Oracle® Universal Content Management

Remote Intradoc Client (RIDC) Developer Guide

10*g* Release 3 (10.1.3.3.3)

November 2008

ORACLE®

Remote Intradoc Client (RIDC) Developer Guide, 10*g* Release 3 (10.1.3.3.3)

Copyright © 1996, 2008, Oracle. All rights reserved.

Primary Author: Will Harris

Contributor: Adam Stuenkel

The Programs (which include both the software and documentation) contain proprietary information; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

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

If the Programs are delivered to the United States Government or anyone licensing or using the Programs on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software--Restricted Rights (June 1987). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

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

Oracle, JD Edwards, PeopleSoft, and Siebel are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

Contents

Preface v

Audience v
Related Documents v
Conventions v

1 Using Remote Intradoc Client (RIDC)

1.1 Initialization 1-1
1.2 Client Configuration 1-2
1.3 SSL Configuration 1-2
1.4 Authentication 1-3
1.5 Usage 1-3
1.6 Connection Handling 1-4
1.7 Streams 1-5
1.8 JSP/JSPX 1-5
1.9 Reusable Binders 1-6

A Secure Socket Layer (SSL) Communication

A.1 Installing the Security Providers Component A-1
A.2 Configuring the Content Server for SSL A-2
A.2.1 Setting Up a New Incoming Provider A-2
A.2.2 Specifying Truststore and Keystore Information A-4
A.3 Certificate Signing Options A-5
A.3.1 Creating the Client and Server Keys A-5
A.3.2 Self-Signing the Certificates A-6
A.3.3 Exporting the Certificates A-7
A.3.4 Importing the Certificates A-7

Index

Preface

The Remote Intradoc Client (RIDC) Administration Guide contains information to assist administrators responsible for configuring RIDC to communicate with Oracle Content Server.

Audience

This guide is intended for application developers and integrators.

Related Documents

For more information, see the following documents:

- Oracle Content Integration Suite (CIS) Administration Guide
- Oracle Content Integration Suite (CIS) Developer Guide
- Oracle Content Integration Suite (CIS) Release Notes
- Oracle Content Portlet Suite (CPS) Installation Guide
- Oracle Content Portlet Suite (CPS) Developer Guide
- Oracle Content Portlet Suite (CPS) Release Notes

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.
/	This guide uses the forward slash (/) to separate directories. Depending on your operating system, you may need to change the separation markers when defining directories.

1 Using Remote Intradoc Client (RIDC)

The Remote Intradoc Client (RIDC) provides a thin communication API for communication with Oracle Content Server. This API removes data abstractions to the content server while still providing a wrapper to handle connection pooling, security, and protocol specifics.

This chapter contains the following sections:

- "Initialization" on page 1-1
- "Client Configuration" on page 1-2
- "SSL Configuration" on page 1-2
- "Authentication" on page 1-3
- "Usage" on page 1-3
- "Connection Handling" on page 1-4
- "Streams" on page 1-5
- "JSP/JSPX" on page 1-5
- "Reusable Binders" on page 1-6

1.1 Initialization

RIDC supports Intradoc socket-based communication and the HTTP protocol.

- Intradoc communication is handled via the content server Intradoc port (typically 4444). This communication method requires a trusted connection between the client and content server. Intradoc communication can also be configured to run over SSL. See "SSL Configuration" on page 1-2 for more information.
- This implementation of the HTTP protocol uses Apache's Jakarta HttpClient. Communication with the content server using the HTTP protocol requires authentication credentials for each request.

  Refer to Apache's Jakarta HttpClient documentation for more information: `http://hc.apache.org/httpclient-3.x`

This table shows the three URL formats that are supported:

URL	Description
http://locahost/contentserver/idcplg	HTTP communication. URL to the content server CGI path.

