Contents

About This Guide **3**

Who Should Use This Guide 3
Using the Documentation 4
How This Guide Is Organized 6
Related Information 7
Documentation Conventions 7
General Conventions 7
Conventions Referring to Directories 9
Contacting Sun 9
Give Us Feedback 9
Obtain Training 9
Contact Product Support 10

Chapter 1 Overview of Clients **11**
Introducing Clients 11
Types of Clients 13
Web Clients 13
Web Services Clients 13
JMS Clients 14
CORBA Clients 14
Application Clients 15

Chapter 2 Using the Application Client Container **17**
Introducing the Application Client Container 17
Application Client Container Features 18
Developing Application Clients 18
Creating an Application Client 19

Locating the Home Interface 19
Creating an Enterprise Bean Instance 20
Invoking a Business Method 20
Creating an ACC Client With Load Balancing and Failover Support (Enterprise Edition) 20
Introducing the Properties that Support LB/FO for ACC Clients 21
Configuration Changes 21
Using an Application Client to Invoke an EJB Module 22
Making a Remote Call on the EJB 23
Using an Application Client to Access JMS Resources 23
Application Client Accessing JMS Resources Without Using the ACC 24
Application Client Packaged in an Application Client Container Accessing JMS Resources . 32
Authenticating an Application Client Using the JAAS Module 34
Invoking an RMI/IIOP-based Client Without Using the ACC 42
Packaging an Application Client Using the ACC 44
Editing the Configuration File 44
Editing the appclient Script 44
Editing the sun-acc.xml File 45
Setting Security Options 45
Using the package-appclient Script 46
Running an Application Client Using the ACC 47
Sample Client Application 48

Chapter 3 Application Client Deployment Descriptors 49
Introducing Application Client Deployment Descriptors 49
Format of Deployment Descriptors 50
Subelements 50
Data 51
Attributes 51
J2EE Application Client Deployment Descriptor 52
Sun Java System Application Client Deployment Descriptor 52
Elements in sun-application-client.xml file 52
Application Client Container Configuration File 57
Elements in the sun-acc.xml File 57

Chapter 4 Java-based CORBA Clients 67
CORBA Client Scenarios 67
Stand-alone Scenario 67
Server to Server Scenario 68
ORB Support Architecture 69
Developing non-ACC 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
Implementing Load Balancing and Failover Capabilities in the Client Application (Enterprise Edition) 72
Running a Stand-alone CORBA Client 75

Chapter 5 C++ Clients 77
Introducing C++ Clients 77
Developing a C++ Client 77
Configuring C++ Clients to Access Sun Java System Application Server 78
Software Requirements 78
Preparing for C++ Client Development 78
Assumptions and Limitations 80
Creating a C++ Client 80
Generating the IDL Files 80
Generating CPP Files from IDL Files 84
Sample Applications 87

Index 89

Sun Java™ System

Application Server 7 Developer's Guide to Clients

2004Q2 Update 1

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

Part No: 819-0595

Copyright © 2004 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at http://www.sun.com/patents and one or more additional patents or pending patent applications in the U.S. and in other countries.

THIS PRODUCT 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, Solaris, Sun™ ONE, Sun™ ONE Studio, iPlanet, J2EE, J2SE, Enterprise JavaBeans, EJB, JavaServer Pages, JSP, JDBC, JDK, JVM, Java Naming and Directory Interface, and JavaMail are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon architecture developed by Sun Microsystems, Inc.

UNIX is a registered trademark in the U.S. and in 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.

DOCUMENTATION IS PROVIDED “AS IS” AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright © 2004 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, Etats-Unis. Tous droits réservés.

Sun Microsystems, Inc. détient les droits de propriété intellectuels relatifs à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et ce sans limitation, ces droits de propriété intellectuelle peuvent inclure un ou plus des brevets américains listés à l’adresse http://www.sun.com/patents et un ou les brevets supplémentaires ou les applications de brevet en attente aux Etat -Unis et dans les autres pays.

CE PRODUIT 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, ECRITE ET PREALABLE DE SUN MICROSYSTEMS, INC.

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, Solaris, Sun™ ONE, Sun™ ONE Studio, iPlanet, J2EE, J2SE, Enterprise JavaBeans, EJB, JavaServer Pages, JSP, JDBC, JDK, JVM, Java Naming and Directory Interface, et JavaMail sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d’autres pays.

Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

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 règlementation 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.

LA DOCUMENTATION EST FOURNIE “EN L’ETAT” ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L’APTITUDE A UNE UTILISATION PARTICULIERE OU A L’ABSENCE DE CONTREFACON.

About This Guide

This manual describes how to create and implement Java™ 2 Platform, Enterprise Edition (J2EE™ platform) applications that follow the Enterprise JavaBean (EJB™) specification in the Sun Java™ System Application Server Standard and Enterprise Edition 7 2004Q2 environment. In addition to describing programming concepts and tasks, this guide offers sample code, implementation tips, and the reference material.

This preface addresses the following topics:

- Who Should Use This Guide
- Using the Documentation
- How This Guide Is Organized
- Related Information
- Documentation Conventions
- Contacting Sun

Who Should Use This Guide

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

This guide assumes you are familiar with the following topics:

- Java programming
- Java APIs as defined in the Java™ Servlet, JavaServer Pages™ (JSP™), Enterprise JavaBeans™ (EJB™), and Java™ Database Connectivity (JDBC™) specifications

- The SQL structured database query languages
- Relational database concepts
- Software development processes, including debugging and source code control

Using the Documentation

The Sun Java System Application Server Standard and 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 Java System Application Server manuals. The manuals marked *(updated for 7 2004Q2)* have been updated for the Sun Java System Application Server Standard and Enterprise Edition 7 2004Q2 release. The manuals not marked in this way have not been updated since the version 7 Enterprise Edition release.

