webMethods EntireX

EntireX Java Wrapper

Version 9.5 SP1

November 2013

This document applies to webMethods EntireX Version 9.5 SP1.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: EXX-EEXXJAVAWRAPPER-95SP1-20140628

Table of Contents

I

1 Introduction to the Java Wrapper

The EntireX Java Wrapper provides access to EntireX RPC-based components from Java applications. EntireX Java RPC enables users to develop both client and server applications written in Java. Java applets can also be used as EntireX RPC clients.

To use the Java Wrapper, the IDL file must be in a Java project.

The Java Wrapper uses the properties of an IDL file to

- use a source folder (the folder that the Java builder uses to compile Java source classes)
- make the classes public
- extend a custom class for the RPC client
- put the client and the tester in a client package
- put the server in a server package.
- set a superclass to be extended by all the bean-compliant client-generated classes.

The Java Wrapper generates multiple Java sources from an IDL file and (internally) the related CVM file (see *CVM File*), if such a file has been generated. The following sources can be generated:

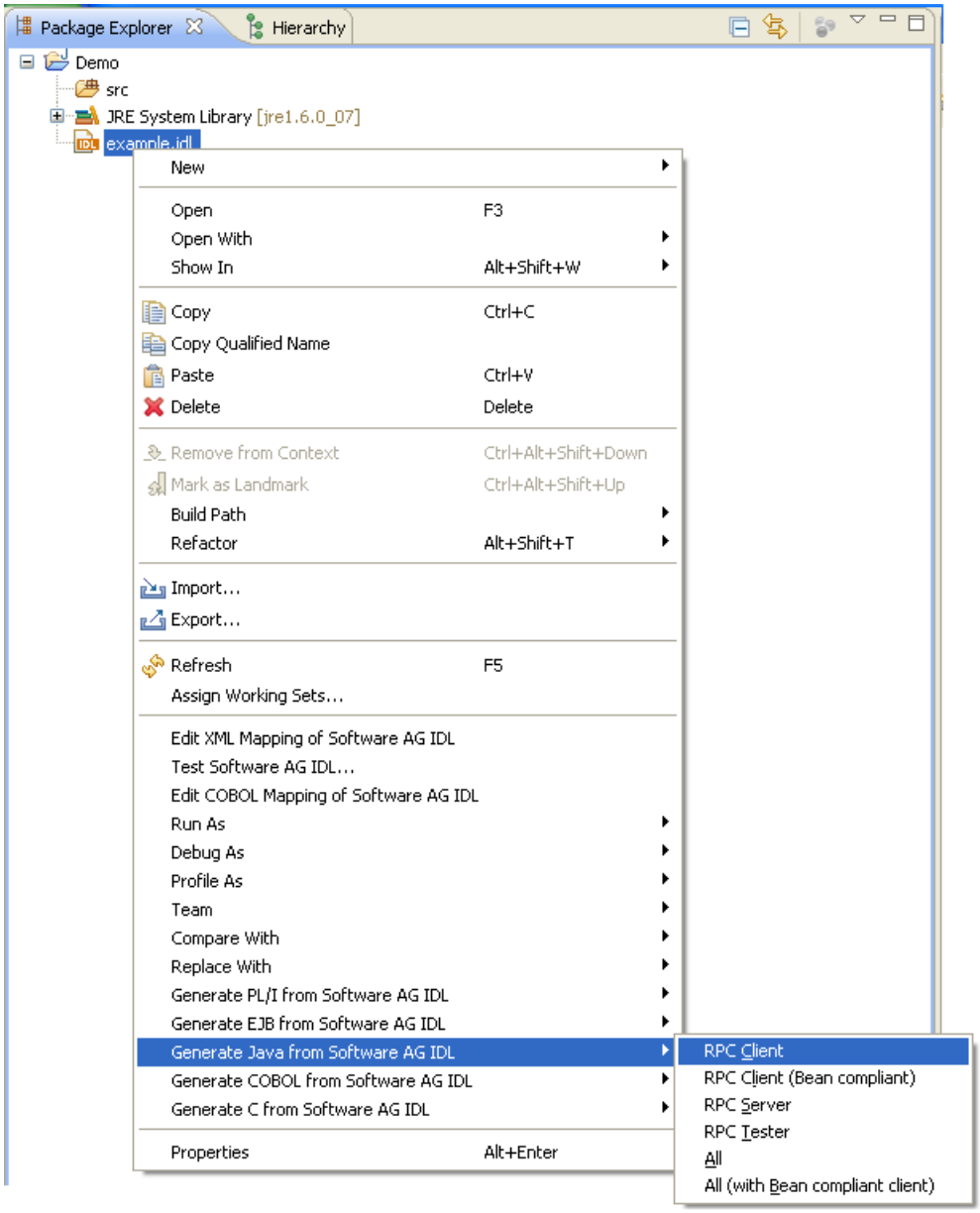
- RPC client
- RPC client (Bean-compliant)
- RPC server
- RPC tester

2 Using the Java Wrapper

Generating Java Sources

Select an IDL File

To generate a Java source, select an IDL file and, using the context menu, choose **All** or **All (with Bean compliant client)**.