URL	Description
idc://localhost:4444	Intradoc communication. Uses the Intradoc port, only requires hostname and port number.
idcs://localhost:4433	SSL communication. Uses SSL over the Intradoc port; requires extra configuration to load the SSL certificates

This code initializes an Intradoc connection:

```
// create the manager
IdcClientManager manager = new IdcClientManager ();

// build a client that will communicate using the intradoc protocol
IdcClient idcClient = manager.createClient
    ("idc://localhost:4444");
```

This code initializes an HTTP connection:

```
// create the manager
IdcClientManager manager = new IdcClientManager ();

// build a client that will communicate using the HTTP protocol
IdcClient idcClient = manager.createClient
    ("http://localhost/contentserver/idcplg");
```

1.2 Client Configuration

Configuration of the clients can be done after they are created. Configuration parameters include setting the socket timeouts, connection pool size, etc. The configuration is specific to the protocol; if you cast the *IdcClient* object to the specific type, you can then retrieve the protocol configuration object for that type. For example, the following code sets the socket timeout and wait time for Intradoc connections:

```
// build a client as cast to specific type
IntradocClient idcClient = (IntradocClient)manager.createClient
    ("idc://localhost:4444");

// get the config object and set properties
idcClient.getConfig ().setSocketTimeout (30000); // 30 seconds
idcClient.getConfig ().setConnectionSize (20);   // 20 connections
```

1.3 SSL Configuration

RIDC allows Secure Socket Layer (SSL) communication with Oracle Content Server using the Intradoc communication protocol.

> **Note:** You must install and enable the Security Providers component on the instance of the content server you wish to access and configure the content server for SSL communication. See Appendix A, "Secure Socket Layer (SSL) Communication" for more information.

An example of using the IDC protocol over SSL:

```
// build a secure IDC client as cast to specific type
IntradocClient idcClient = (IntradocClient)manager.createClient
```

```
    ("idcs://localhost:4433");                // idcs connection

// set the SSL socket options
config.setKeystoreFile ("ketstore/client_keystore"); // keystore file
config.setKeystorePassword ("password");      // keystore password
config.setKeystoreAlias ("SecureClient");     // keystore aliaa
config.setKeystoreAliasPassword ("password"); // keystore alias password
```

1.4 Authentication

All calls to RIDC require some user identity. Optionally, this identity can be accompanied by credentials as required by the protocol used. The user identity is represented by the *IdcContext* object; once created, it can be reused for all subsequent calls. To create an identity, you pass in the username and optionally some credentials:

```
// create a simple identity with no password
IdcContext userContext = new IdcContext ("sysadmin");

// create an identity with a password
IdcContext userPasswordContext = new IdcContext ("sysadmin", "idc");
```

For Intradoc URLs, you do not need any credentials as the request is trusted between the content server and the client. For HTTP URLs, the context requires credentials. The credentials can be a simple password or anything that the HttpClient package supports.

1.5 Usage

To invoke a service, use the *ServiceRequest* object, which can be obtained from the client. Creating a new request will also create a new binder and set the service name in the binder along with any other default parameters. You can then populate the binder as needed for the request. This code executes a service request and gets back a data binder of the results:

```
// get the binder
DataBinder binder = idcClient.createBinder ();

// populate the binder with the parameters
binder.putLocal ("IdcService", "GET_SEARCH_RESULTS");
binder.putLocal ("QueryText", "");
binder.putLocal ("ResultCount", "20");

// execute the request
ServiceResponse response = idcClient.sendRequest (userContext, binder);

// get the binder
DataBinder serverBinder = response.getResponseAsBinder ();
```

The ServiceResponse contains the response from the content server. From the response, you can access the stream from the content server directly or you can parse it into a databinder and query the results. This code takes the above ServiceResponse and get the search results, printing out the title and author values:

```
// get the binder
DataBinder binder = response.getResponseAsBinder ();
DataResultSet resultSet = binder.getResultSet ("SearchResults");

// loop over the results
for (DataObject dataObject : resultSet.getRows ()) {
```

```
        System.out.println ("Title is: " + dataObject.get ("dDocTitle"));
        System.out.println ("Author is: " + dataObject.get ("dDocAuthor"));
    }
```

1.6 Connection Handling

The RIDC client pools connections, this requires that the caller of the code close resources when done with a response. This is done automatically when calling *getResponseAsBinder* or by calling the *close* method on the stream returned via a call to *getResponseStream*. If a user does not want to examine the results, the *close* method must still be called, either by getting the stream and closing it directly or by calling *close* on the *ServiceResponse* object.

