

Developer's Guide to Clients

Sun™ ONE Application Server

Version 7, Enterprise Edition

817-2150-10
September 2003

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.

Copyright © 2003 Sun Microsystems, Inc. All rights reserved.

THIS SOFTWARE CONTAINS CONFIDENTIAL INFORMATION AND TRADE SECRETS OF SUN MICROSYSTEMS, INC. USE, DISCLOSURE OR REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR EXPRESS WRITTEN PERMISSION OF SUN MICROSYSTEMS, INC. U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements. Use is subject to license terms.

This distribution may include materials developed by third parties.

Sun, Sun Microsystems, the Sun logo, Java, Sun™ ONE, the Java Coffee Cup logo and the Sun™ ONE logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

This product is covered and controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

Copyright © 2003 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, Etats-Unis. Tous droits réservés.

CE LOGICIEL CONTIENT DES INFORMATIONS CONFIDENTIELLES ET DES SECRETS COMMERCIAUX DE SUN MICROSYSTEMS, INC. SON UTILISATION, SA DIVULGATION ET SA REPRODUCTION SONT INTERDITES SANS L'AUTORISATION EXPRESSE, ÉCRITE ET PRÉALABLE DE SUN MICROSYSTEMS, INC. Droits du gouvernement américain, utilisateurs gouvernementaux - logiciel commercial. Les utilisateurs gouvernementaux sont soumis au contrat de licence standard de Sun Microsystems, Inc., ainsi qu'aux dispositions en vigueur de la FAR (Federal Acquisition Regulations) et des suppléments à celles-ci. L'utilisation est soumise aux termes de la Licence.

Cette distribution peut comprendre des composants développés par des tierces parties.

Sun, Sun Microsystems, le logo Sun, Java, Sun™ ONE, le logo Java Coffee Cup et le logo Sun™ ONE sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Ce produit est soumis à la législation américaine en matière de contrôle des exportations et peut être soumis à la réglementation en vigueur dans d'autres pays dans le domaine des exportations et importations. Les utilisations, ou utilisateurs finaux, pour des armes nucléaires, des missiles, des armes biologiques et chimiques ou du nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou réexportations vers les pays sous embargo américain, ou vers des entités figurant sur les listes d'exclusion d'exportation américaines, y compris, mais de manière non exhaustive, la liste de personnes qui font objet d'un ordre de ne pas participer, d'une façon directe ou indirecte, aux exportations des produits ou des services qui sont régis par la législation américaine en matière de contrôle des exportations et la liste de ressortissants spécifiquement désignés, sont rigoureusement interdites.

Contents

About This Document	7
Who Should Use This Guide	7
Using the Documentation	8
How This Guide Is Organized	10
Reference Information	11
Documentation Conventions	11
General Conventions	11
Conventions Referring to Directories	12
Product Support	13
Chapter 1 Overview of Clients	15
Introducing Clients	15
Types of Clients	17
Web Clients	17
Web Services Clients	17
JMS Clients	18
CORBA Clients	18
Application Clients	19
Chapter 2 Using the Application Client Container	21
Introducing the Application Client Container	21
Application Client Container Features	22
Developing Applications Using the ACC	22
Creating an Application Client	23
Locating the Home Interface	23
Creating an Enterprise Bean Instance	24
Invoking a Business Method	24

Using an Application Client to Invoke an EJB Module	24
Making a Remote Call on the EJB	26
Using an Application Client to Access JMS Resources	26
Application Client Accessing JMS Resources Without Using the ACC	26
Application Client Packaged in an Application Client Container Accessing JMS Resources	33
Invoking an RMI/IIOP-based Client Without Using the ACC	36
Authenticating an Application Client Using the JAAS Module	37
Packaging an Application Client Using the ACC	45
Editing the Configuration File	45
Editing the appclient Script	46
Editing the sun-acc.xml File	46
Setting Security Options	47
Using the package-appclient Script	47
Running an Application Client Using the ACC	49
Sample Client Application	50
Application Client Deployment Descriptors	50
Format of Deployment Descriptors	50
Subelements	51
Data	52
Attributes	52
J2EE Application Client Deployment Descriptor	53
Sun ONE Application Client Deployment Descriptor	53
Elements in sun-application-client.xml file	53
Application Client Container Configuration File	57
Elements in the sun-acc.xml File	57
 Chapter 3 Java-based CORBA Clients	67
CORBA Client Scenarios	67
Stand-alone Scenario	67
Server to Server Scenario	68
ORB Support Architecture	69
Developing Java-based CORBA Clients	70
Creating a Stand-alone CORBA Client	70
Specifying the Naming Factory Class	71
Specifying the JNDI Name of an EJB	71
Sun ONE ORB Configuration	72
Running a Stand-alone CORBA Client	73
Third Party ORB Support	74
Accessing EJBs in a Remote Application Server Instance From a Servlet/Enterprise JavaBean ..	74
Specifying the Naming Factory Class	75
Specifying the JNDI Name of an EJB	75
Configuring Back End Access Using Third Party Client ORBs Within Sun ONE Application Server	

Installing Orbix	76
Configuring Sun ONE Application Server to Use Orbix	76
Overriding the Built-in ORB	77
Chapter 4 C++ Clients	81
Introducing C++ Clients	81
Developing a C++ Client	81
Configuring C++ Clients to Access Sun ONE Application Server	82
Software Requirements	82
Preparing for C++ Client Development	82
Assumptions and Limitations	84
Creating a C++ Client	84
Generating the IDL Files	84
Generating CPP Files from IDL Files	88
Sample Applications	91

About This Document

This guide describes how to create and run Java(2) Platform, Enterprise Edition™ (J2EE) clients that access Enterprise JavaBeans™ (EJBs) on Sun™ 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](#)
- [Using the Documentation](#)
- [How This Guide Is Organized](#)
- [Reference Information](#)
- [Documentation Conventions](#)
- [Product Support](#)

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™ 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 7, Enterprise Edition manuals are available as online files in Portable Document Format (PDF) and Hypertext Markup Language (HTML).