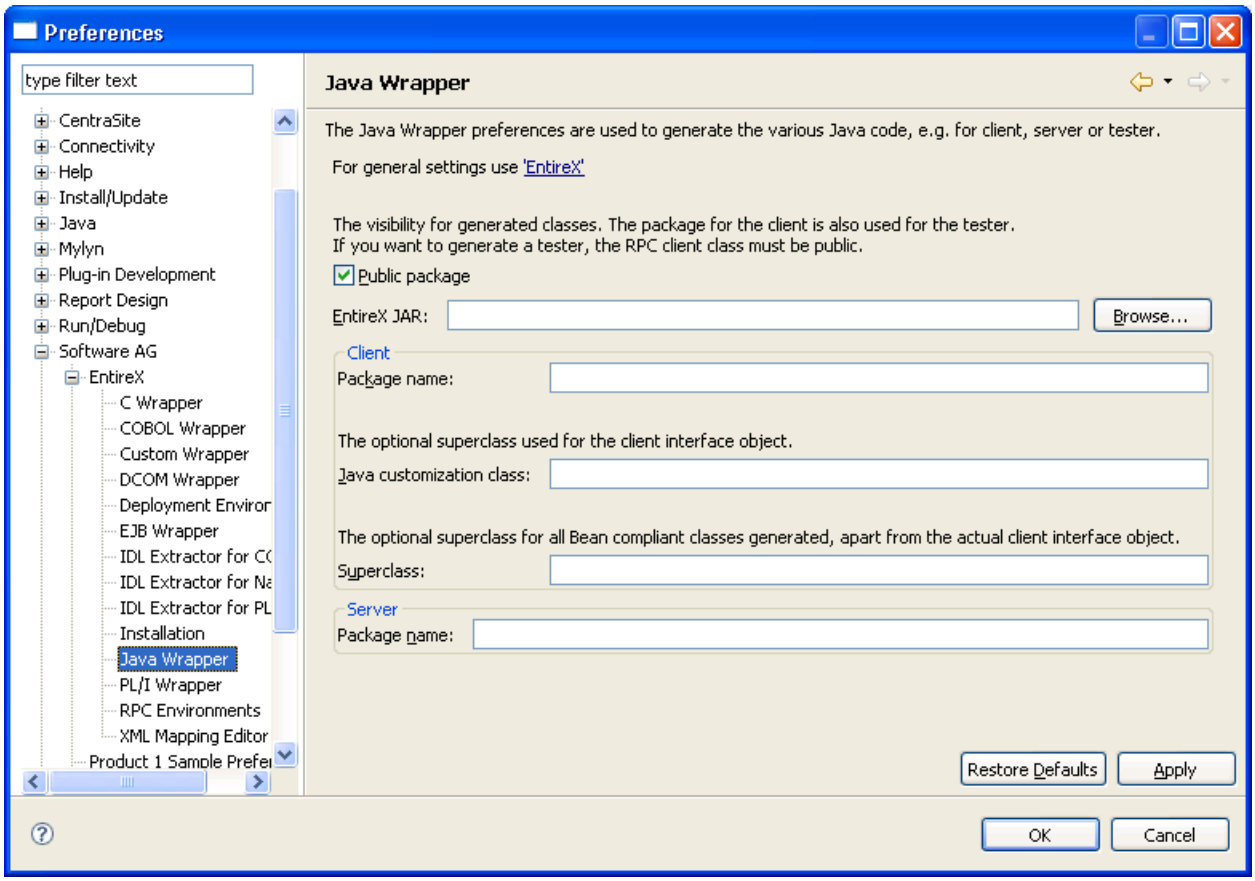
In addition to the standard commands of Eclipse, the context menu of a Java file contains a group of commands for the Java Wrapper.

Command	Description
All	Executes the next three steps described below. The RPC Client (Bean-compliant) will not be executed.
RPC Tester	Generates a client test program.
RPC Server	Generates a Java server class and a server skeleton for your own implementation.
RPC Client	Generates a Java client class.
All (with Bean-compliant client)	Same as **All**, but the generated client will be Bean-compliant.
RPC Client (Bean-compliant)	Generates Java (client) classes instead of inner classes. There is one client class generated for each library in the Software AG IDL file.

Important: If the IDL file is in a Java project, the Java Wrapper uses the project to compile the Java files. If the IDL file is in a simple project, the Java files are generated, but not compiled.

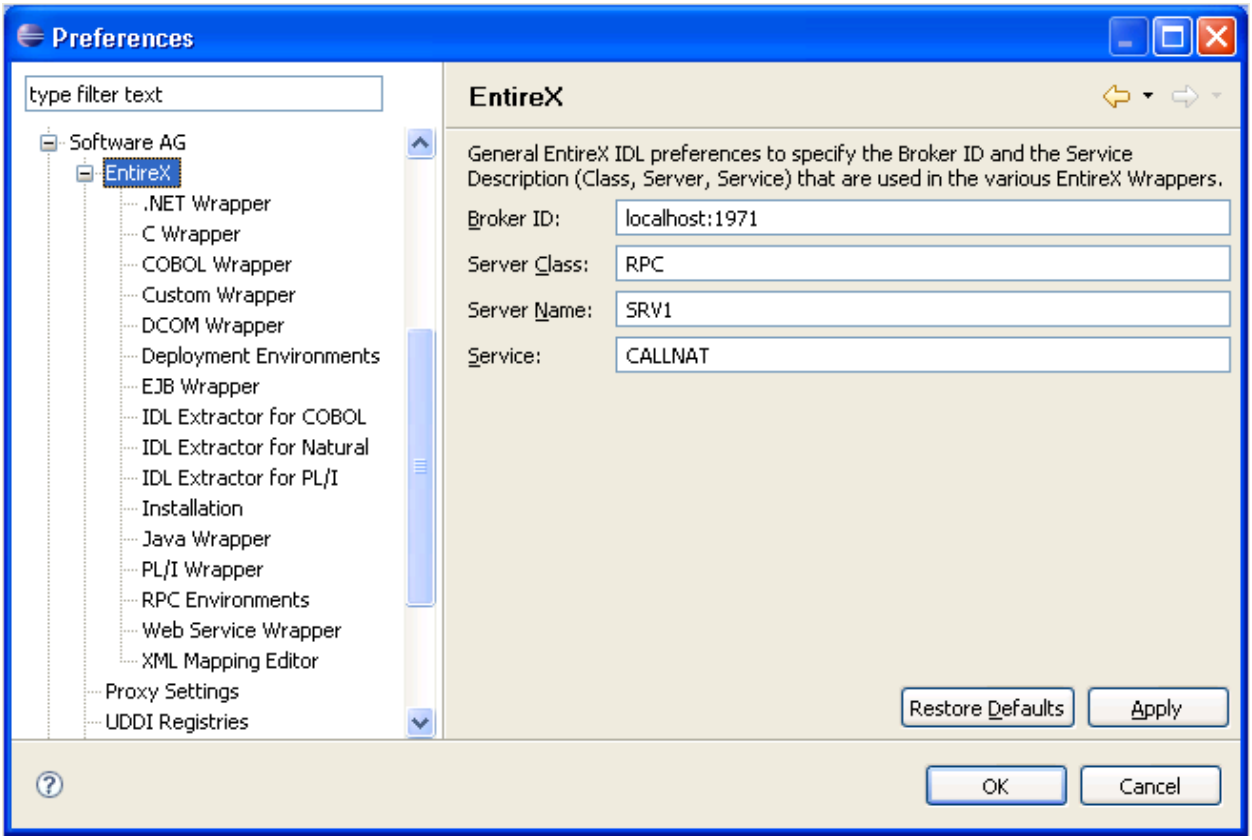
To compile the files generated by the Java Wrapper, file *entirex.jar* must be included in the build path of the project. The Java Wrapper checks whether *entirex.jar* is in the project's build path. If not, the setting for *entirex.jar* on the preference page is added to the build path.