Closing via the response as binder:

```
// execute the request
ServiceResponse response = idcClient.sendRequest (userContext, binder);

// get a binder closes the response automatically
response.getResponseAsBinder ();
```

Closing via the stream:

```
// execute the request
ServiceResponse response = idcClient.sendRequest (userContext, binder);

// get the result stream and read it
InputStream stream = response.getResponseStream ();
int read = 0;
while ((read = stream.read ()) != -1) {
}
//close the stream
stream.close ();
```

Closing via the *close* method on the *ServiceResponse* object:

```
// execute the request
ServiceResponse response = idcClient.sendRequest (userContext, binder);

// close the response (which closes the stream directly)
response.close ();
```

Pooling Options

How RIDC handles the pooling of connections is configurable via the property defined in IdcClientConfig.getConnectionPool. The *getConnectionPool* method has two options: *pool* or *simple*.

- **pool**: This is the default setting. An internal pool is used which allows a configurable number of active connections at a time. By default, RIDC allows 20 active connections to the content server. The number of active connections is configurable via the property in IdcClientConfig.getConnectionSize. See "Client Configuration" on page 1-2 for more information.
- **simple**: This option does not enforce a connection maximum. Every connection is allowed to proceed without blocking.

A different pool implementation can be registered via the *registerPool()* method of IdcClientManager.getConnectionPoolManager(). This maps a name to an implementation of the *ConnectionPool* interface. The name can then be used in the *IdcClientConfig* object to select that pool for a particular client.

1.7 Streams

Streams are sent to the content server via the *TransferStream* interface. This interface wraps the actual stream with metadata about the stream (length, name, etc.). This code will perform a checkin to the content server:

```
// create request
DataBinder binder = idcClient.createBinder();
binder.putLocal ("IdcService", "CHECKIN_UNIVERSAL");

// get the binder
binder.putLocal ("dDocTitle", "Test File");
binder.putLocal ("dDocName", "test-checkin-6");
binder.putLocal ("dDocType", "ADACCT");
binder.putLocal ("dSecurityGroup", "Public");

// add a file
binder.addFile ("primaryFile", new File ("test.doc"));

// checkin the file
idcClient.sendRequest (userContext, binder);
```

A stream can be received from the content server via the *ServiceResponse* object; the response is not converted into a data binder unless specifically requested. If you only want the raw HDA data, you can retrieve that directly. You can also convert the response to a string or data binder.

```
// create request
DataBinder binder = idcClient.createBinder ();

// execute the service
ServiceResponse response = idcClient.sendRequest (userContext, binder);

// get the response stream
InputStream stream = response.getResponseStream ();

// get the response as a string
String responseString = response.getResponseAsString ();

// parse into data binder
DataBinder dataBinder = response.getResponseAsBinder ();
```

1.8 JSP/JSPX

The RIDC objects follow the standard Java Collection paradigms which makes them extremely easy to consume from a JSP/JSPX page. Assume the above*ServerResponse* object is available in a HttpServletRequest in an attribute called *idcResponse*; this JSPX code will iterate over the response and create a small table of data:

```
<table>
  <tr>
    <th>Title</th>
    <th>Author</th>
    <th>Release Date</th>
  </tr>
<c:forEach var="row" items="${idcResponse.dataBinder.SearchResults.rows}">
  <tr>
    <td>${row.dDocTitle}</td>
    <td>${row.dDocAuthor}</td>
    <td>${row.dInDate}</td>
```

```
   </tr>
</c:forEach>
</table>
```