The following table lists tasks and concepts described in the Sun ONE Application Server manuals.

Table 1 Sun ONE Application Server Documentation Roadmap

For information about	See the following
Late-breaking information about the software and the documentation.	<i>Release Notes</i>
Comprehensive, table-based summary of supported hardware, operating system, JDK, and JDBC/RDBMS.	<i>Platform Summary</i>
Sun ONE Application Server 7 overview, features available with each product edition.	<i>Product Overview</i>
Diagrams and descriptions of server architecture, benefits of the Sun ONE Application Server architectural approach.	<i>Server Architecture</i>
New enterprise, developer, and operational features of Sun ONE Application Server 7.	<i>What's New</i>
How to get started with the Sun ONE Application Server 7 product. Includes new features, architectural overview, and sample application tutorial.	<i>Getting Started Guide</i>
Installing the Sun ONE Application Server software and its components, such as sample applications, the Administration interface, and the high-availability components. Instructions for implementing a basic high-availability configuration are included.	<i>Installation Guide</i>
Evaluating your system needs and enterprise to ensure that you deploy Sun ONE Application Server in a manner that best suits your site. General issues and concerns that you must be aware of when deploying an application server are also discussed.	<i>System Deployment Guide</i>
Best practices for HTTP session availability that application architects and developers can use.	<i>Application Design Guidelines for Storing Session State</i>

Table 1 Sun ONE Application Server Documentation Roadmap (*Continued*)

For information about	See the following
Creating and implementing Java™ 2 Platform, Enterprise Edition (J2EE™ platform) applications intended to run on the Sun ONE Application Server 7 that follow the open Java standards model for J2EE components such as servlets, Enterprise JavaBeans™ (EJBs™), and JavaServer Pages™ (JSPs™). Includes general information about application design, developer tools, security, assembly, deployment, debugging, and creating lifecycle modules. A comprehensive Sun ONE Application Server glossary is included.	<i>Developer's Guide</i>
Creating and implementing J2EE web applications that follow the Java™ Servlet and JavaServer Pages (JSP) specifications on the Sun ONE Application Server 7. Discusses web application programming concepts and tasks, and provides sample code, implementation tips, and reference material. Topics include results caching, JSP precompilation, session management, security, deployment, SHTML, and CGI.	<i>Developer's Guide to Web Applications</i>
Creating and implementing J2EE applications that follow the open Java standards model for enterprise beans on the Sun ONE Application Server 7. Discusses Enterprise JavaBeans (EJB) programming concepts and tasks, and provides sample code, implementation tips, and reference material. Topics include container-managed persistence, read-only beans, and the XML and DTD files associated with enterprise beans.	<i>Developer's Guide to Enterprise JavaBeans Technology</i>
Creating Application Client Container (ACC) clients that access J2EE applications on the Sun ONE Application Server 7.	<i>Developer's Guide to Clients</i>
Creating web services in the Sun ONE Application Server environment.	<i>Developer's Guide to Web Services</i>
Java™ Database Connectivity (JDBC™), transaction, Java Naming and Directory Interface™ (JNDI), Java™ Message Service (JMS), and JavaMail™ APIs.	<i>Developer's Guide to J2EE Services and APIs</i>
Creating custom NSAPI plug-ins.	<i>Developer's Guide to NSAPI</i>
Information and instructions on the configuration, management, and deployment of the Sun ONE Application Server subsystems and components, from both the Administration interface and the command-line interface. Topics include cluster management, the high-availability database, load balancing, and session persistence. A comprehensive Sun ONE Application Server glossary is included.	<i>Administrator's Guide</i>
Editing Sun ONE Application Server configuration files, such as the <code>server.xml</code> file.	<i>Administrator's Configuration File Reference</i>
Configuring and administering security for the Sun ONE Application Server operational environment. Includes information on general security, certificates, and SSL/TLS encryption. HTTP server-based security is also addressed.	<i>Administrator's Guide to Security</i>

Table 1 Sun ONE Application Server Documentation Roadmap (*Continued*)

For information about	See the following
Configuring and administering service provider implementation for J2EE™ Connector Architecture (CA) connectors for the Sun ONE Application Server 7. Topics include the Administration Tool, Pooling Monitor, deploying a JCA connector, and sample connectors and sample applications.	<i>J2EE CA Service Provider Implementation Administrator's Guide</i>
Migrating your applications to the new Sun ONE Application Server 7 programming model, specifically from iPlanet Application Server 6.x and from Netscape Application Server 4.0. Includes a sample migration.	<i>Migrating and Redploying Server Applications Guide</i>
How and why to tune your Sun ONE Application Server to improve performance.	<i>Performance Tuning Guide</i>
Information on solving Sun ONE Application Server problems.	<i>Troubleshooting Guide</i>
Messages that you may encounter while running Sun ONE Application Server 7. Includes a description of the likely cause and guidelines on how to address the condition that caused the message to be generated.	<i>Error Message Reference</i>
Utility commands available with the Sun ONE Application Server; written in manpage style.	<i>Utility Reference Manual</i>
Using the Sun™ Open Net Environment (Sun ONE) Message Queue software.	The Sun ONE Message Queue documentation at: http://docs.sun.com/db?prod/sl.slmsgqu

How This Guide Is Organized

This guide provides instructions for the development, assembly, and the deployment 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, *Index* is provided.

Reference Information

The following additional reading is recommended:

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).
- **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:

```
/opt/SUNWappserver7
```

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:

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

Conventions Referring to Directories

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

- **For Solaris 8 and 9 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 general feedback on the product or documentation, please send this to `appserver-feedback@sun.com`.

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/supporttraining/`
- 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

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 Services and APIs*.

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™ 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 Services and APIs*.