Table 1 Sun Java System Application Server Documentation Roadmap

For information about	See the following
(Updated for 7 2004Q2) Late-breaking information about the software and the documentation. Includes a comprehensive, table-based summary of supported hardware, operating system, JDK, and JDBC/RDBMS.	*Release Notes*
Sun Java System Application Server 7 overview, including the features available with each product edition.	*Product Overview*
Diagrams and descriptions of server architecture and the benefits of the Sun Java System Application Server architectural approach.	*Server Architecture*
(Updated for 7 2004Q2) How to get started with the Sun Java System Application Server product. Includes a sample application tutorial. There are two guides, one for Standard Edition and one for Enterprise Edition.	*Getting Started Guide*
(Updated for 7 2004Q2) Installing the Sun Java System Application Server Standard Edition and Enterprise Edition software and its components, such as sample applications and the Administration interface. For the Enterprise Edition software, instructions are provided for implementing the high-availability configuration.	*Installation Guide*
(Updated for 7 2004Q2) Evaluating your system needs and enterprise to ensure that you deploy Sun Java System 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.	*System Deployment Guide*

Table 1 Sun Java System 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 Java System Application Server 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 Java System Application Server glossary is included.	*Developer's Guide*
(Updated for 7 2004Q2) Creating and implementing J2EE web applications that follow the Java™ Servlet and JavaServer Pages (JSP) specifications on the Sun Java System Application Server. 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.	*Developer's Guide to Web Applications*
(Updated for 7 2004Q2) Creating and implementing J2EE applications that follow the open Java standards model for enterprise beans on the Sun Java System Application Server. 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.	*Developer's Guide to Enterprise JavaBeans Technology*
(Updated for 7 2004Q2) Creating Application Client Container (ACC) clients that access J2EE applications on the Sun Java System Application Server.	*Developer's Guide to Clients*
Creating web services in the Sun Java System Application Server environment.	*Developer's Guide to Web Services*
(Updated for 7 2004Q2) Java™ Database Connectivity (JDBC™), transaction, Java Naming and Directory Interface™ (JNDI), Java™ Message Service (JMS), and JavaMail™ APIs.	*Developer's Guide to J2EE Services and APIs*
Creating custom NSAPI plug-ins.	*Developer's Guide to NSAPI*
(Updated for 7 2004Q2) Information and instructions on the configuration, management, and deployment of the Sun Java System 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 Java System Application Server glossary is included.	*Administration Guide*
(Updated for 7 2004Q2) Editing Sun Java System Application Server configuration files, such as the `server.xml` file.	*Administrator's Configuration File Reference*
Configuring and administering security for the Sun Java System Application Server operational environment. Includes information on general security, certificates, and SSL/TLS encryption. HTTP server-based security is also addressed.	*Administrator's Guide to Security*

Table 1 Sun Java System 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 Java System Application Server. Topics include the Administration Tool, Pooling Monitor, deploying a JCA connector, and sample connectors and sample applications.	*J2EE CA Service Provider Implementation Administrator's Guide*
(Updated for 7 2004Q2) Migrating your applications to the new Sun Java System Application Server programming model, specifically from iPlanet Application Server 6.x and Sun ONE Application Server 7.0. Includes a sample migration.	*Migrating and Redeploying Server Applications Guide*
(Updated for 7 2004Q2) How and why to tune your Sun Java System Application Server to improve performance.	*Performance Tuning Guide*
(Updated for 7 2004Q2) Information on solving Sun Java System Application Server problems.	*Troubleshooting Guide*
(Updated for 7 2004Q2) Information on solving Sun Java System Application Server error messages.	*Error Message Reference*
(Updated for 7 2004Q2) Utility commands available with the Sun Java System Application Server; written in manpage style.	*Utility Reference Manual*
Using the Sun™ Java System Message Queue 3.5 software.	The Sun Java System Message Queue documentation at: `http://docs.sun.com/db?p=prod/s1.s1msgqu`

How This Guide Is Organized

This guide provides instructions for the development, assembly, and the deployment of J2EE clients to Sun Java System Application Server.

- Chapter 1, "Overview of Clients"

  This chapter introduces you to various types of clients that are supported by Sun Java System 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, "Application Client Deployment Descriptors"

  This chapter describes the application deployment descriptors.

- Chapter 4, "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 5, “C++ Clients”

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

Finally, *Index* is provided.

Related 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). For Windows versions, the directory paths are the same, except that backslashes are used to separate directories.

- **URLs** are given in the format:

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

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

- **Font conventions** include:

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

- **Installation root directories** for most platforms are indicated by *install_dir* in this document. Exceptions are noted in "Conventions Referring to Directories" on page 9.

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

  - Solaris and Linux file-based installations:

    user's home directory`/sun/appserver7`

  - Windows, all installations:

    system drive`:\Sun\AppServer7`

  For the platforms listed above, *default_config_dir* and *install_config_dir* are identical to *install_dir*. See "Conventions Referring to Directories" on page 9 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*

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

Conventions Referring to Directories

By default, when using the Solaris package-based or Linux RPM-based installation, the application server files are spread across several root directories. 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.