1.9 Reusable Binders

Binders can be reused among multiple requests. A binder from one request can be sent in to another request. For example, this code pages the search results by reusing the same binder for multiple calls to the content server:

```
// create the user context
IdcContext idcContext = new IdcContext ("sysadmin", "idc");

// build the search request binder
DataBinder binder = idcClient.createBinder();
binder.putLocal("IdcService", "GET_SEARCH_RESULTS");
binder.putLocal("QueryText", "");
binder.putLocal("ResultCount", "20");

// send the initial request
ServiceResponse response = idcClient.sendRequest (binder, idcContext);
DataBinder responseBinder = response.getResponseAsBinder();

// get the next page
binder.putLocal("StartRow", "21");
response = idcConnection.executeRequest(idcContext, binder);
responseBinder = response.getResponseAsBinder();

// get the next page
binder.putLocal("StartRow", "41");
response = idcConnection.executeRequest(binder, idcContext);
responseBinder = response.getResponseAsBinder();
```

A

Secure Socket Layer (SSL) Communication

RIDC allows Secure Socket Layer (SSL) communication with Oracle Content Server. This section provides basic information on SSL communication including how to set up a sample implementation for testing purposes. This sample implementation, uses a JDK utility to create self-signed keypair and certificates. Oracle does not provide signed certificates. For most implementations, you will want a certificate signed by a universally recognized Certificate Authority.

This chapter contains the following sections:

- "Installing the Security Providers Component" on page A-1
- "Configuring the Content Server for SSL" on page A-2
- "Certificate Signing Options" on page A-5

A.1 Installing the Security Providers Component

You must have a valid KeyStore / TrustManager with signed trusted certificates on both the client and on the content server. See "Certificate Signing Options" on page A-5 for information on generating these items.

You must install and enable the Security Providers component on the instance of the content server you wish to access. The SecurityProviders.zip file is provided with Oracle Content Server 10*g*.

For setup instructions, refer to the readme.txt included with the Security Providers component or follow these steps to install the Security Providers component using the Component Manager:

1. Log onto the content server as an administrator.
2. Click **Administration** and then **Admin Server**.
3. Click the button that corresponds with your server.

   The options and status page of the content server instance is displayed.
4. Click **Component Manager** in the menu on the left.

   The Component Manager page appears.
5. Click **Browse**.

   A file selection dialog box opens.
6. Select the SecurityProviders.zip file and close the dialog box.
7. Click **Install**.

   A page appears, confirming what will be installed.

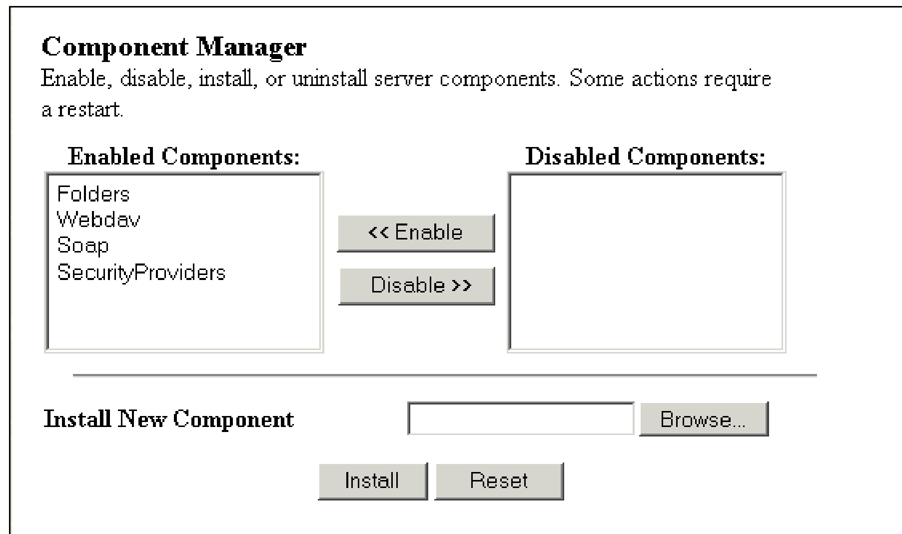
8. Click **Continue**.

   After this process is completed, a message appears, stating that the component was uploaded and installed successfully.