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 JavaTM 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:

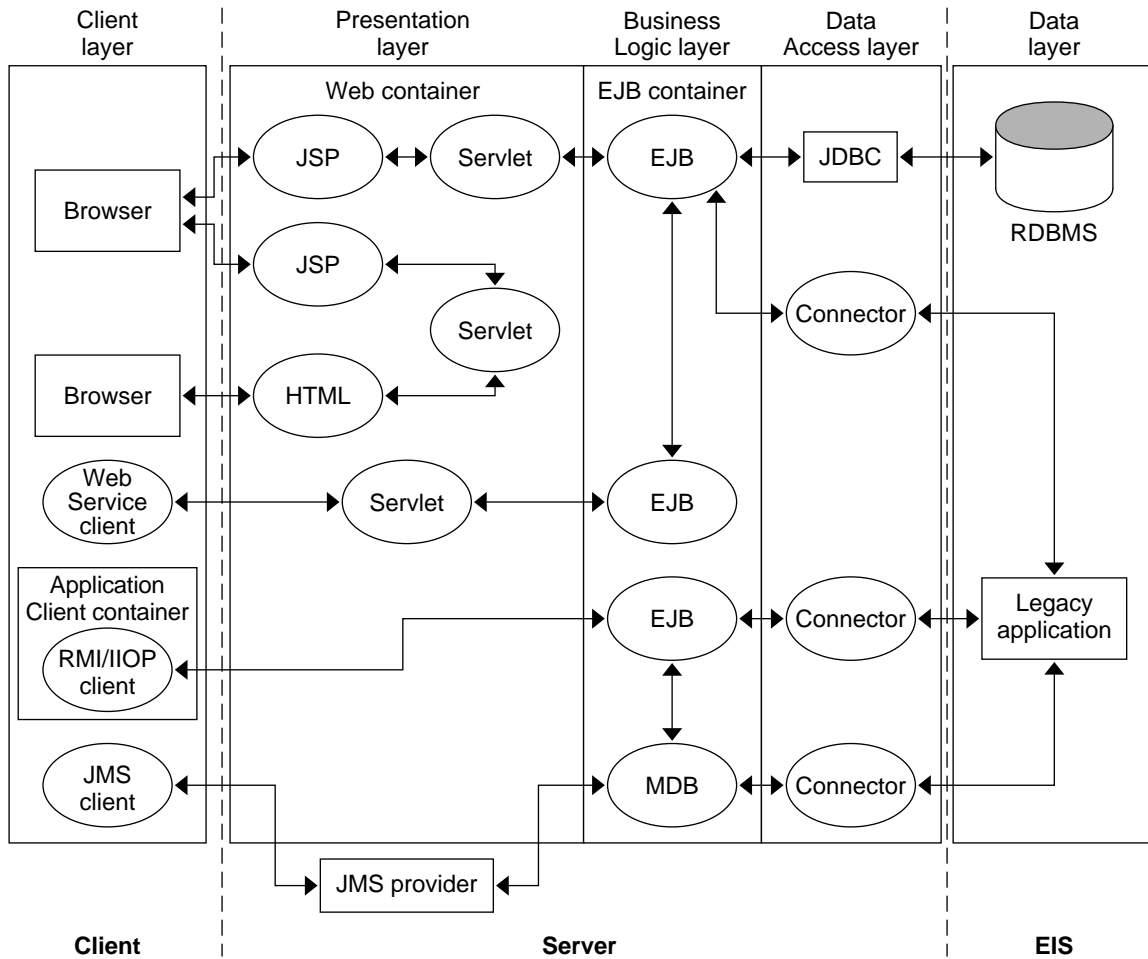
- JNDI is not supported. However, you can build name translations and do lookups using standard COSNaming binding.
- SSL over RMI/IIOP is not supported.
- Features that are configurable in the `sun-application-client.xml` and `sun-acc.xml` files are not available.

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

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](#)
- [Developing Applications Using the ACC](#)
- [Application Client Deployment Descriptors](#)

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 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™ Authentication and Authorization Service (JAAS) module. See [“Authenticating an Application Client Using the JAAS Module” on page 37](#).

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](#)
- [Using an Application Client to Access JMS Resources](#)
- [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 the Java Naming and Directory Interface™ (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/RMConverter");
```

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 `application-client.xml` file. The deployer provides the JNDI name for the `<ejb-ref>` in the corresponding `sun-application-client.xml` file.

For more information on the `sun-application-client.xml` file, see [“Sun ONE Application Client Deployment Descriptor” on page 53](#).

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:
 - o a Java class to access the bean
 - o `application-client.xml`
 - o `sun-application-client-.xml`
 - o 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
```

- o `-client` is required and specifies the name and location of the application client jar file.
 - o `-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.
 - o `-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.
 - o `-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.
 - o `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` file.

- 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*.

Using an Application Client to Access JMS Resources

This section describes the procedure to develop an application client that can access JMS resources to send a JMS message to a destination. The following two scenarios are discussed:

- [Application Client Accessing JMS Resources Without Using the ACC](#)
- [Application Client Packaged in an Application Client Container Accessing JMS Resources](#)

Before creating the client application, you must create JMS resources on the server. For information on creating JMS resources, see the *Sun ONE Application Server Developer's Guide to J2EE Services and APIs*.

Application Client Accessing JMS Resources Without Using the ACC

A stand-alone client uses the RMI/IIOP standard to communicate with Sun ONE Application Server. The J2EE 1.3 specification requires that a stand-alone client operate within the ACC context. However, Sun ONE Application Server allows Java platform clients to directly access the resources residing on the server. This section describes how you can develop a stand-alone client that can access the JMS resources directly without using the ACC path.

The sample application `SimpleQueueClient.java` is used here to describe the steps involved in developing a stand-alone client that looks up the JMS resources outside ACC and also send and receive messages to a queue on Sun ONE Application Server.