Contacting Sun

You might want to contact Sun Microsystems in order to:

- Give Us Feedback
- Obtain Training
- Contact Product Support

Give Us Feedback

If you have general feedback on the product or documentation, please send this to `http://www.sun.com/hwdocs/feedback`

Obtain Training

Application Server training courses are available at:

`http://training.sun.com/US/catalog/enterprise/web_application.html/`

Visit this site often for new course availability on the Sun Java System Application Server.

Contact Product Support

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

- The online support web site at:

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

- The telephone dispatch number associated with your maintenance contract

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

- Description of the problem, including the situation where the problem occurs and its impact on your operation
- Machine type, operating system version, and product version, including any patches and other software that might be affecting the problem. Here are some of the commonly used commands:
  - **Solaris:** `pkginfo, showrev`
  - **Linux:** `rpm`
  - **All:** `asadmin version --verbose`
- Detailed steps on the methods you have used to reproduce the problem
- Any error logs or core dumps
- Configuration files such as:
  - *instance_dir*`/config/server.xml`
  - a web application's `web.xml` file,
    when a web application is involved in the problem
- For an application, whether the problem appears when it is running in a cluster or standalone

Chapter 1

Overview of Clients

A client can be a simple web browser or an application that runs on the client system. Sun Java System Application Server 7 2004Q2 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 Java System 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 Java System Application Server is a Java application server and is fully compliant with the J2EE 1.3 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 Java System 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 Java System 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 Java System 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 Java System 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 Java System Application Server Developer's Guide.*

Types of Clients

This section introduces the following types of clients that are supported by Sun Java System 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 Java System 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 Java System 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 Java Studio 4, which is bundled with Sun Java System Application Server.

For information on developing and deploying Web Services clients, see the *Sun Java System 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 Java System 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 Java System Application Server. The Application Server supports access to EJBs using the Internet Inter-ORB Protocol (IIOP) as specified in the Enterprise JavaBeans Specification, V2.0, and the Enterprise JavaBeans to CORBA Mapping Specification. These clients use Java Naming and Directory Interface (JNDI) to locate EJBs, and use Java™ Remote Method Invocation / Internet Inter-ORB Protocol (RMI / IIOP) to access business methods of remote EJBs.

Sun Java System Application Server supports remote reference from the following client applications. Remote references essentially is an InterOperable Reference (IOR), for an EJB that is used by the clients to invoke a remote operation.

- Java applications that are executing in the ACC accessing EJBs deployed on an application server instance.
- Java applications, not running in the ACC accessing EJBs deployed on an application server instance.
- servlets and JSPs in web applications executing in a different JVM than the target server instance.
- EJBs executing in a different application server instance from the target instance.

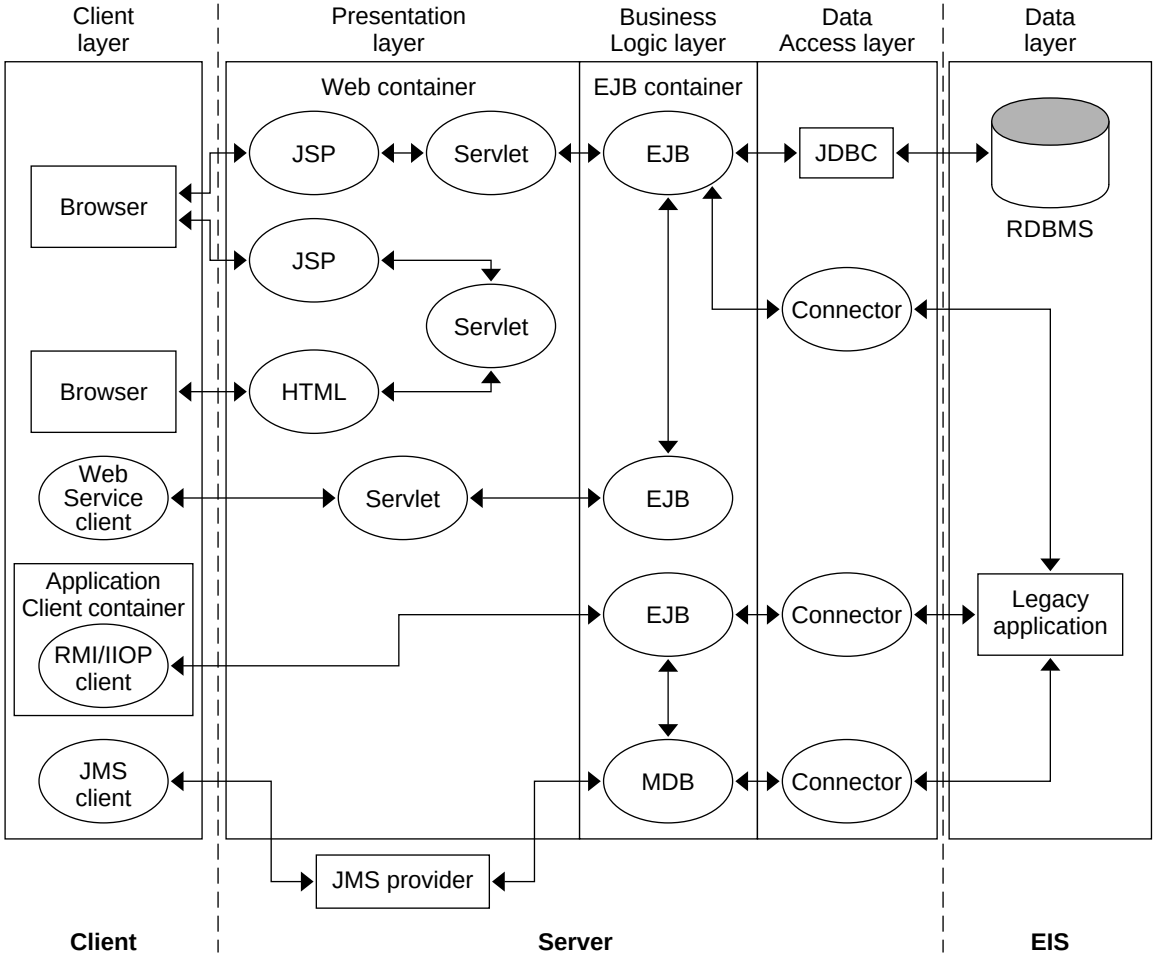
Application Clients

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 Java System 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 Java System 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 Java System 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 Java System Application Server Architecture