Preferences



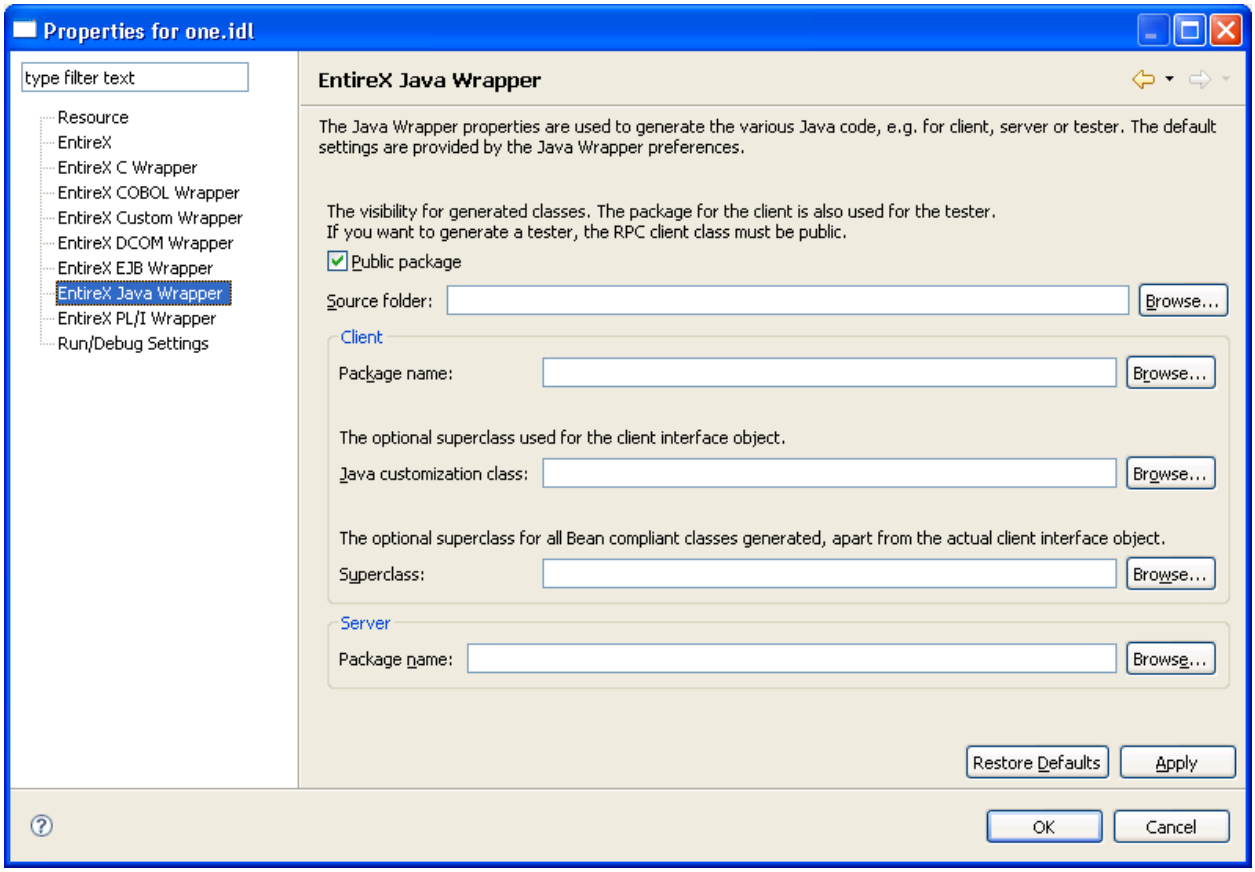
In general, the preferences of the Java Wrapper are used to set the Customization Class and the package name for the RPC client and the RPC server. The package for the client is also used for the tester. If you want to generate a tester, the RPC client class must be public. The **Superclass** field is used to specify an extension class for all Bean-compliant generated classes apart from the actual client interface object.