To create an application client:

1. Import the JMS packages.

```
import javax.jms.*;
import javax.naming.*;
```

2. Create an initial context.

```
Context initialContext = new InitialContext();
```

Do not pass any environment properties to the `InitialContext` constructor. Instead, obtain the ORBhost name and port number through the command line options.

3. Look up the Queue by its JNDI name. Use the `jms/sampleQCF` string to lookup the JMS destination.

```
private static final String LOOKUP_STRING_FACTORY =
    "jms/sampleQCF";

private static final String LOOKUP_STRING_QUEUE = "jms/sampleQ";

factory = (QueueConnectionFactory)
    initialContext.lookup(LOOKUP_STRING_FACTORY);

queue = (Queue) initialContext.lookup(LOOKUP_STRING_QUEUE);
```

`InitialContext` method is used to retrieve administered objects.

4. To send and receive messages, you must follow the procedure to create a JMS client:

- Create a `QueueConnection` to the message service. The connection provides access to the underlying transport of the message, and is also used to create sessions. Use the `CreateQueueConnection()` method on the factory object to create a connection.
- Start the connection. Unless the connection is started, `MessageConsumers` associated with the messages cannot receive any messages.
- Create a `QueueSession`. Sessions provide context for producing and consuming messages. Sessions are used to create message producers and message consumers, as well as build message themselves.
- Create message producers. Use the session and destination to create a message producer. In this example, a `QueueSender` is created.
- Create message consumers. Use the session and destination to create message consumer. In this example, a `QueueReceiver` is created.
- Build a message. Use session to create an empty message and add the data.

- Send the message. The message is passed to the send method on the QueueSender.
- Receive the message. Use the QueueReceiver method to receive the message.
- Retrieve the message contents. Call the receive method with a timeout argument (in milliseconds) greater than 0.
- Close all JMS resources.

For detailed instructions on developing a JMS client, see the *Sun ONE Application Server Developer's Guide to J2EE Services and APIs*.

5. Next, configure JMS resources on Sun ONE Application Server. You can either use the Administration Interface or the command line options to configure the resources.

You need to configure the following General properties:

- jmshost - Application Server host name
- adminusr - Admin instance user name
- adminpwd - Admin instance password
- adminport - Admin instance port number

Configure the following Connection Factory and Destination resource.

Connection Factory:

- JNDI Name: jms/sampleQCF
- Resource Type: javax.jms.QueueConnectionFactory

Destination Resource:

- JNDI Name: jms/sampleQ
- Resource Type: javax.jms.Queue

For information on configuring JMS resources, see the *Sun ONE Application Server Administrator's Guide*.

NOTE	You do not have to deploy this application on an ACC or Sun ONE Application Server as it is a stand-alone client.
-------------	---

6. Run the client.

- a. Set the environment variable LD_LIBRARY_PATH. This variable should point to the Application Server, the Sun ONE MQ jar files and shared libraries:

```
LD_LIBRARY_PATH=/usr/lib/mps:/opt/SUNWappserver7/lib:/usr/lib
```

If the Application Server is on a different system, copy all the jar files and shared libraries from the /opt/SUNWappserver7/lib, /usr/share/lib/imq and /usr/lib/mps directories to the target system.

- b. Before running the client, set the values for the Java Virtual Machine startup options:

```
jvmarg value = "-Dorg.omg.CORBA.ORBInitialHost=${ORBhost}"
```

```
jvmarg value = "-Dorg.omg.CORBA.ORBInitialPort=${ORBport}"
```

Here ORBhost is the Application Server hostname and ORBport is the ORB port number (default 3700 for server1 instance).

- c. Run the client.

The code of the sample application is given below:

```
package samples.jms.client;

import javax.jms.*;
import javax.naming.*;
import java.io.IOException;
import java.util.*;

public class SimpleQueueClient {

    private QueueConnectionFactory factory;
    private Queue queue;

    private static final String LOOKUP_STRING_FACTORY =
"jms/sampleQCF";
    private static final String LOOKUP_STRING_QUEUE =
"jms/sampleQ";

    public static void main(String[] args) throws Exception
    {
        SimpleQueueClient client = new SimpleQueueClient ( );
        client.execute();
    }
}
```

```

public SimpleQueueClient ( ) throws Exception {
    try {
        // create the initial context
        Context initialContext = new InitialContext();

        out("Looking up the queue connection factory from JNDI
        :"+LOOKUP_STRING_FACTORY);

        // look up the connection factory from the object store
        factory = (QueueConnectionFactory)
        initialContext.lookup(LOOKUP_STRING_FACTORY);
        out(LOOKUP_STRING_FACTORY + ": " + factory);

        // look up queue from the object store
        out("Looking up the queue from JNDI");
        queue = (Queue) initialContext.lookup(LOOKUP_STRING_QUEUE);
        out(LOOKUP_STRING_QUEUE + ": " + queue);
    }

    catch (NamingException e) {
        String msg = "An error was encountered trying to lookup an
        object from JNDI";
        out(msg);
        e.printStackTrace();
    }
}

public void execute()
    throws IOException {
    final StringmessageBody = "This is a sample message. It was "
    + "sent at " + new Date();

    QueueSession          session          =null;
    QueueConnection        connection      =null;
    QueueSender            queueSender     = null;
    QueueReceiver          queueReceiver   = null;
    String                 successText     ="SUCCESSFUL";
    TextMessage            msgReceived     =null;

    try {
        // Creating a QueueConnection to the Message service
        out("Creating QueueConnection using the factory");
    }
}