Chapter 2

Using the Application Client Container

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

This chapter contains the following sections:

- Introducing the Application Client Container
- Developing Application Clients

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 34.

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 Application Clients

This section describes the procedure to develop, assemble, and deploy client applications, how to use the ACC to package such applications and deploy them to the server. This section describes the following topics:

- Creating an Application Client
- Creating an ACC Client With Load Balancing and Failover Support (Enterprise Edition)
- 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");
   ```

   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 `RMCConverter`.

   Object objref = myEnv.lookup("ejb/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);
```

Creating an ACC Client With Load Balancing and Failover Support (Enterprise Edition)

Sun Java System Application Server, Enterprise Edition provides a highly available J2EE application through the use of load balancing and a sophisticated failover mechanism on the RMI/IIOP path.

The following features are supported:

- Load balancing of requests from client applications on the RMI/IIOP path
- High availability of remote references for RMI/IIOP invocations from stand-alone clients and ACC clients.

High availability of J2EE application means that, if between method invocations, the server instance to the EJB object becomes unavailable, then subsequent invocations are redirected to an alternate available server instance in the cluster.

For more information on High Availability, see the *Sun Java System Application Server Administration Guide.*

Introducing the Properties that Support LB/FO for ACC Clients

In order to enable load balancing capabilities in your ACC client, Sun Java System Application Server supports the following two properties:

- `com.sun.appserv.iiop.endpoints`

  This property defines the list of one or more IIOP endpoints. An endpoint is specified as *host:port* where host is the name or IP address of the system where Sun Java System Application Server is running. Port is the IIOP port at which the server is listening for IIOP requests.

- `com.sun.appserv.iiop.loadbalancingpolicy`

  If the endpoint property is specified, then, this property is used to specify the load balancing policy. The value for this property must be InitialContext-based. The value used to define this property is `ic-based`.

Configuration Changes

Define the load balancing properties in the `sun-acc.xml` file to provide a highly available ACC client. The properties are defined as property elements in the `sun-acc.xml` file.

For example:

```
<client-container>

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

    <property name="com.sun.appserv.iiop.loadbalancingpolicy"
value="ic-based" />

      <property name="com.sun.appserv.iiop.endpoints"
value="qasol-e1:3700", “qasol-el:3800” />

</client-container>
```

To failover an ACC client on the RMI/IIOP path, information about all the endpoints in a cluster to which the RMI/ IIOP requests can be failed over must be available. You must have defined the IIOP endpoints in the `server.xml` file. The `iiop-cluster` element under the `availability-service` element defines the IIOP endpoints.

NOTE The endpoints are categorized as those configured for non-SSL and those configured for SSL. Only endpoints configured for non-SSL are supported. For more information on defining IIOP endpoints, see the *Sun Java System Application Server Administration Guide.*

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 Java System Application Client Deployment Descriptor" on page 52.

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 Java System Application Server Administration Guide.*

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

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

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

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

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

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

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

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

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

Making a Remote Call on the EJB

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

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

This information can be obtained from the server.xml file on the remote system. For more information on server.xml file, see the *Sun Java System 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 Java System 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 Java System Application Server. The J2EE 1.3 specification requires that a stand-alone client operate within the ACC context. However, Sun Java System 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 Java System 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 Java System Application Server Developer's Guide to J2EE Services and APIs.*

5. Next, configure JMS resources on Sun Java System 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 Java System Application Server Administration Guide*.

NOTE You do not have to deploy this application on an ACC or Sun Java System 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 Java System 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 miliseconds

               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 (JMSException 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 (JMSException e) {
                out("There was an error closing the receiver");
                e.printStackTrace();
                }
        }

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

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


       if (connection != null) {
              try {
                     out("Closing connection");
                     connection.close();
                     } catch (JMSException 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 Java System Application Server specific J2EE client application

   For information on `sun-application-client.xml` file, see "Sun Java System Application Client Deployment Descriptor" on page 52.

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 Application Client 1.3//EN'

'http://www.sun.com/software/appserver/dtds/sun-application-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 44.

   `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 47.

Authenticating an Application Client Using the JAAS Module

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

1. Write the `LoginModule` interface.

```
public class ClientPasswordLoginModule implements LoginModule{
private static Logger _logger=null;
    static{
       _logger=LogDomains.getLogger(LogDomains.SECURITY_LOGGER);
        }
}
```

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

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

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

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

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

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

}
```

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

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

```
public boolean login() throws LoginException {

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

}
```

The login information is fetched using the `CallBackHandler`.

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

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

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

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

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

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

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

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

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

```
public boolean logout() throws LoginException {

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

6. Edit the `sun-acc.xml` deployment descriptor to configure JAAS authentication for the client. See "auth-realm" on page 63.

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 44.

