also* application, component, lifecycle module.

NTV Name, Type, Value.

object persistence *See* persistence.

O/R mapping tool Object-to-relational [database] tool. A mapping tool within the Sun ONE Application Server Administrative interface that creates XML deployment descriptors for entity beans.

package A collection of related classes that are stored in a common directory. They are often literally packaged together in a Java archive JAR file. *See also* assembly, deployment.

parameter A name/value pair sent from the client, including form field data, HTTP header information, and so on, and encapsulated in a request object. Contrast with attribute. More generally, an argument to a Java method or database-prepared command.

passivation A method of releasing a bean's resources from memory without destroying the bean. In this way, a bean is made to be persistent, and can be recalled without the overhead of instantiation.

permission A set of privileges granted or denied to a user or group. *See also* ACL.

persistence For enterprise beans, the protocol for transferring the state of an entity bean between its instance variables and an underlying database. Opposite of transience. For sessions, the session storage mechanism.

persistence manager The entity responsible for the persistence of the entity beans installed in the container.

persistent state Where the state of an object is kept in persistent storage, usually a database.

pluggable authentication A mechanism that allows J2EE applications to use the Java Authentication and Authorization Service (JAAS) feature from the J2SE platform. Developers can plug in their own authentication mechanisms.

point-to-point delivery model Producers address messages to specific queues; consumers extract messages from queues established to hold their messages. A message is delivered to only one message consumer.

pooling The process of providing a number of preconfigured resources to improve performance. If a resource is pooled, a component can use an existing instance from the pool rather than instantiating a new one. In the Sun ONE Application Server, database connections, servlet instances, and enterprise bean instances can all be pooled.

POP3 Post Office Protocol

prepared command A database command (in SQL) that is precompiled to make repeated execution more efficient. Prepared commands can contain parameters. A prepared statement contains one or more prepared commands.

prepared statement A class that encapsulates a `QUERY`, `UPDATE`, or `INSERT` statement that is used repeatedly to fetch data. A prepared statement contains one or more prepared commands.

presentation layout The format of web page content.

presentation logic Activities that create a page in an application, including processing a request, generating content in response, and formatting the page for the client. Usually handled by a web application.

primary key The unique identifier that enables the client to locate a particular entity bean.

primary key class name A variable that specifies the fully qualified class name of a bean's primary key. Used for JNDI lookups.

principal The identity assigned to an entity as a result of authentication.

private key *See* public key cryptography.

process Execution sequence of an active program. A process is made up of one or more threads.

programmatic security The process of controlling security explicitly in code rather than allowing the component's container (for instance, a bean's container or a servlet engine) to handle it. Opposite of declarative security.

programmer-demarcated transaction *See* bean-managed transaction.

property A single attribute that defines the behavior of an application component. In the `server.xml` file, a property is an element that contains a name/value pair.

public key cryptography A form of cryptography in which each user has a public key and a private key. Messages are sent encrypted with the receiver's public key; the receiver decrypts them using the private key. Using this method, the private key never has to be revealed to anyone other than the user.

publish/subscribe delivery model Publishers and subscribers are generally anonymous and may dynamically publish or subscribe to a topic. The system distributes messages arriving from a topic's multiple publishers to its multiple subscribers.

queue An object created by an administrator to implement the point-to-point delivery model. A queue is always available to hold messages even when the client that consumes its messages is inactive. A queue is used as an intermediary holding place between producers and consumers.

QOS QOS (Quality of Service) refers to the performance limits you set for a server instance or virtual server. For example, if you are an ISP, you might want to charge different amounts of money for virtual servers depending on how much bandwidth is provided. You can limit two areas: the amount of bandwidth and the number of connections.

RAR file Resource ARchive. A JAR archive that contains a resource adapter.

RDB Relational database.

RDBMS Relational database management system.

read-only bean An entity bean that is never modified by an EJB client. *See also* entity bean.

realm A scope over which a common security policy is defined and enforced by the security administrator of the security service. Also called a *security policy domain* or *security domain* in the J2EE specification.

remote interface One of two interfaces for an Enterprise JavaBean. The remote interface defines the business methods callable by a client.

request object An object that contains page and session data produced by a client, passed as an input parameter to a servlet or JavaServer Page (JSP).

resource manager An object that acts as a facilitator between a resource such as a database or message broker, and client(s) of the resource such as Sun ONE Application Server processes. Controls globally-available data sources.

resource reference An element in a deployment descriptor that identifies the component's coded name for the resource.

response object An object that references the calling client and provides methods for generating output for the client.