```

```

connection = factory.createQueueConnection();
    out("Starting the Connection");
    connection.start();

// Creating a session within the connection
session =
connection.createQueueSession(false,Session.AUTO_ACKNOWLEDGE)
;
out("Creating a QueueSender");
queueSender = session.createSender(queue);
out("Creating a QueueReceiver");
queueReceiver = session.createReceiver(queue);

// Building a message text

    out("Building a message" );
    TextMessage msgSent = session.createTextMessage();
    msgSent.setText(messageBody);

// Sending message to the target queue

    out("Sending message to " + queue.getQueueName() );
    queueSender.send(msgSent);
    out( "Waiting for the return message" );

/* comment the following line to leave the message on the queue.
then use the message queue product's admin tools to verify that the
message was placed on the queue.
*/

// Retrieving the next message that arrives within the timeout
interval of 2000 milliseconds

    msgReceived = (TextMessage)
        queueReceiver.receive(2000);

    if (msgReceived == null) {
        out("An error has occurred. The return message was
not received.");
        successText = "UNSUCCESSFUL";
    } else {

//Retreive the contents of the message.
if (msgReceived instanceof TextMessage) {

        TextMessage txtMsg = (TextMessage) msgReceived;
        out("\nMessage received: " + txtMsg.getText());

    }

}

```

```

    }

    catch (JMSEException e) {

        out("An unexpected exception occurred: " + e);
        Exception linkedException = e.getLinkedException();
        if (linkedException != null) {
            out("The linked exception is: " + linkedException);
        }

        e.printStackTrace();
        successText = "UNSUCCESSFUL";
    } finally {

        // Close all JMS resources
        if (queueReceiver != null) {
            try {
                out("Closing QueueReceiver");
                queueReceiver.close();
            } catch (JMSEException e) {
                out("There was an error closing the receiver");
                e.printStackTrace();
            }
        }

        if (queueSender != null) {
            try {
                out("Closing QueueSender");
                queueSender.close();
            } catch (JMSEException e) {
                out("There was an error closing the sender");
                e.printStackTrace();
            }
        }

        if (session != null) {
            try {
                out("Closing session");
                session.close();
            } catch (JMSEException e) {

```



```

        out("There was an error closing the session");
        e.printStackTrace()
    }
}

    if (connection != null) {
        try {
            out("Closing connection");
            connection.close();
        } catch (JMSEException e) {
            out("There was an error closing the
            connection");
            e.printStackTrace();
        }
    }
    destroy();
}

}

public void destroy() {
    factory = null;
    queue = null;
}

private void out(String message) {
    System.out.println(message);
}

}

```

Application Client Packaged in an Application Client Container Accessing JMS Resources

When the application client is packaged in an application client container, make the following changes to the code. In the sample application

`SimpleQueueClient.java`, make the following changes:

1. A J2EE application can be packaged using the Application Client Container. Use the `java:comp/env/jms/` string to lookup the JMS resources. This is the J2EE application namespace.

```
private static final String LOOKUP_STRING_FACTORY =
    "java:comp/env/jms/sampleQCF";

private static final String LOOKUP_STRING_QUEUE    =
    "java:comp/env/jms/sampleQ";
```

2. The Application Client Container gets the ORB hostname and port number from the ACC configuration file `sun-acc.xml`.

The `<name>` entry in this file will be the Application Server hostname and the port is the ORB port number (default 3700 for server1 instance).

3. Assemble the application client to create a jar file. Include the two configuration files in the client jar file.

`sun-application-client.xml` - Sun ONE Application Server specific J2EE client application

For information on `sun-application-client.xml` file, see [“Sun ONE Application Client Deployment Descriptor” on page 53](#).

The contents of the `sun-application-client.xml` is as follows:

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

<!DOCTYPE sun-application-client PUBLIC "-//Sun Microsystems,
Inc.//DTD Sun ONE Application Server 7.0 Application Client
1.3//EN"

'http://www.sun.com/software/sunone/appserver/dtds/sun-applicati
on-client_1_3-0.dtd'>

<sun-application-client>

    <resource-ref>

        <res-ref-name>jms/sampleQ</res-ref-name>
        <jndi-name>jms/sampleQ</jndi-name>

    </resource-ref>

    <resource-ref>

        <res-ref-name>jms/sampleQCF</res-ref-name>
        <jndi-name>jms/sampleQCF</jndi-name>

    </resource-ref>

</sun-application-client>
```

application-client.xml - J2EE 1.3 application client deployment descriptor

The contents of the application-client.xml is as follows:

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

<!DOCTYPE application-client PUBLIC "-//Sun Microsystems,
Inc.//DTD J2EE Application Client 1.3//EN"
'http://java.sun.com/dtd/application-client_1_3.dtd'>

<application-client>

    <display-name>SimpleQueue</display-name>

    <resource-ref>

        <res-ref-name>jms/sampleQ</res-ref-name>

        <res-type>javax.jms.Queue</res-type>

        <res-auth>Container</res-auth>

    </resource-ref>

    <resource-ref>

        <res-ref-name>jms/sampleQCF</res-ref-name>

        <res-type>javax.jms.QueueConnectionFactory</res-type>

        <res-auth>Container</res-auth>

    </resource-ref>

</application-client>
```

These deployment descriptors describe the external JMS resources (administered objects) referenced by the sample application.

4. Package the application client using the appclient script. See [“Packaging an Application Client Using the ACC” on page 45](#).

package-appclient script also creates a MANIFEST file that contains the main class, which states the complete package prefix and classname of the Java platform client.

5. Run the application client using the ACC. For instructions, see [“Running an Application Client Using the ACC” on page 49](#).

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.

Solaris:

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

- o libcis.so
- o libnspr4.so
- o libplc4.so
- o libnss3.so
- o 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:

- o appserv-ext.jar
- o appserv-rt.jar
- o fscontext.jar
- o imq.jar
- o imqadmin.jar
- o 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 void main(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(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.cosnaming.CNCtxFactory");
env.put(Context.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,

- o *CP* is the CLASSPATH which includes the application client jar file and the `appserv-ext.jar`.
- o *ClientApp* refers to the client program.
- o *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. `LoginModules` 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;
}
```

- o The parameter `subject`, is the subject to be authenticated.
- o `callbackHandler`, for communicating with the end user which prompts for the username and password.
- o `sharedState`, is the shared `LoginModule` state.
- o `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 64](#).

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 45](#).

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 = null;
                }
                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;
}
}

```

NOTE Sun ONE Application Server does not support authentication of RMI/IIOP Clients that do not use the ACC (non-ACC Clients).