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.getPrivateCredent
ials().add(pc);

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

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

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

public boolean logout() throws LoginException {

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

```
}
userPrincipal = null;
return true;
}
}
```

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

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`:

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

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

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

The following steps describe the procedure to create a client:

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

```
public static 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,

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

JNDIName matches the *JNDIName* specified in the deployment file.

Packaging an Application Client Using the ACC

After installing Sun Java System 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="qasol-e1" address="qasol-e1" port="3700">

    <log-service file=" " level="WARNING"/>
</client-container>
```

- If you want to enable load balancing and failover capabilities for the ACC client, follow the steps described in the section "Creating an ACC Client With Load Balancing and Failover Support (Enterprise Edition)" on page 20.

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-clien
t-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_d
es_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 Java System Application Server:

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

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

2. Copy the *install_dir*`/lib/appclient/appclient.jar` file to the desired location. The `appclient.jar` file contains the following files:
   - `appclient/bin` - contains the `appclient` script which you use to launch the ACC.
   - `appclient/lib` - contains the JAR and runtime shared library files.
   - `appclient/lib/appclient` - contains the following files:

- `sun-acc.xml` - the ACC configuration file.
- `client.policy` file- the security manager policy file for the ACC.
- `appclientlogin.conf` file - the login configuration file.
- `client.jar` file - is created during the deployment of the client application.

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

client.policy

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

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

For more information about the permissions you can set in the `client.policy` file, visit the following URL:
`http://java.sun.com/j2se/1.4/docs/guide/security/permissions.html`

Running an Application Client Using the ACC

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

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

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

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

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

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

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

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

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

Sample Client Application

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

install_dir/`samples/rmi-iiop/simple`

Chapter 3

Application Client Deployment Descriptors

This chapter describes the application client deployment descriptors. This chapter contains the following topics:

- Introducing Application Client Deployment Descriptors
- J2EE Application Client Deployment Descriptor
- Sun Java System Application Client Deployment Descriptor
- Application Client Container Configuration File

Introducing 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 Java System Application Server application clients require three deployment descriptors files:

- A J2EE standard file (`application.client.xml`), described in the J2EE Specification.
- An optional Sun Java System Application Server specific client deployment descriptor file (`sun-application-client.xml`), described in this section.
- An optional Sun Java System Application Server specific Application Client Container Configuration file (`sun-acc.xml`), described in this section.

Format of Deployment Descriptors

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

NOTE	Do not edit the DTD files. Their contents change only with new versions of Sun Java System 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 3-1 requirement rules for subelement suffixes

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

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

Data

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

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

For example:

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

Attributes

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

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

For example:

```
<!ATTLIST client-container user-name CDATA #REQUIRED
                password CDATA #REQUIRED
                realm CDATA #IMPLIED>
```

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

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

The `#IMPLIED` label means that the attribute is optional, and that Sun Java System 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 Java System 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 Java System 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 3-2 sun-application-client subelements

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

resource-ref

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

Subelements

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

Table 3-3 `resource-ref` subelements

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

res-ref-name

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

Subelements

none

default-resource-principal

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

If this element is used in conjunction with a JMS Connection Factory resource, the `name` and `password` subelements must be valid entries in Sun Java Message Queue's broker user repository. See the "Security Management" chapter in the *Sun Java System 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 3-4 `default-resource-principal` subelements

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

name

Contains data that specifies the name of the principal.

Subelement

none

password

Contains data that specifies the password for the principal.

Subelement

none

ejb-ref

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

Subelements

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

Table 3-5 `ejb-ref` subelements

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

ejb-ref-name

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

Subelement

none

resource-env-ref

Specifies the name of a resource env reference.

Subelements

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

Table 3-6 `resource-env-ref` subelements

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

resource-env-ref-name

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

Subelements

none

jndi-name

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

Subelement

none

Application Client Container Configuration File

The `sun-acc.xml` file tracks changes in Sun Java System 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 Java System 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 3-7 `client-container` subelements

Element	Required	Description
`target-server`	zero or more	Specifies the IIOP listener configuration of the target server.
`auth-realm`	only one	Specifies the optional configuration for JAAS authentication realm.

Table 3-7 `client-container` subelements *(Continued)*

Element	Required	Description
`client-credential`	only one	Specifies the default client credential that will be sent to the server.
`log-service`	only one	Specifies the default log file and the severity level of the message.
`property`	zero or more	Specifies a property which has a name and a value.

Attributes

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

Table 3-8 `client-container` attributes

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

target-server

Defines the IIOP listener configuration of the target server.

Subelements

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

Table 3-9 `target-server` subelements

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

Attributes

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

Table 3-10 `target-server` attributes

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

description

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

Subelement

none

Attributes

none

client-credential

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

Subelements

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

Table 3-11 `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 3-12 `client-credential` attributes

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

log-service

Specifies configuration settings for the log file.