9. Click **Continue**.

10. Enable the component and restart the content server.

Figure A–1 Security Providers Component Enabled in Component Manager



A.2 Configuring the Content Server for SSL

For SSL communication, the content server must be configured with a new incoming provider and the truststore/keystore information must be specified.

This section contains the following topics:

- "Setting Up a New Incoming Provider" on page A-2
- "Specifying Truststore and Keystore Information" on page A-4

A.2.1 Setting Up a New Incoming Provider

You can set up a new keepalive incoming socket provider or a new SSL incoming socket provider. The set up steps for both are listed below. Using keepalive improves the performance of a session and is recommended for most implementations.

1. Login to the content server as an administrator.

2. Click **Administration** and then **Providers**.

Figure A–2 List of Providers in Content Server

Create a New Provider

Provider Type	Description	Action
outgoing	Configuring an outgoing provider.	Add
database	Configuring a database provider.	Add
incoming	Configuring an incoming provider.	Add
preview	Configuring a preview provider.	Add
ldapuser	Configuring an LDAP user provider.	Add
keepaliveincoming	Configure a keepalive incoming socket provider.	Add
keepaliveoutgoing	Configure a keepalive outgoing socket provider.	Add
sslincoming	Configure an SSL incoming socket provider.	Add
ssloutgoing	Configure an SSL incoming socket provider.	Add

3. For sslincoming, click **Add**.

   The Add Incoming Provider page appears.

4. Enter a Provider Name and Provider Description.

   When your new provider is set up, a directory with your provider name is created as a sub-directory of the /providers directory.

5. Enter an open Server Port.

6. Enter configuration information for either a new SSL keepalive incoming socket provider or a new SSL incoming socket provider. The set up steps for both are listed below. Using keepalive improves the performance of a session and is recommended for most implementations.

   SSL keepalive incoming socket provider

   Provider Class = **idc.provider.ssl.SSLSocketIncomingProvider**

   Connection Class = **idc.provider.KeepaliveSocketIncomingConnection**

   Server Thread Class = **idc.server.KeepaliveIdcServerThread**

Figure A–3 Keepalive Provider

Provider Class	idc.provider.ssl.SSLSocketIncomingProvider
Connection Class	idc.provider.KeepaliveSocketIncomingConnection
Configuration Class	
Server Thread Class	idc.server.KeepaliveIdcServerThread

   SSL incoming socket provider

   Provider Class = **idc.provider.ssl.SSLSocketIncomingProvider**

   Connection Class = **intradoc.provider.SocketIncomingConnection**

   Server Thread Class = **intradoc.server.IdcServerThread**

Figure A–4 Incoming Provider

Provider Class	idc.provider.ssl.SSLSocketIncomingProvider
Connection Class	intradoc.provider.SocketIncomingConnection
Configuration Class	
Server Thread Class	intradoc.server.IdcServerThread

7. Click **Add**.

After you have completed setting up a new incoming provider you must also specify truststore and keystore information. See "Specifying Truststore and Keystore Information" on page A-4 for these steps.

A.2.2 Specifying Truststore and Keystore Information

After you have set up a new incoming provider you must specify truststore and keystore information. You must create a file named sslconfig.hda that provides the truststore and keystore information.

1. Locate your SSL provider directory. When you set up a new provider, a directory with your provider name was created as a sub-directory of the /providers directory.
2. Create a text file that defines truststore and keystore information:

Attribute	Description
TruststoreFile	Complete directory path to your truststore file (to validate client certificates). If you followed the steps in this guide, both the certificates and keys are in the same keystore.
TruststorePassword	Password to access the truststore.
KeystoreFile	Complete directory path to the keystore file.
KeystorePassword	Password to access the keystore.
KeystoreAlias	Alias of the keystore.
KeystoreAliasPassword	Password to access the keystore.

Example:

```
@Properties LocalData
  TruststoreFile=/tmp/ssl/server_keystore
  TruststorePassword=idcidc
  KeystoreFile=/tmp/ssl/server_keystore
  KeystorePassword=idcidc
  KeystoreAlias=SecureServer
  KeystoreAliasPassword=idcidc
@end
```

3. Save as sslconfig.hda and place in your SSL provider directory. Refer to the readme.txt included with the Security Providers component for more information.