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=/install_dir/appclient`.
- `$AS_NSS` to reference the location of the nss libs.

For example:

UNIX:

```
$AS_NSS=/install_dir/appclient/lib
```

- `$AS_JAVA` to reference the location where you have installed the JDK.
- `$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: *instance_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.

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="gasol-e1" address="gasol-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 57](#).

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 57.](#)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE client-container SYSTEM
"file:///export3/sun/appserver7/appserv/lib/dtds/sun-application-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-container_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 the `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 <i>zero or more</i> of this subelement.
?	Can contain <i>zero or one</i> of this subelement.
+	Must contain <i>one or more</i> of this subelement.
(none)	Must contain <i>only one</i> 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
<code>resource-ref</code>	zero or more	Maps the absolute JNDI name to the <code>resource-ref</code> in the corresponding J2EE XML file.
<code>ejb-ref</code>	zero or more	Maps the absolute JNDI name to the <code>ejb-ref</code> in the corresponding J2EE XML file.
<code>resource-env-ref</code>	zero or more	Maps the absolute JNDI name to the <code>resource-env-ref</code> 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
<code>res-ref-name</code>	only one	Specifies the <code>res-ref-name</code> in the corresponding J2EE <code>application-client.xml</code> file.
<code>jndi-name</code>	only one	Specifies the absolute jndi name of a resource.
<code>default-resource-principal</code>	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
<code>name</code>	only one	Specifies the name of the principal.
<code>password</code>	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
<code>ejb-ref-name</code>	only one	Specifies the name of a ejb reference in the corresponding J2EE <code>appclient.xml</code> file.
<code>jndi-name</code>	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
<code>resource-env-ref-name</code>	only one	Specifies the <code>res-ref-name</code> in the corresponding J2EE <code>application-client.xml</code> file <code>resource-env-ref</code> entry.
<code>default-resource-principal</code>	only one	Specifies the default principal (user) that the container uses to access a resource.
<code>jndi-name</code>	only one	Specifies the <code>jndi-name</code> 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
<code>target-server</code>	zero or more	Specifies the IIOP listener configuration of the target server.
<code>auth-realm</code>	only one	Specifies the optional configuration for JAAS authentication realm.
<code>client-credential</code>	only one	Specifies the default client credential that will be sent to the server.
<code>log-service</code>	only one	Specifies the default log file and the severity level of the message.
<code>property</code>	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
<code>sendPassword</code>	<code>none</code>	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
<code>description</code>	zero or more	Specifies the description of the target server.
<code>security</code>	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 <i>Sun ONE Application Server Administrator's Guide</i> 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
<code>user-name</code>	none	The user name used to authenticate the Application client container.
<code>password</code>	none	The password used to authenticate the Application client container.
<code>realm</code>	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
<code>property</code>	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 <i>your_Acc_dir</i> /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
<code>cert-nickname</code>	<code>none</code>	The nickname of the server certificate in the certificate database or the PKCS#11 token. In the certificate, the name format is <i>tokenname:nickname</i> . Including the <i>tokenname</i> : part of the name in this attribute is optional.
<code>ssl2-enabled</code>	<code>none</code>	(Optional) Determines whether SSL2 is enabled.
<code>ssl3-enabled</code>	<code>none</code>	(Optional) Determines whether SSL3 is enabled.
<code>ssl2-ciphers</code>	<code>none</code>	(Optional) A space-separated list of the SSL2 ciphers used with the prefix <code>+</code> to enable or <code>-</code> to disable. For example, <code>+rc4</code> . Allowed values are <code>rc4</code> , <code>rc4export</code> , <code>rc2</code> , <code>rc2export</code> , <code>idea</code> , <code>des</code> , <code>desede3</code> .
<code>ssl3-tls-ciphers</code>	<code>none</code>	(Optional) A space-separated list of the SSL3 ciphers used, with the prefix <code>+</code> to enable or <code>-</code> to disable, for example <code>+rsa_des_sha</code> . Allowed SSL3 values are <code>rsa_rc4_128_md5</code> , <code>rsa_des_sha</code> , <code>rsa_rc4_40_md5</code> , <code>rsa_rc2_40_md5</code> , <code>rsa_null_md5</code> . Allowed TLS values are <code>rsa_des_56_sha</code> , <code>rsa_rc4_56_sha</code> .
<code>tls-enabled</code>	<code>none</code>	Determines whether TLS is enabled.
<code>tls-rollback-enabled</code>	<code>none</code>	Determines whether TLS rollback is enabled. TLS rollback should be enabled for Microsoft Internet Explorer 5.0 and 5.5.
<code>client-auth-enabled</code>	<code>none</code>	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
<code>cert-db-path</code>	none	Specifies the absolute path of the certificate database (<code>cert7.db</code>).
<code>cert-db-password</code>	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.ipplanet.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
<code>property</code>	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
<code>auth-realm-name</code>	<code>none</code>	Defines the name of this realm.
<code>classname</code>	<code>none</code>	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
<code>name</code>	<code>none</code>	Specifies the name of the property.
<code>value</code>	<code>none</code>	Specifies the value of the property.