ResultSet An object that implements the `java.sql.ResultSet` interface. `ResultSets` are used to encapsulate a set of rows retrieved from a database or other source of tabular data.

reusable component A component created so that it can be used in more than one capacity, for instance, by more than one resource or application.

RMI Remote Method Invocation. A Java standard set of APIs that enable developers to write remote interfaces that can pass objects to remote processes.

RMIC Remote Method Invocation Compiler.

role A functional grouping of subjects in an application, represented by one or more groups in a deployed environment. *See also* user, group.

rollback Cancellation of a transaction.

row A single data record that contains values for each column in a table.

RowSet An object that encapsulates a set of rows retrieved from a database or other source of tabular data. `RowSet` extends the `java.sql.ResultSet` interface, enabling `ResultSet` to act as a JavaBeans component.

RPC Remote Procedure Call. A mechanism for accessing a remote object or service.

runtime system The software environment in which programs run. The runtime system includes all the code necessary to load programs written in the Java programming language, dynamically link native methods, manage memory, and handle exceptions. An implementation of the Java virtual machine is included, which may be a Java interpreter.

SAF Server Application Function. A function that participates in request processing and other server activities

schema The structure of the underlying database, including the names of tables, the names and types of columns, index information, and relationship (primary and foreign key) information.

Secure Socket Layer *See* SSL.

security A screening mechanism that ensures that application resources are only accessed by authorized clients.

serializable object An object that can be deconstructed and reconstructed, which enables it to be stored or distributed among multiple servers.

server instance A Sun ONE Application Server can contain multiple instances in the same installation on the same machine. Each instance has its own directory structure, configuration, and deployed applications. Each instance can also contain multiple virtual servers. *See also* virtual server.

servlet An instance of the `Servlet` class. A servlet is a reusable application that runs on a server. In the Sun ONE Application Server, a servlet acts as the central dispatcher for each interaction in an application by performing presentation logic, invoking business logic, and invoking or performing presentation layout.

servlet engine An internal object that handles all servlet metafunctions. Collectively, a set of processes that provide services for a servlet, including instantiation and execution.

servlet runner The part of the servlet engine that invokes a servlet with a request object and a response object. *See* servlet engine.

session An object used by a servlet to track a user's interaction with a web application across multiple HTTP requests.

session bean An enterprise bean that is created by a client; usually exists only for the duration of a single client-server session. A session bean performs operations for the client, such as calculations or accessing other EJBs. While a session bean may be transactional, it is not recoverable if a system crash occurs. Session bean objects can be either stateless (not associated with a particular client) or stateful (associated with a particular client), that is, they can maintain conversational state across methods and transactions. *See also* stateful session bean, stateless session bean.

session cookie A cookie that is returned to the client containing a user session identifier. *See also* sticky cookie.

session timeout A specified duration after which the Sun ONE Application Server can invalidate a user session. *See* session.

single sign-on A situation where a user's authentication state can be shared across multiple J2EE applications in a single virtual server instance.

SMTP Simple Mail Transport Protocol

SNMP SNMP (Simple Network Management Protocol) is a protocol used to exchange data about network activity. With SNMP, data travels between a managed device and a network management station (NMS). A managed device is anything that runs SNMP: hosts, routers, your web server, and other servers on your network. The NMS is a machine used to remotely manage that network.

SOAP The Simple Object Access Protocol (SOAP) uses a combination of XML-based data structuring and Hyper Text Transfer Protocol (HTTP) to define a standardized way of invoking methods in objects distributed in diverse operating environments across the Internet.

SQL Structured Query Language. A language commonly used in relational database applications. SQL2 and SQL3 designate versions of the language.

SSL Secure Sockets Layer. A protocol designed to provide secure communications on the Internet.

state **1.** The circumstances or condition of an entity at any given time. **2.** A distributed data storage mechanism which you can use to store the state of an application using the Sun ONE Application Server feature interface `IState2`. *See also* conversational state, persistent state.

stateful session bean A session bean that represents a session with a particular client and which automatically maintains state across multiple client-invoked methods.

stateless session bean A session bean that represents a stateless service. A stateless session bean is completely transient and encapsulates a temporary piece of business logic needed by a specific client for a limited time span.

sticky cookie A cookie that is returned to the client to force it to always connect to the same server process. *See also* session cookie.

stored procedure A block of statements written in SQL and stored in a database. You can use stored procedures to perform any type of database operation, such as modifying, inserting, or deleting records. The use of stored procedures improves database performance by reducing the amount of information that is sent over a network.

streaming A technique for managing how data is communicated through HTTP. When results are streamed, the first portion of the data is available for use immediately. When results are not streamed, the whole result must be received before any part of it can be used. Streaming provides a way to allow large amounts of data to be returned in a more efficient way, improving the perceived performance of the application.

system administrator The person who administers Sun ONE Application Server software and deploys Sun ONE Application Server applications.