Subelements

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

Table 3-13 `log-service` subelement

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

Attributes

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

Table 3-14 `log-service` attributes

Attribute	Default Value	Description
`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 *Appclient_Root/logs/client.log.* If the value for the file attribute is set to Null (" "), the log messages are displayed on the console. The log level is set to the highest level (INFO). Log level can not be set when the output mode is console.
`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 3-15 `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 3-16 `ssl attributes`

Attribute	Default Value	Description
`cert-nickname`	none	The nickname of the server certificate in the certificate database or the PKCS#11 token.In the certificate, the name format is *tokenname:nickname*. Including the *tokenname*: part of the name in this attribute is optional.
`ssl2-enabled`	none	(Optional) Determines whether SSL2 is enabled.
`ssl3-enabled`	none	(Optional) Determines whether SSL3 is enabled.
`ssl2-ciphers`	none	(Optional) A space-separated list of the SSL2 ciphers used with the prefix + to enable or - to disable. For example, `+rc4`. Allowed values are `rc4, rc4export, rc2, rc2export, idea, des, desede3`.
`ssl3-tls-ciphers`	none	(Optional) A space-separated list of the SSL3 ciphers used, with the prefix + to enable or - to disable, for example `+rsa_des_sha`. Allowed SSL3 values are `rsa_rc4_128_md5, , rsa_des_sha, rsa_rc4_40_md5, rsa_rc2_40_md5, rsa_null_md5. Allowed TLS values are rsa_des_56_sha, rsa_rc4_56_sha`.
`tls-enabled`	none	Determines whether TLS is enabled.
`tls-rollback-enabled`	none	Determines whether TLS rollback is enabled.TLS rollback should be enabled for MicroSoft Internet Explorer 5.0 and 5.5.
`client-auth-enabled`	none	Determines whether SSL3 client authentication is performed on every request, independent of ACL-based access control.

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

cert-db

Location and password to read the certificate database. Sun Java System 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 3-17 `cert-db` attributes

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

auth-realm

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

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

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

Here is an example of the default file realm:

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

```
classname="com.iplanet.ias.security.auth.realm.file.FileRealm">
<property name="file" value="instance_dir/config/keyfile"/>
<property name="jaas-context" value="fileRealm"/>
</auth-realm>
```

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

Subelements

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

Table 3-18 `auth-realm` subelement

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

Attributes

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

Table 3-19 `auth-realm` attributes

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

property

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

Subelement

none

Attributes

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

Table 3-20 `property` attributes

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

Chapter 4

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 non-ACC Java-based CORBA Clients

CORBA Client Scenarios

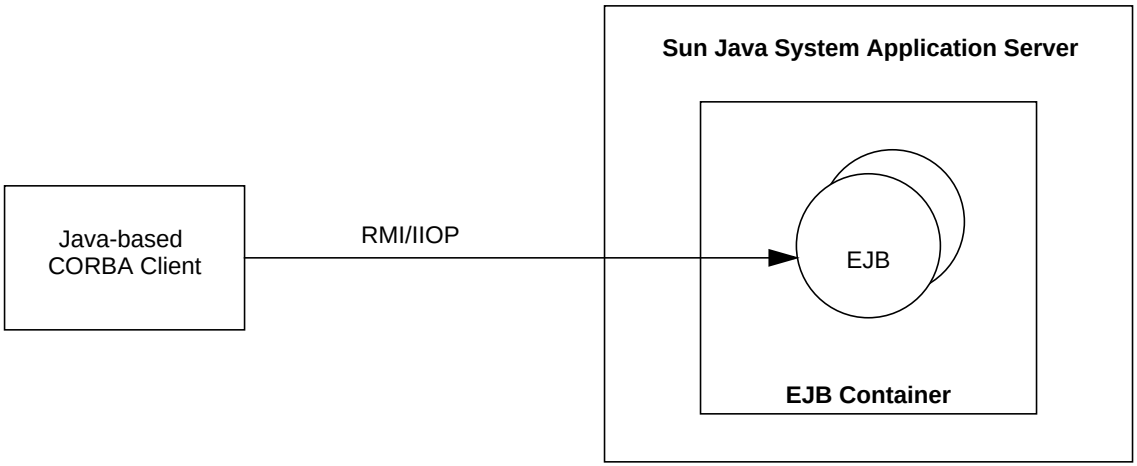
The most common scenarios in which CORBA clients are used are when either a stand-alone program or another application server acts as a client to EJBs deployed to Application Server. This section describes the following scenarios:

- 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.

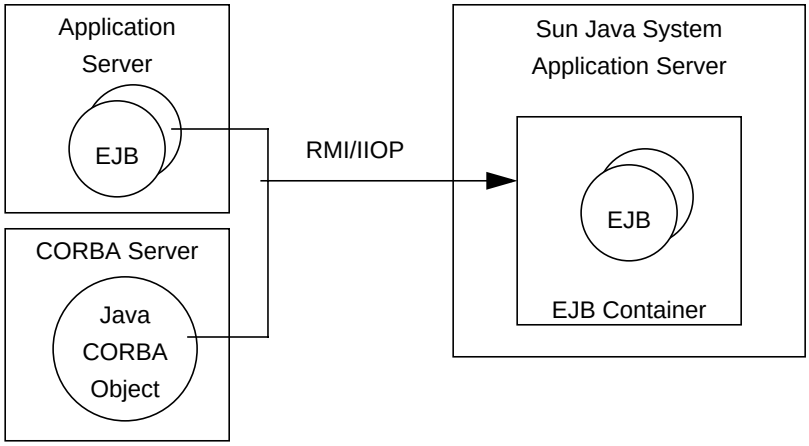
Figure 4-1 Stand-alone Client Accessing the EJB Components



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.