A.3 Certificate Signing Options

For most implementations, you will want a certificate signed by a universally recognized Certificate Authority. However, if you control both the client and server and only want to ensure that your transmissions are not intercepted, or if you are simply testing your implementation, you can create your own self-signed keypair and certificates using the JDK utility called keytool.

Sun's Key and Certificate Management Tool (keytool) is a key and certificate management utility that enables users to administer their own public/private key pairs and associated certificates for use in self-authentication. It is provided as part of Sun's JDK. keytool is a command line utility. The executable is located in the <jdk-home>/bin subdirectory.

This section contains the following topics:

- "Creating the Client and Server Keys" on page A-5
- "Self-Signing the Certificates" on page A-6
- "Exporting the Certificates" on page A-7
- "Importing the Certificates" on page A-7

A.3.1 Creating the Client and Server Keys

From a command line prompt, navigate to the <jdk-home>/bin subdirectory and issue the -genkey command (this command generates a new key and takes several arguments). These arguments are used with this command:

Argument	Description
-alias	Alias of the key being created (this is the way a keystore knows which element in the file you are referring to when you perform operations on it).
-keyalg	Encryption algorithm to use for the key.
-keystore	Name of the binary output file for the keystore.
-dname	Distinguished name that will identify the key.
-keypass	Password for the key that is being generated.
-storepass	Password used to control access to the keystore.

Generate a separate key pair for both the client and server. To do this you will need to run the -genkey command twice, each time placing it into a separate keystore.

You will need to specify the alias, the algorithm to use, the keystore name, the distinguished name, and passwords for the keys and the keystore. This example uses RSA for the algorithm and *idcidc* for the passwords.

Client Argument Values

Use these argument values for the client:

- -alias SecureClient
- -keyalg RSA
- -keystore client_keystore
- -dname "cn=SecureClient"

- -keypass idcidc
- -storepass idcidc

```
# keytool -genkey -alias SecureClient -keyalg RSA -keystore client_keystore
  -dname "cn=SecureClient" -keypass idcidc -storepass idcidc
```

Server Argument Values

Use these argument values for the server:

- -alias SecureServer
- -keyalg RSA
- -keystore server_keystore
- -dname "cn=SecureServer"
- -keypass idcidc
- -storepass idcidc

```
# keytool -genkey -alias SecureClient -keyalg RSA -keystore client_keystore
  -dname "cn=SecureClient" -keypass idcidc -storepass idcidc

# keytool -genkey -alias SecureServer -keyalg RSA -keystore server_keystore
  -dname "cn=SecureServer" -keypass idcidc -storepass idcidc
```

Each of these commands will generate a key pair wrapped in a self-signed certificate and stored in a single-element certificate chain.

A.3.2 Self-Signing the Certificates

Keys are unusable unless they are signed. The keytool will self-sign them for you so that you can use the certificates for internal testing, however, these keys are not signed for general use.

From a command line prompt, issue the -selfcert command (this command self-signs your certificates and takes several arguments). Run the -selfcert command twice, once for the client and again for the server.

Client Argument Values

Use these argument values for the client:

- -alias SecureClient
- -keystore client_keystore
- -keypass idcidc
- -storepass idcidc

Server Argument Values

Use these argument values for the server:

- -alias SecureServer
- -keystore server_keystore
- -keypass idcidc
- -storepass idcid

```
# keytool -selfcert -alias SecureClient -keystore client_keystore
  -keypass idcidc -storepass idcidc

# keytool -selfcert -alias SecureServer -keystore server_keystore
  -keypass idcidc -storepass idcidc
```

The certificate is now signed by its private and public key, resulting in a single-element certificate chain. This replaces the one that you generated previously.

A.3.3 Exporting the Certificates

After you have created the client and server keys, and self-signed the certificates, you now have two keypairs (public and private keys) in two certificates locked in two keystores. Since each application will need to have the public key of the other in order to encrypt and decrypt data, we need to place a copy of the public keys in the other keystore.