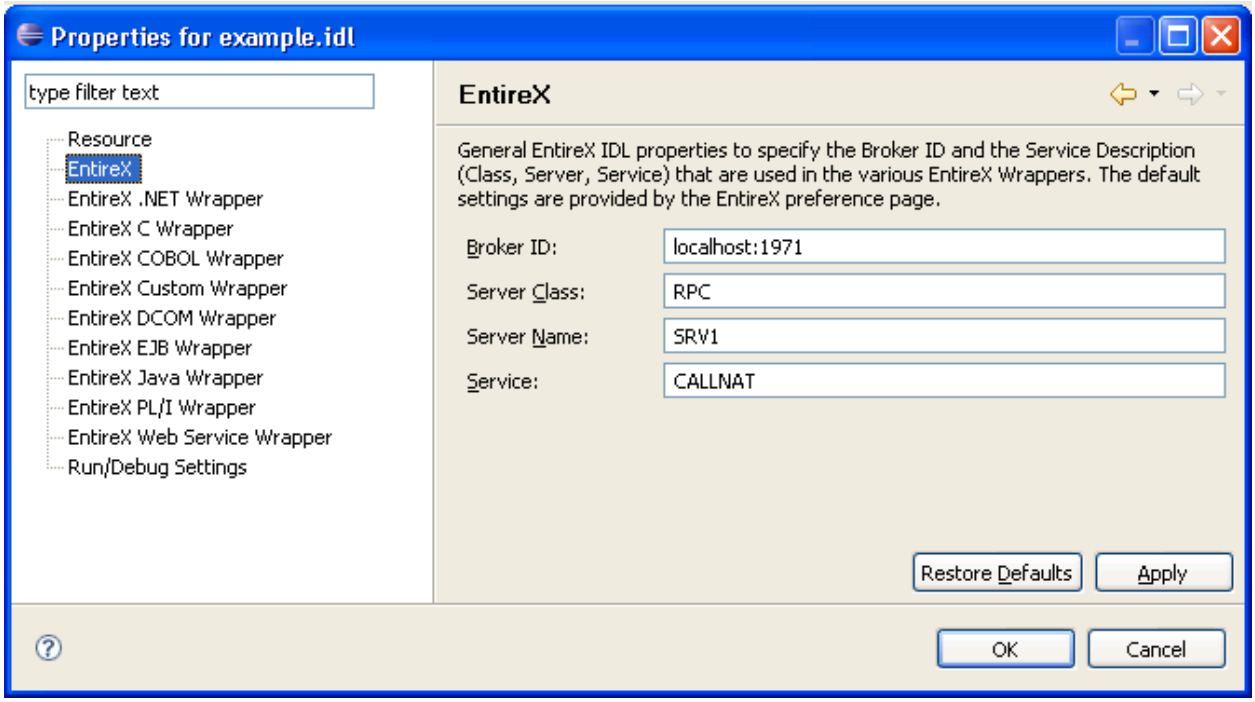
To set the broker ID and the server address for all new IDL files in the workspace, use the preference page “EntireX”.



Properties

For the settings of an individual IDL file, use the properties of this file. The property pages include the same fields to set as the preference pages. In addition, the property page of the Java Wrapper includes the project-specific setting of the source folder. This is the package root of the generated files.


Properties for one.idl
type filter text
Resource
EntireX
EntireX C Wrapper
EntireX COBOL Wrapper
EntireX Custom Wrapper
EntireX DCOM Wrapper
EntireX EJB Wrapper
EntireX Java Wrapper
EntireX PL/I Wrapper
Run/Debug Settings
EntireX Java Wrapper
The Java Wrapper properties are used to generate the various Java code, e.g. for client, server or tester. The default settings are provided by the Java Wrapper preferences.
The visibility for generated classes. The package for the client is also used for the tester.
If you want to generate a tester, the RPC client class must be public.
Public package
Source folder:
Browse...
Client
Package name:
Browse...
The optional superclass used for the client interface object.
Java customization class:
Browse...
The optional superclass for all Bean compliant classes generated, apart from the actual client interface object.
Superclass:
Browse...
Server
Package name:
Browse...
Restore Defaults
Apply
OK
Cancel


Properties for example.idl
type filter text
Resource
EntireX
EntireX .NET Wrapper
EntireX C Wrapper
EntireX COBOL Wrapper
EntireX Custom Wrapper
EntireX DCOM Wrapper
EntireX EJB Wrapper
EntireX Java Wrapper
EntireX PL/I Wrapper
EntireX Web Service Wrapper
Run/Debug Settings
EntireX
General EntireX IDL properties to specify the Broker ID and the Service Description (Class, Server, Service) that are used in the various EntireX Wrappers. The default settings are provided by the EntireX preference page.
Broker ID:
localhost:1971
Server Class:
RPC
Server Name:
SRV1
Service:
CALLNAT
Restore Defaults
Apply
OK
Cancel

Starting the IDL Tester

There are two alternatives for starting the EntireX IDL Tester:

- **From the Context Menu**
  This is the preferred method. In the context menu of the IDL file, choose **Test Software AG IDL...**. A dialog appears for choosing the program to test.

  The IDL Tester is generated and launched as a separate Java Application. See *EntireX IDL Tester* for more details.

- **From Generated Test Program**
  To start the IDL Tester, select the generated test program in the Navigator or Package Explorer and choose **Run** from the context menu or toolbar.

  The IDL Tester is started as a separate application. See ***Using the IDL Tester***.

Generating a Java Client Interface Object

To generate a Java client interface object

1 In the Navigator view or in the Package Explorer, select the Software AG IDL file.

2 From the context menu, choose **Generate Java from Software AG IDL > RPC Client**.

This starts the generation of the Java source. The Java source files are written to the source folder of the IDL file. The source folder is set in the properties of the IDL file.

This starts the generation and compiles the generated Java sources. The Java source files and the class files are written to the directory of the IDL file.

File	Description
<Library name>.java	The Java source code of the generated client interface object. The library name is used to build the file name and the class name. Do not change this file.

If more than one library is defined in the IDL file, separate client interface object files will be generated for each library.

Generating a Java Client Interface Object without inner Classes (Bean-compliant)

When using the Java Wrapper to generate an RPC client (Bean-compliant), the resulting client interface object contains no inner classes. Instead, there will be separate classes generated for each structure within the IDL file.

Note: A superclass to be extended by all the newly generated classes can be specified in the setup menus for *Preferences* and *Properties*.