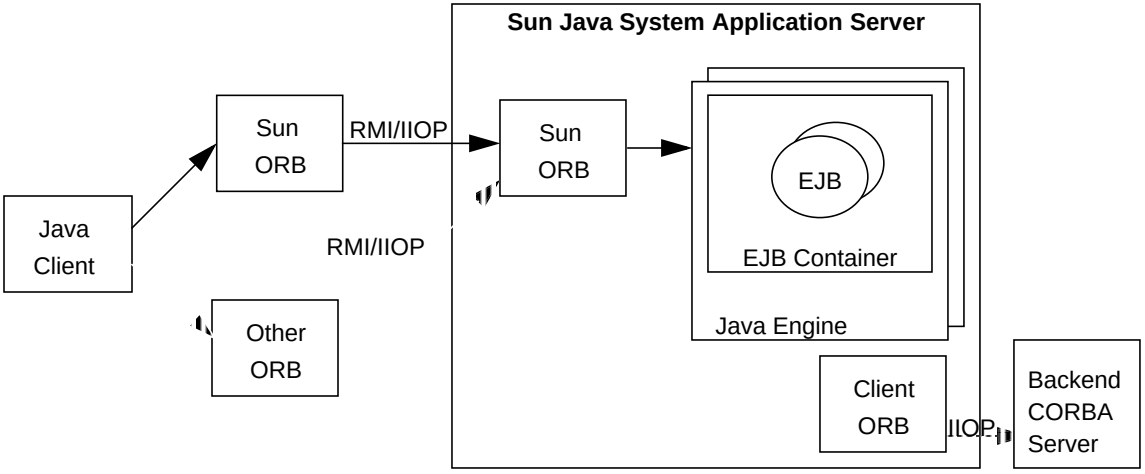
Figure 4-2 Application Server and CORBA Objects Accessing EJB Components



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 4-3 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 non-ACC 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 remote home interface to get a reference to EJB components remote interface.

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

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

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

Specifying the Naming Factory Class

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

For example:

```
...
Properties env = new Properties();
env.put("java.naming.factory.initial","com.sun.jndi.cosnaming.CNCtxFact
ory");
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 JNDI name of the bean in the JNDI lookup. For example, the lookup in the `rmiconverter` sample could be written as follows:

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

The" jndiname" can be found in the `sun-ejb-jar.xml` file under the `<ejb>` element identified by the element `<jndi-name>`.

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

Implementing Load Balancing and Failover Capabilities in the Client Application (Enterprise Edition)

Sun Java Systems Application Server, Enterprise Edition supports the load balancing and failover of IIOP requests from stand-alone and ACC clients, thus providing high availability of J2EE application on the RMI/IIOP path.

In order to enable failover of IIOP requests, IIOP endpoints that constitute the cluster in Sun Java System Application Server must be defined either using the Administration Console or using the command line interface. The IIOP endpoints definitions will be stored in the `server.xml` file. For more information, see the *Sun Java System Application Server Administration Guide* or refer to Admin Console online help.

The failover of J2EE applications happen only for those requests that cannot reach the server and cause a CORBA COMM_FAILURE exception with a return status of COMPLETED_NO on the client. When the server becomes inaccessible, the client side application server ORB will failover the request to another accessible iiop endpoint of the iiop cluster. When failing over the request, ORB randomly selects alternate accessible iiop endpoint of the cluster.

See the *Sun Java System Application Server Administration Guide* for more information on configuring an iiop cluster.

The properties to be set in order to enable load balancing and failover features in your stand-alone clients are:

- `java.naming.factory.initial`

This property is used in specifying the Context Factory that should be used for load balancing the IIOP requests, Set the property to `com.sun.appserv.naming.S1ASCtxFactory`.

- com.sun.appserv.iiop.endpoints

This property specifies the list of IIOP endpoints defined in the server.xml. An IIOP endpoint is specified as *host*:*port* where *host* is the host name or the IP address of the system where Sun Java System Application Server is running and *port* is the IIOP port number at which the server is listening for the IIOP requests.

- com.sun.appserv.iiop.loadbalancingpolicy

If the endpoint property is specified, the, this property is used to specify the load balancing policy. The value for this property must be InitialContext-based load balancing policy. The value used to define this property is ic-based.

In order to implement load balancing capabilities in your client code perform the following steps:

1. Set the following JVM property to configure the ORB.

   ```
   com.sun.CORBA.connection.ORBSocketFactoryClass=com.sun.appserv.enterprise.iiop.EEIIOPSocketFactory

   org.omg.PortableInterceptor.ORBInitializerClass=com.sun.appserv.ee.iiop.EEORBInitializer
   ```

2. Set the classpath to appserv-rt.jar and appserv-rt-ee.jar. These jar files are located in the *install_dir*/lib directory.

3. Use the following property of S1ASCtxFactory class, prior to the instantiation of the InitialContext:

   ```
   Properties env = new Properties();

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

   env.put("com.sun.appserv.iiop.endpoints","trident:3600,
   exodus:3700");

   env.put("com.sun.iiop.loadbalancingpolicy", "ic-based");

   //create an initial naming context
   Context initial = new InitialContext(env);
   ```

This client code instantiates the JNDI InitialContext Object by calling the new InitialContext(env), where env is the list of JNDI SPI properties.

You can also set the stand-alone client load balancing properties as JVM start-up arguments. The properties are set using the following command syntax:

-D<*Propertyname*>=<*Propertyvalue*>

The Java command will look something like the following command:

```
java -Dpropname1=value1 -Dpropname2=value2 <other vm options> classname
program-arguments
```

Sun Java System ORB Configuration

Sun Java System Application Server continues to supports the following load balancing implementation used in Sun ONE Application Server 7.

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

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

```
Properties env = new Properties();

env.setProperty("java.naming.factory.initial","com.sun.appserv.naming.S
1ASCtxFactory");