From a command line prompt, issue the -export command (this command exports your certificates and takes several arguments). Run the -export command twice, once for the client and again for the server. Use the -file argument to redirect the output to a file instead of the console.

Client Argument Values

Use these argument values for the client:

- -alias SecureClient
- -file client_cert
- -keystore client_keystore
- -storepass idcidc

Server Argument Values

Use these argument values for the server:

- -alias SecureServer
- -file server_cert
- -keystore server_keystore
- -storepass idcidc

```
# keytool -export -alias SecureClient -file client_cert -keystore client_keystore
  -storepass idcidc
Certificate stored in file client_cert

# keytool -export -alias SecureServer -file server_cert -keystore server_keystore
  -storepass idcidc
Certificate stored in file server_cert
```

We have now exported the certificate (containing the public key and signer information) to a binary certificate file.

A.3.4 Importing the Certificates

The final step in setting up your self-signed certificates is to import the public certificates of each program into the keystore of the other. Keytool will present you

with the details of the certificates you are requesting to be imported and provide a request confirmation.

From a command line prompt, issue the **-import** command (this command imports your certificates and takes several arguments). Run the **-import** command twice, once for the client and again for the server. Notice that the **-keystore** values are reversed.

Client Argument Values

Use these argument values for the client:

- -alias SecureClient
- -file client_cert
- -keystore server_keystore
- -storepass idcidc

Server Argument Values

Use these argument values for the server:

- -alias SecureServer
- -file server_cert
- -keystore client_keystore
- -storepass idcidc

```
# keytool -import -alias SecureClient -file client_cert
  -keystore server_keystore -storepass idcidc

Owner: CN=SecureClient
Issuer: CN=SecureClient
Serial number: 3c42e605
Valid from: Mon Jan 14 08:07:01 CST 2002 until: Sun Apr 14 09:07:01 CDT 2002
Certificate fingerprints:
  MD5:ÿ 17:51:83:84:36:D2:23:A2:8D:91:B7:14:84:93:3C:FF
  SHA1: 61:8F:00:E6:E7:4B:64:53:B4:6B:95:F3:B7:DF:56:D3:4A:09:A8:FF
Trust this certificate? [no]:ÿ y
Certificate was added to keystore

# keytool -import -alias SecureServer -file server_cert -keystore client_keystore
  -storepass idcidc

Owner: CN=SecureServer
Issuer: CN=SecureServer
Serial number: 3c42e61e
Valid from: Mon Jan 14 08:07:26 CST 2002 until: Sun Apr 14 09:07:26 CDT 2002
Certificate fingerprints:
  MD5:ÿ 43:2F:7D:B6:A7:D3:AE:A7:2E:21:7C:C4:52:49:42:B1
  SHA1: ED:B3:BB:62:2E:4F:D3:78:B9:62:3B:52:08:15:8E:B3:5A:31:23:6C
Trust this certificate? [no]:ÿ y
Certificate was added to keystore
```

We have now imported the certificates of each program into the keystore of the other.

Index

A

Apache Jakarta HttpClient, 1-1
Authentication, 1-3

C

Certificate Authority, A-1, A-5
Client Configuration, 1-2
ConnectionPool interface, 1-4

H

HTTP protocol, 1-1
HttpClient, 1-1

I

IDC protocol over SSL, 1-2
IdcClient object, 1-2
IdcClientConfig object, 1-4
IdcContext object, 1-3
Initialization, 1-1
Intradoc communication, 1-1

J

Jakarta HttpClient, 1-1

K

Key and Certificate Management Tool (keytool), A-5
key and certificate management utility, A-5
keytool, A-5

P

Pooling Options, 1-4
public/private key pairs, A-5

R

Reusable Binders, 1-6
RIDC objects, 1-5

S

Secure Socket Layer (SSL) communication, 1-2, A-1
Security Providers component, 1-2, A-1
ServiceRequest object, 1-3
ServiceResponse object, 1-4, 1-5
SSL Configuration, 1-2

T

TransferStream interface, 1-5

U

URL formats, 1-1