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](#)

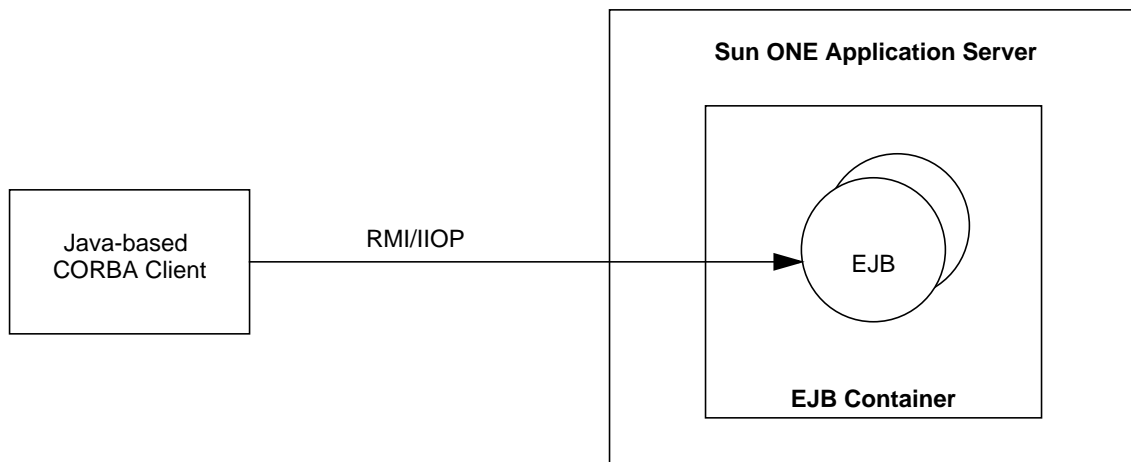
CORBA Client Scenarios

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 Application Server . This section describes the following scenarios:

- [Stand-alone Scenario](#)
- [Server to Server Scenario](#)

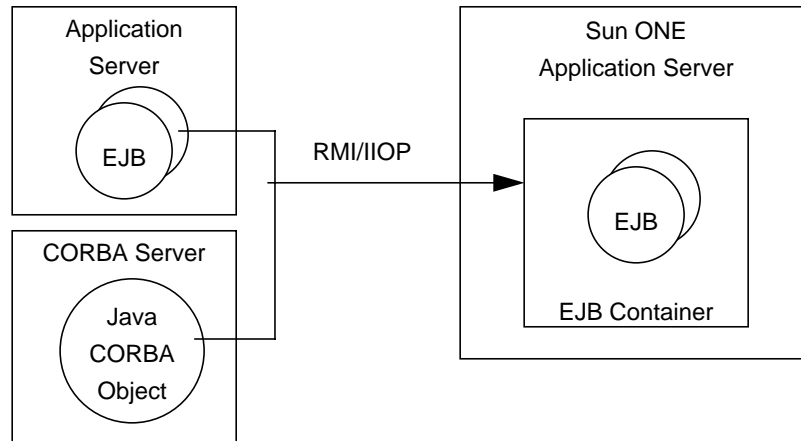
Stand-alone Scenario

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”](#) on page 68.

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

Server to Server Scenario

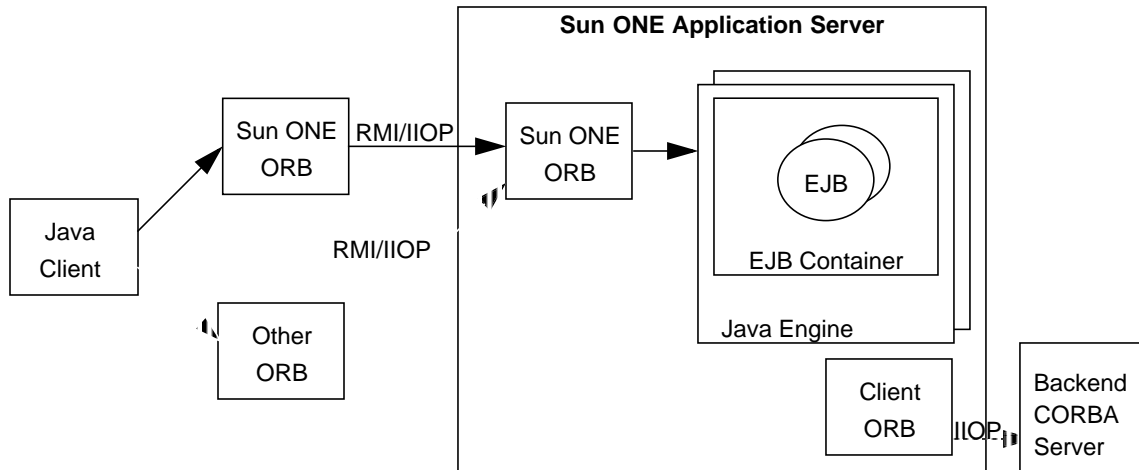
CORBA objects, and other application servers can use IIOP to access EJB components housed in Application Server, as shown in the figure [“Application Server and CORBA Objects Accessing EJB Components”](#) on page 69.

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

ORB Support Architecture

CORBA client support in 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” on page 70](#).

Figure 3-3 ORB Support Architecture



You can use the ORB that is bundled as part of the 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.CNContextFactory` 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 `SIASCtxFactory` class:

```
Properties env = new Properties();

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

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.SLASCtxFactory");
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 `SLASCtxFactory` 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 `SLASCtxFactory` 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 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 JNDI 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. You can find the exact JNDI name of the bean by looking at the `sun-ejb-jar.xml` deployment descriptor of the EJB. The JNDI name of the bean is represented by the `<jndi-name>` element under the `<ejb>` element.

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

For example, here is a lookup using the value `rmiConverter`:

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

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

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/iona/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.ionacorba.art.artimpl.ORBImpl");
orbProperties.put("org.omg.CORBA.ORBSingletonClass", "com.ionacorba.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.ionacorb.art.artimpl.ORBImpl
org.omg.CORBA.ORBSingletonClass=com.ionacorb.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.ionacorb.art.artimpl.ORBImpl
</jvm-options>
<jvm-options>
    -Dorg.omg.CORBA.ORBSingletonClass=com.ionacorb.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/

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](#)
- [Developing a C++ Client](#)

Introducing C++ Clients

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 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™ 2 Platform, Standard Edition (J2SE™ platform) 1.4

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 82](#).
2. Install Java Development Kit (JDK) 1.4.

3. Install ORBacus 4.1.

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

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
```

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/*
-

4. Install Application Server and test for basic functionality.

5. 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 Application Server or independent of 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™ 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/BookCartApp
Ejb_jar:install_dir/lib/appserv-ext.jar
-idl samples.rmi_iiop.cpp.ejb.Cart
```

- III. `rmic -classpath`

```
instance_dir/applications/j2ee-apps/BookCartApp_1/BookCartApp
Ejb_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. -Iclasspath to IDL files -Iorbacus_home/idl/  
-Iorbacus_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 from 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.