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

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

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

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

```
Properties env = new Properties();

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

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

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

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

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

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*.

Chapter 5

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's 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 Java System Application Server.

A C++ client can access EJB components via IIOP. However, this can not be achieved using the Sun's ORB due to the absence of a Sun ORB for C++ clients. A C++ client requires an ORB implementation on its side; the Sun 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 Java System Application Server
- Creating a C++ Client

Configuring C++ Clients to Access Sun Java System Application Server

This section describes how to configure C++ clients to access Sun Java System 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 Java System 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 78.

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 Java System Application Server is installed, copy the following classes to your client system:
  - The `appserv-ext.jar` part of Sun Java System 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 Java System 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 Java System Application Server Administration 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 Java System 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/BookCartAppEjb_jar`
         `:`*install_dir*`/lib/appserv-ext.jar`
         `-idl samples.rmi_iiop.cpp.ejb.CartHome`

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

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

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

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

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

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

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

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

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

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

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

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

4. Develop the library for the valuetype implementations.

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

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

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

      ```
      idl --impl-all -I. -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 form the .idl files, perform the following steps:

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

   a. paths to all the application IDLs

   b. paths to all the JDK related IDLs

   c. *ORBacus_home*`/idl`

   d. *ORBacus_home*`/idl/OB`

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

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

   For example,

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

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

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

4. Modify the generated classes.

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

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

The following examples illustrate the scenarios:

Example 1

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

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

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

Example 2

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

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

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

5. Implement the valuetypes.

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

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

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

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

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

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

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/Name
   Service
   ```

   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 Java System Application Server. These samples have been augmented with detailed setup instructions for deploying the application to Sun Java System Application Server. The setup documentation and source code are available at the following location:

install_dir`/samples/rmi-iiop/`

Index

A

ACC
- features 18
- naming 18
- security 18

acc 17

acc clients
- failover
  - properties 21
- load balancing
  - properties 21

acc flag 43

acc package
- asenv configuration settings 44
- editing sun-acc.xml 45
- modifying appclient script 44
- using package-appclient script 46

appclient.jar file 46
- contents 46

application client 15
- accessing EJB 22
- appclient script 47
- create bean instance 20
- creating using the ACC 19
- invoke business method 20
- invoking an EJB module 22
- locate EJB home interface 19
- making a remote call 23
- running 47
- using SSL with CA 45

application client container 17

application client container package
- client.policy file 47

application clients
- authenticating using JAAS 34
- security 34

application-client.xml 52

ATTLIST tag 51

attributes
- #IMPLIED label 51
- #REQUIRED label 51

authentication realm 63

C

c++ clients 77
- configuring 78
- developing 80
- preparing for development 78
- required classes 79
- running 87

client 11, 57
- architecture 15
- web services clients 13

client types 13

clients
- application clients 15
- CORBA clients 14
- JMS clients 14
- RMI-IIOP clients 14
- web client 13
- web services clients 13

client-side load balancing 74
configuring Sun Java System ORB 74
CORBA clients 14
  scenarios 67
cpp files 84
create bean instance
  create method 20
Creating 20

D

deployment descriptors 49
  application client 52
  application client container 57
  attributes 51
  data 51
  element 50
  format 50
  J2EE application client 52
  subelements 50
developing c++ clients
  generate cpp files 84
  generate IDL files 80, 81
  generate valuetypes 82
  implementing valuetypes 86
  modifying the generated IDL files 83
  registering valuefactories 86

E

EJBs
  accessing with IIOP 67
  specifying JNDI name 71

F

form-hint-field attribute 58

I

IDL files
  generate 81
    rmic tool 81
IIOP 14
  accessing EJBs 67
  accessing servers 68
IIOP listener configuration 58
IIOP / SSL configuration 61
InitialContext 71
invoking a J2EE client without using acc 42

J

J2EE application client 19
J2EE platform layers 11
  Business logic layer 12
  client 11
  database 12
  presentation 11
J2SE policy file 47
JAAS module 34
  LoginModule 34, 35
JMS clients 14
JNDI 14
  specifying EJB name 71

L

launching acc 47
library for valuetype implementation
  developing 82
load balancing 74
logging messages 45
LoginModule
  CallBackHandler 35
  commit() method 36
  integrate 37
  login() method 35
  logout() method 36

M

message-driven beans 14
modifying the generated IDL files
  changing valuetypes 83
  deleting duplicate variables 83

N

naming factory class 71

O

ORB architecture 69

P

param-name element 54
presentation layer
  J2EE components 11
  non-J2EE components 12

R

RMI/IIOP 14
RMI/IIOP client
  load balancing and failover 72
rpm 10

S

S1ASCtxFactory class 74
scenarios
  server-server 68
  stand-alone 67
security
  authentication data 18
  JAAS module 18
  using SSL with CA 45
setting the ORB port 45
showrev 10
SSL 18
SSL processing parameters 62
stand-alone clients
  load balancing 72
stand-alone CORBA client
  creating 70
  running 75
subelements
  requirement rules 50
Sun customer support 10
Sun's ORB 77
sun-acc.xml elements
  auth-realm 63
  cert-db 63
  client-container 57
  client-credential 59
  description 59
  log-service 60
  property 64
  security 61
  ssl 62
  target-server 58
sun-acc.xml file 57
  elements in 57
sun-application element
  definition in sun-application_1_3-0.dtd file 50
sun-application-client.xml 52
sun-application-client.xml elements
  default-resource-principal 54
  ejb-ref 55
  ejb-ref-name 55
  jndi-name 56
  name 55
  password 55
  resource-env-ref 56
  resource-env-ref-name 56
  resource-ref 53
  resource-ref-name 54
  sun-application-client 53

sun-application-client.xml file 53
elements in 52

T

thin client 13

W

web client 13
web services clients 13