To generate a Java client interface object (Bean-compliant)

1 Select an IDL file.

2 From the context menu, choose **Generate Java from Software AG IDL > RPC Client (Bean-compliant)**.

As a result, the generation of the Java source is started. The Java source files are written to the source folder of the IDL file and the generated Java sources are compiled.

Note: The source folder can be specified in the setup menu for *Properties*.

The Java source files and class files are written to the directory of the IDL file. The following table gives a short description:

File	Description
<Library name>.java	The Java source code of the generated client interface object. The library name is used to build the file name and the class name. Do not change this file.
<Structure name>.java	A Java class is generated for each structure and group within the input IDL file(s).

Note: If more than one library is defined in the IDL file, separate client interface object files will be generated for each library.

Generating a Java Server Interface Object

To generate a Java server interface object

1 In the Navigator view or the Package Explorer, select the Software AG IDL file.

2 From the **Context** menu, choose **Generate Java from Software AG IDL > RPC Server**.

The Java Wrapper produces the following files for the server interface object in the source folder of the IDL file.

File	Description
<Library name>Stub.java	The Java source code of the generated server interface object. The library name followed by Interface Object is used to build the file name. Do not change this file.
<Library name>Server.java	A Java source file that contains a server skeleton. This is a complete Java class that can be compiled. It contains all methods the server has to implement. Add your application-specific coding in the places marked with the `// insert your application specific code here` comment. The library name followed by "Server" is used to build the file name. If this file exists, it will not be generated.
Abstract<Library name>Server.java	A Java source file that contains the generated part of the server as an abstract class. The server skeleton *<Library name>Server.java* extends this class and contains the application-specific code. Separating the generated code and the application-specific code simplifies re-generation of the RPC server.

If more than one library is defined in the IDL file, separate server interface object files will be generated for each library. The server package name is used as the package name in the generated server files. The server package is part of the Java Wrapper properties of the IDL file. At runtime, configure the server packages in the Java RPC Server configuration. The Java RPC Server uses the library name (which is part of the RPC request from the client) to dynamically load a class named *<Library name>Stub.class*. The RPC server searches for this server interface object class as well as the server class using the actual classpath.

Using the IDL Tester

The client test program is an easy-to-use utility to check whether the remote call works. The client test program supports most of the data types and features of the IDL.

If there is no client interface object already defined, the IDL Tester will generate a Bean-compliant client interface object. However, if there is a previously generated client interface object, it will not be overwritten, regardless if it is Bean-compliant or not.

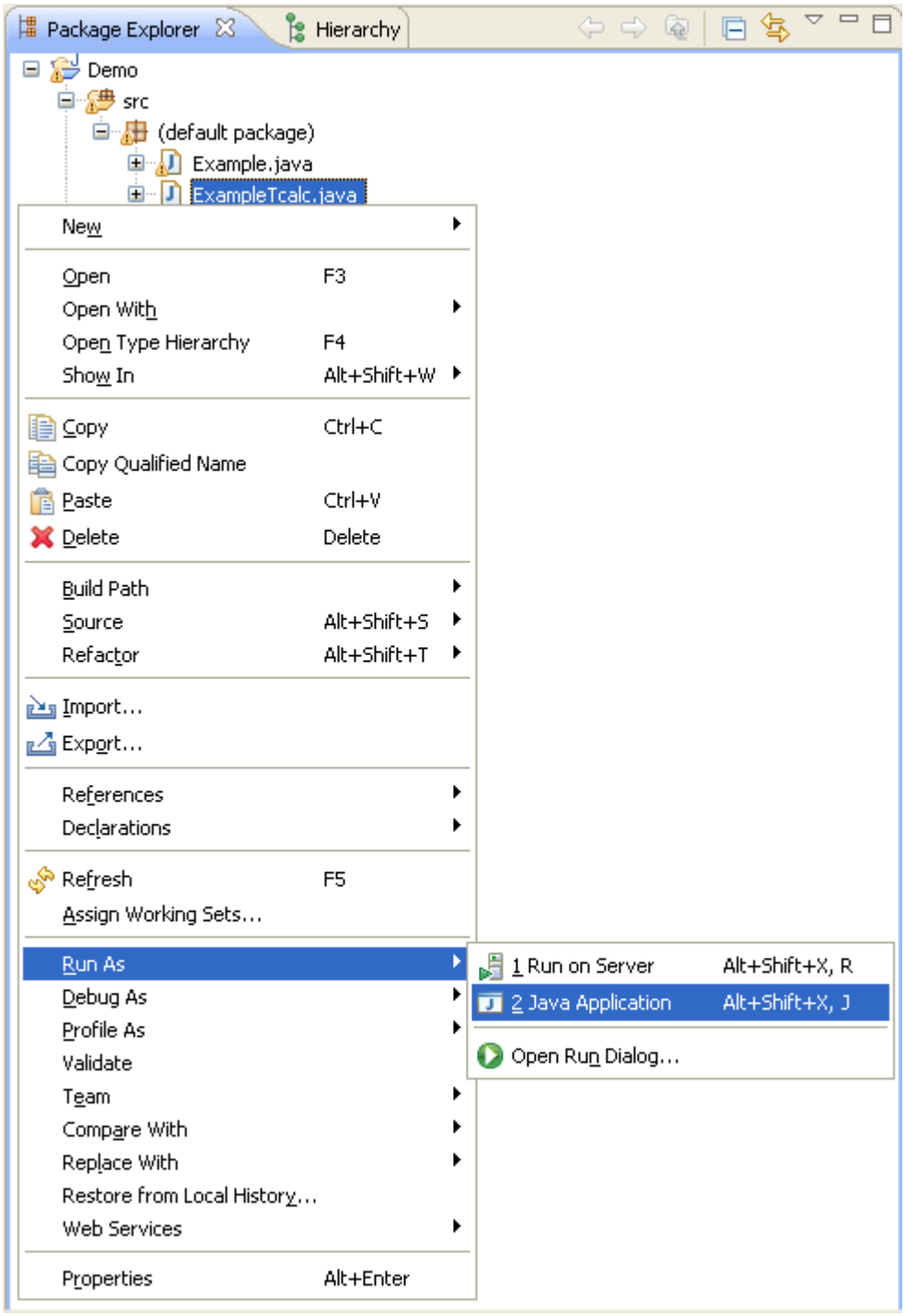
There are two alternatives for generating and running the standard client test program:

- From the context menu of an IDL file. This is the preferred method. See *EntireX IDL Tester* in the EntireX Workbench documentation.
- Using **Generate Java... > RPC Tester**. See below.