Sun ONE Application Server RowSet A `RowSet` object that incorporates the Sun ONE Application Server extensions.

Sun ONE Directory Server The Sun ONE version of Lightweight Directory Access Protocol (LDAP). Every instance of Sun ONE Application Server uses Sun ONE Directory Server to store shared server information, including information about users and groups. *See also* LDAP.

Sun ONE Message Queue The Sun ONE enterprise messaging system that implements the Java Message Service (JMS) open standard: it is a JMS provider.

TLS Transport Layer Security. A protocol that provides encryption and certification at the transport layer, so that data can flow through a secure channel without requiring significant changes to the client and server applications.

table A named group of related data in rows and columns in a database.

thread An execution sequence inside a process. A process may allow many simultaneous threads, in which case it is multi-threaded. If a process executes each thread sequentially, it is single-threaded.

topic An object created by an administrator to implement the publish / subscribe delivery model. A topic may be viewed as node in a content hierarchy that is responsible for gathering and distributing messages addressed to it. By using a topic as an intermediary, message publishers are kept separate from message subscribers.

transaction context A transaction's scope, either local or global. *See* local transaction, global transaction.

transaction isolation level Determines the extent to which concurrent transactions on a database are visible to one-another.

transaction manager An object that controls a global transaction, normally using the XA protocol. *See* global transaction.

transaction A set of database commands that succeed or fail as a group. All the commands involved must succeed for the entire transaction to succeed.

Transaction Recovery Automatic or manual recovery of distributed transactions.

Transaction Attribute A transaction attribute controls the scope of a transaction.

transience A protocol that releases a resource when it is not being used. Opposite of persistence.

trust database I security file that contains the public and private keys; also referred to as the key-pair file.

URI Universal Resource Identifier. Describes a specific resource at a domain. Locally described as a subset of a base directory, so that `/ham/burger` is the base directory and a URI specifies `toppings/cheese.html`. A corresponding URL would be `http://domain:port/toppings/cheese.html`.

URL Uniform Resource Locator. An address that uniquely identifies an HTML page or other resource. A web browser uses URLs to specify which pages to display. A URL describes a transport protocol (for example, HTTP, FTP), a domain (for example, `www.my-domain.com`), and optionally a URI.

user A person who uses an application. Programmatically, a user consists of a user name, password, and set of attributes that enables an application to recognize a client. *See also* group, role.

user session A series of user application interactions that are tracked by the server. Sessions maintain user state, persistent objects, and identity authentication.

versioning *See* dynamic reloading.

virtual server A virtual web server that serves content targeted for a specific URL. Multiple virtual servers may serve content using the same or different host names, port numbers, or IP addresses. The HTTP service can direct incoming web requests to different virtual servers based on the URL. Also called a virtual host.

A web application can be assigned to a specific virtual server. A server instance can have multiple virtual servers. *See also* server instance.

WAR file Web ARchive. A Java archive that contains a web module. WAR files have the`.war` extension.

web application A collection of servlets, JavaServer Pages, HTML documents, and other web resources, which might include image files, compressed archives, and other data. A web application may be packaged into an archive (a WAR file) or exist in an open directory structure.

Sun ONE Application Server also supports some non-Java web application technologies, such as SHTML and CGI.

web cache An Sun ONE Application Server feature that enables a servlet or JSP to cache its results for a specific duration in order to improve performance. Subsequent calls to that servlet or JSP within the duration are given the cached results so that the servlet or JSP does not have to execute again.

web connector plug-in An extension to a web server that enables it to communicate with the Sun ONE Application Server.

web container *See* container.

web module An individually deployed web application. *See* web application.

web server A host that stores and manages HTML pages and web applications, but not full J2EE applications. The web server responds to user requests from web browsers.

Web Server Plugin The web server plugin is an HTTP reverse proxy plugin that allows you to instruct a Sun One Web Server or Sun ONE Application Server to forward certain HTTP requests to another server.

web service A service offered via the web. A self-contained, self-describing, modular application that can accept a request from a system across the Internet or an intranet, process it, and return a response.

WSDL Web Service Description Language. An XML-based language used to define web services in a standardized way. It essentially describes three fundamental properties of a web service: definition of the web service, how to access that web service, and the location of that web service.

UDDI Universal Description, Discovery, and Integration. Provides worldwide registry of web services for discovery and integration.

XA protocol A database industry standard protocol for distributed transactions.

XML Extensible Markup Language. A language that uses HTML-style tags to identify the kinds of information used in documents as well as to format documents.

Index