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(),complexObjectVf);
```

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.
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/NameService
```

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/

Index

A

ACC

- features 22
- naming 22
- security 22

acc 21

acc flag 37

acc package

- asenv configuration settings 45
- editing sun-acc.xml 46
- modifying appclient script 46
- using package-appclient script 47

appclient.jar file 48

- contents 48

application client 19

- accessing EJB 25
- appclient script 49
- create bean instance 24
- creating using the ACC 23
- invoke business method 24
- invoking an EJB module 24
- locate EJB home interface 23
- making a remote call 26
- running 49
- using SSL with CA 47

application client container 21

application client container package

- client.policy file 48

application clients

- authenticating using JAAS 37
- security 37

application-client.xml 53

ATTLIST tag 52

attributes

- #IMPLIED label 52
- #REQUIRED label 52

authentication realm 64

C

c++ clients 81

- configuring 82
- developing 84
- preparing for development 82
- required classes 83
- running 91

client 15, 58

- architecture 19
- web services clients 17

client types 17

clients

- application clients 19
- CORBA clients 18
- JMS clients 18
- RMI-IIOP clients 18
- web client 17
- web services clients 17

client-side load balancing 72

configure to use orbix 76

configuring Sun ONE ORB 72

CORBA clients 18

- scenarios 67

- cpp files 88
- create bean instance
 - create method 24

D

- deployment descriptors 50
 - application client container 57
 - attributes 52
 - data 52
 - element 51
 - format 50
 - J2EE application client 53
 - subelements 51
 - Sun ONE application client 53
- developing c++ clients
 - generate cpp files 88
 - generate IDL files 84, 85
 - generate valuetypes 86
 - implementing valuetypes 90
 - modifying the generated IDL files 87
 - registering valuefactories 90

E

- EJBs
 - accessing with IIOP 67
 - specifying JNDI name 71, 75

F

- form-hint-field attribute 59

I

- IDL files
 - generate 85
 - rmic tool 85

- IIOP 18
 - accessing EJBs 67
 - accessing servers 68
- IIOP listener configuration 59
- IIOP/SSL configuration 62
- InitialContext 71, 75
- invoking a J2EE client without using acc 36

J

- J2EE application client 23
- J2EE platform layers 15
 - Business logic layer 16
- client 15
- database 16
- presentation 15
- J2SE policy file 48
- JAAS module 37
 - LoginModule 37, 38
- JMS clients 18
- JNDI 18
 - specifying EJB name 71, 75
- JVM arguments in server.xml 78

L

- launching acc 49
- library for valuetype implementation
 - developing 86
- load balancing 72
- logging messages 46
- LoginModule
 - CallBackHandler 39
 - commit() method 39
 - integrate 40
 - login() method 38
 - logout() method 39

M

message-driven beans [18](#)
 modifying the generated IDL files
 changing valuetypes [87](#)
 deleting duplicate variables [87](#)

N

naming factory class [71](#), [75](#)

O

ORB architecture [69](#)
 overriding built-in ORB [77](#)
 approaches [77](#)
 ORB.init properties approach [77](#)
 ORB.init() properties [77](#)
 orb.properties [78](#)
 orb.properties approach [78](#)
 provide JVM start-up arguments [78](#)
 providing JVM arguments [78](#)

P

param-name element [55](#)
 presentation layer
 J2EE components [15](#)
 non-J2EE components [16](#)

R

RMI/IIOP [18](#)

S

S1ASCtxFactory class [72](#)
 scenarios
 server-server [68](#)
 stand-alone [67](#)
 security
 authentication data [22](#)
 JAAS module [22](#)
 using SSL with CA [47](#)
 setting the ORB port [46](#)
 singleton approach [78](#)
 SSL [22](#)
 SSL processing parameters [62](#)
 stand-alone CORBA client
 creating [70](#)
 running [73](#)
 subelements
 requirement rules [51](#)
 Sun customer support [13](#)
 Sun ONE ORB [81](#)
 sun-acc.xml elements
 auth-realm [64](#)
 cert-db [64](#)
 client-container [58](#)
 client-credential [60](#)
 description [60](#)
 log-service [61](#)
 property [65](#)
 security [62](#)
 ssl [62](#)
 target-server [59](#)
 sun-acc.xml file [57](#)
 elements in [57](#)
 sun-application element
 definition in sun-application_1_3-0.dtd file [51](#)
 sun-application-client.xml [53](#)
 sun-application-client.xml elements
 default-resource-principal [55](#)
 ejb-ref [56](#)
 ejb-ref-name [56](#)
 jndi-name [57](#)
 name [56](#)
 password [56](#)
 resource-env-ref [56](#)

- resource-env-ref-name [57](#)
- resource-ref [54](#)
- resource-ref-name [55](#)
- sun-application-client [54](#)
- sun-application-client.xml file [54](#)
 - elements in [53](#)

T

- thin client [17](#)
- third party ORB [74](#)
 - accessing another server instance [74](#)
 - accessing backend [76](#)
 - configure Orbix ORB [76](#)

W

- web client [17](#)
- web services clients [17](#)