This section covers the following topics:

Calling the IDL Tester using Generate Java ... > RPC Tester

1. In the Navigator view or in the Package Explorer, select the Software AG IDL file.
2. From the context menu, choose **Generate Java from Software AG IDL > RPC Tester**. For each program in the IDL file, one class with the name `<Library name>T<program name>.java` is generated. The class `<Library name>T<program name>` can be started as a standalone Java application.



3. In the Navigator view or the Package Explorer, select the file *<Library name>T<program name>.java* and choose **Run As** from the context menu or **Run...** from the Run menu. This creates a launch configuration and starts the tester. See also ***Running the Delivered Examples***.

See *EntireX IDL Tester* in the EntireX Workbench documentation for more information.

Using the IDL Tester in Batch Mode

To start the Tester in Batch mode

- Enter the following command

```
java -classpath <your classpath> <library>T<program> -batch
```

where <your classpath> contains the class of the RPC tester and the file *entirex.jar*.
<library> is the name of the library and
<program> is the name of the program.

For the delivered *example.idl*, the following RPC testers are provided: *ExampleTcalc*, *ExampleThello*, *ExampleTpower*.

An RPC is executed with the default values.

If you add `-both` instead of `-batch`, the GUI of the tester is opened, but the messages and parameter values are written to SYSOUT, too.

To change the broker ID, use `-b <broker id>`. To change the server address, use `-s <class/server/service>`, for example:

```
java ExampleTcalc -b localhost:1971 -s RPC/SRV1/CALLNAT + 3 5
```

To modify the default values

- In the command line add the parameters to the commands.

They will be assigned to the input values one after the other. Enter, for example `java ExampleTcalc + 3 5` to calculate 8.

3 Using the Java Wrapper in Command-line Mode

See *Using the EntireX Workbench in Command-line Mode* for the general command-line syntax.

Command-line Options

Note: The commands `-java:allbeancompliant`, `-java:tester` and `-java:clientbeancompliant` commands will generate a Bean-compliant Java client. To generate client with inner classes (the old way) use `-java:client` or `-java:all` commands.

Task	Command	Option	Description
Generate all Java source files for the specified IDL file(s).	`-java:all`	`-clientpackage`	The client package name for the wrapper class.
		`-customclass`	A non-default superclass of the wrapper class.
		`-help`	Display this usage message.
		`-serverpackage`	The server package name for the server interface object class.
		`-sourcefolder`	The folder for the generated classes.
Generate the JavaBean-compliant Java client(s) for the specified IDL file(s).	`-java:allbeancompliant`	`-clientpackage`	The client package name for the wrapper class.
		`-customclass`	A non-default superclass of the wrapper class.
		`-help`	Display this usage message.
		`-serverpackage`	The server package name for the server interface object class.
		`-sourcefolder`	The folder for the generated classes.
Generate the Java client(s) for the specified IDL file(s).	`-java:client`	`-clientpackage`	The client package name for the wrapper class.
		`-customclass`	A non-default superclass of the wrapper class.
		`-help`	Display this usage message.
		`-public`	Generate a public wrapper class.
		`-sourcefolder`	The folder for the generated classes.
Generate the JavaBean-compliant Java client(s) for the specified IDL file(s).	`-java:clientbeancompliant`	`-clientpackage`	The client package name for the wrapper class.
		`-customclass`	A non-default superclass of the wrapper class.
		`-help`	Display this usage message.

Task	Command	Option	Description
		`-public`	Generate a public wrapper class.
		`-sourcefolder`	The folder for the generated classes.
Generate the Java server(s) for the specified IDL file(s).	`-java:server`	`-help`	Display this usage message.
		`-serverpackage`	The server package name for the server interface object class.
		`-sourcefolder`	The folder for the generated classes.
Generate the Java client(s) and tester(s) for the specified IDL file(s).	`-java:tester`	`-clientpackage`	The client package name for the wrapper class.
		`-customclass`	A non-default superclass of the wrapper class.
		`-help`	Display this usage message.
		`-sourcefolder`	The folder for the generated classes.

Example

```
<workbench> -java:client /Demo/Example.idl -sourcefolder /Demo/src1 -clientpackage ↩
com.client
```

where <*workbench*> is a placeholder for the actual Workbench starter as described under *Using the EntireX Workbench in Command-line Mode*.

The name of the IDL file and the source folder include the project name. In the example, the project *Demo* is used. If the IDL file name describes a file inside the Eclipse workspace, the name is case-sensitive.

If the first part of the IDL file name is not a project name in the current workspace, the IDL file name is used as a file name in the file system. Thus, the IDL files do not need to be part of an Eclipse project.

If the source folder does not exist in the workspace but the first part describing the project exists, the source folder is created.

If the IDL file is located outside the Eclipse workspace, the source folder is also a folder in the file system.

Status and processing messages are written to standard output (stdout), which is normally set to the executing shell window.

Further Examples

Windows

- **Example 1:**

```
<workbench> -java:client C:\Temp\example.idl -sourcefolder src -clientpackage ↩
com.client
```

Uses the IDL file at *C:\Temp\example.idl* and generates the Java source files to the subfolder *src\com\client* of the current working directory.

Output to standard output:

```
Using workspace file:/C:/myWorkspace/.
Processing IDL file C:\Temp\example.idl
Writing to file src/com/client/Example.java.
Exit value: 0
```

- **Example 2:**

```
<workbench> -java:client C:\Temp\*idl -sourcefolder src -clientpackage com.client
```

Generates Java clients for all IDL files in *C:\Temp*.

- **Example 3:**

```
<workbench> -java:client C:\Temp\example.idl -sourcefolder C:\Temp\src ↩
-clientpackage com.client
```

Uses the IDL file at *C:\Temp\example.idl* and generates the Java source files to *C:\Temp\src\com\client*.

- **Example 4:**

```
<workbench> -java:client C:/Temp/example.idl -sourcefolder C:/Temp/src ↩
-clientpackage com.client
```

The same as above. Both slashes and backslashes are permitted.

- **Example 5:**

```
<workbench> -java:client -help
```

or

```
<workbench> -help -java:client
```

Both show a short help for the Java client wrapper.

Linux

- **Example 1:**

```
<workbench> -java:client /Demo/Example.idl -sourcefolder /Demo/src1 -clientpackage ↩
com.client
```

If the project *Demo* exists in the workspace and *Example.idl* exists in this project, this file is used. Otherwise, */Demo/Example.idl* is used from file system.

- **Example 2:**

```
<workbench> -java:client /Demo/*.idl -sourcefolder /Demo/src1 -clientpackage ↩
com.client
```

Generates Java clients for all IDL files in project *Demo* (or in folder */Demo* if the project does not exist). The generated files are in */Demo/src1/com/client*.

- **Example 3:**

```
<workbench> -java:client -help
```

or

```
<workbench> -help -java:client
```

Both show a short help for the Java client wrapper.

4 Software AG IDL to Java Mapping

Mapping IDL Data Types to Java Data Types

In the table below, the following metasymbols and informal terms are used for the IDL.

- The metasymbols [and] surround optional lexical entities.
- The informal term *number* (or *number* [. *number*]) is a sequence of numeric characters, for example 123.

Software AG IDL	Description	Java Data Types	Note
A*number*	Alphanumeric	`String`	1, 3
AV	Alphanumeric variable length	`String`	
AV[*number*]	Alphanumeric variable length with maximum length	`String`	1
B *number*	Binary	`byte[]`	1, 6
BV	Binary variable length	`byte[]`	
BV[*number*]	Binary variable length with maximum length	`byte[]`	1
D	Date	`java.util.Date`	5
F4	Floating point (small)	`float`	2
F8	Floating point (large)	`double`	2
I1	Integer (small)	`byte`	
I2	Integer (medium)	`short`	
I4	Integer (large)	`int`	
K *number*	Kanji	`String`	1
KV	Kanji variable length	`String`	
KV[*number*]	Kanji variable length with maximum length	`String`	1
L	Logical	`boolean`	
N *number* [.*number*]	Unpacked decimal	`java.math.BigDecimal`	4
NU *number* [.*number*]	Unpacked decimal unsigned	`java.math.BigDecimal`	4
P *number* [.*number*]	Packed decimal	`java.math.BigDecimal`	4
PU *number* [.*number*]	Packed decimal unsigned	`java.math.BigDecimal`	4
T	Time	`java.util.Date`	5
U *number*	Unicode	`String`	7
UV	Unicode variable length	`String`	7
UV *number*	Unicode variable length with maximum length	`String`	7

Notes:

1. The field length is given in bytes.

2. If floating-point data types are used, rounding errors can occur. Therefore, the values of sender and receiver might differ slightly.
3. If you use the value null (null pointer) as an input parameter (for `IN` and `INOUT` parameters) for type A, a blank string will be used.
4. If you use the value null (null pointer) as an input parameter (for `IN` and `INOUT` parameters) for types N/P, the value 0 (or 0.0) will be used.
5. If you use the value null (null pointer) as an input parameter (for `IN` and `INOUT` parameters) for types D/T, the current date/time will be used. You change this with the property `entirex.marshal.date`. Setting `entirex.marshal.date=null` will map the value null to the invalid date 0000-01-01 of the RPC marshalling. This is the invalid date value in Natural, too. With this setting the invalid date as an output parameter will be mapped to null. The default is to map the invalid date to 0001-01-01.
6. If you use the value null (null pointer) as an input parameter (for `IN` and `INOUT` parameters) for type B, all binary values will be set to zero.
7. The length is given in 2-byte Unicode code units following the Unicode standard UTF-16. The maximum length is 805306367 code units.

Please note also hints and restrictions on the Software AG IDL data types valid for all programming language bindings. See *IDL Data Types* under *Software AG IDL File* in the IDL Editor documentation.

Mapping Library Name and Alias

The library name as specified in the IDL file is sent from a client to the server. Special characters are not replaced. The library alias is not sent to the server.

In the RPC server, the IDL library name sent may be used to locate the target server. See *Locating and Calling the Target Server* in the platform-specific administration or RPC server documentation.

The library name as given in the library definition of the IDL file is mapped to the class name of the generated Java classes. See `library-definition` under *Software AG IDL Grammar* in the *IDL Editor* documentation. For the server interface object, the names of the class are composed as *`library name`* Interface Object and *`library name`* Server. For the client interface object, no suffix is appended. When the class names are built, the library name is capitalized to match Java naming conventions.

The special characters '#' and '-' in the library name are replaced by the character '_'.

If there is an alias for the library name in the `library-definition` under *Software AG IDL Grammar* in the *IDL Editor* documentation, this alias is used as is to form the client class name. Therefore, this alias must be a valid Java class name. On the server side, the alias is used as is to form the class name of the server class.

Example:

- library name Hu#G-O is converted to Hu_g_o

Mapping Program Name and Alias

The program name is sent from a client to the server. Special characters are not replaced. The program alias is not sent to the server.

In the RPC server, the IDL program name sent is used to locate the target server. See *Locating and Calling the Target Server* in the platform-specific administration or RPC server documentation.

The program name as given in the `program-definition` under *Software AG IDL Grammar* in the *IDL Editor* documentation of the IDL file is mapped to method names within the generated Java classes. To match Java naming conventions the program name is converted to lowercase.

The special characters '#' and '-' in the program name are replaced by the character '_'.

If there is an alias for the program name in the `program-definition` under *Software AG IDL Grammar* in the *IDL Editor* documentation, this alias is used as is for the method name. Therefore, this alias must be a valid Java method name. On the server side, the alias is used as is for the method name in the server class.

Mapping Parameter Names

The parameter names are mapped to fields inside the classes (see ***Mapping the Direction Attributes*** `IN`, `OUT`, ***and*** `INOUT`).

Example:

- parameter name `Hu#G-O` is converted to `hu_g_o`

Mapping Fixed and Unbounded Arrays

Arrays in the IDL file are mapped to Java arrays. If an array value does not have the correct number of dimensions or elements, this will result in a `NullPointerException` or an `ArrayIndexOutOfBoundsException`. If you use the value null (null pointer) as an input parameter (for `IN` and `INOUT` parameters), an array will be instantiated.

Mapping Groups and Periodic Groups

Groups (structures) in the IDL file are mapped to inner classes. If the Bean-compliant generation mode is used, they are mapped to normal classes in their own files. The group members (structure fields) are implemented as public fields of the inner class. If the bean-compliant generation is used, the members (structure fields) are implemented as private fields with getter and setter methods.

Example

The following example shows how to program with groups in a Java client and server. The IDL program consists of three groups, each with the same fields, but with different directions. The client shows how to initialize the fields in the groups for the In and InOut parameters and how to get the results from the Out and InOut parameters. The server part shows only the implemented server method, not the other parts of the generated server skeleton. The server just moves the data from the `In` parameters to the `Out` parameters and fills the gaps. We assume that `ClientGroup.class` and the client interface object Libgroup.class are in the same folder. To compile and run the client and the server you need the *entirex.jar*. For the server we assume that `LibgroupServer.class` and `LibgroupStub.class` are in the same folder and this folder is in the classpath of the EntireX Java RPC Server.

IDL

```
library 'LibGroup' is
    program 'Program1' is
        define data parameter
        1 Group1    (/3)    In Out
            2 Field01   (A10)
            2 Field02   (N2)
            2 Field03   (I4)
        1 Group2    (/1)    In
            2 Field01   (A10)
            2 Field02   (N2)
            2 Field03   (I4)
        1 Group3    (/2)    Out
            2 Field01   (A10)
            2 Field02   (N2)
            2 Field03   (I4)
        end-define
```

Client

```
import com.softwareag.entirex.aci.Broker;
import com.softwareag.entirex.aci.BrokerException;
import java.math.BigDecimal;

public class ClientGroup {
    public static void main(String[] args) {
        try {
            Broker broker = new Broker(Libgroup.DEFAULT_BROKERID, "User1");
            broker.logon();
            // create the wrapper object.
            Libgroup lib = new Libgroup(broker, Libgroup.DEFAULT_SERVER);
            // /*
            //  * Using the old style:
            //  * Get the reference for group1 from wrapper object and
            //  * fill group1 with data. Since group1 is InOut, there exists a
            //  * reference.
            //  */
            // Group1[] group1 = lib.getGroup1();
            // for (int i = 0; i < group1.length; i++) {
            //     // create a new instance of each array element of group1.
            //     group1[i] = new Group1();
            //     // fill the data in each field.
            //     group1[i].setField01("group1 " + i);
            //     group1[i].setField02(new BigDecimal(i));
            //     group1[i].setField03(2 * i);
            // }
      /*
       * Fill the group1 parameters, using the new methods for indexed access.
       */
      Group1[] group1 = lib.getGroup1();
      for (int i = 0; i < group1.length; i++) {
        Group1 group = new Group1();
        group.setField01("group1 " + i);
        group.setField02(new BigDecimal(i));
        group.setField03(2 * i);
        lib.setGroup1(i, group);
      }

            /*
             *  Create an instance for group2. There is no reference for group2
             *  since this is an In parameter. Fill group2 with data.
             */
            Group1[] group2 = new Group1[1];
            for (int i = 0; i < group2.length; i++) {
                // create a new instance of each array element of group2.
                group2[i] = new Group1();
                // fill the data in each field.
                group2[i].setField01("group2 " + i);
                group2[i].setField02(new BigDecimal(i));
```

```
                group2[i].setField03(2 * i);
            }
            // do the RPC.
            lib.program1(group2);

            // /*
            //  * Using the old style:
            //  * We can use the reference group1, it is not modified.
            //  */
            // for (int i = 0; i < group1.length; i++) {
            //     // get the data from the group and print.
            //     System.out.println("Result of Program1; group1[" + i + "] "
            //         + group1[i].getField01() + ", " + group1[i].getField02() + ↩
", "
            //          + group1[i].getField03());
            // }
      /*
       * Retrieve the group1 elements, using the new indexed access method.
       */
      for (int i = 0; i < 3; i++) {
        // get the data from the group and print.
        System.out.println("Result of Program1; group1[" + i + "] "
            + lib.getGroup1(i).getField01() + ", "
            + lib.getGroup1(i).getField02() + ", "
            + lib.getGroup1(i).getField03());
      }
            // /*
            //  * Using the old style:
            //  * Get the reference for group3. group3 is Out.
            //  */
            // Group1[] group3 = lib.getGroup3();
            // for (int i = 0; i < group3.length; i++) {
            //     // get the data from the group and print.
            //     System.out.println("Result of Program1; group3[" + i + "] "
            //         + group3[i].getField01() + ", " + group3[i].getField02() + ↩
", "
            //          + group3[i].getField03());
            // }
      /*
       * Retrieve the group3 elements, using the new indexed access method.
       */
      for (int i = 0; i < 2; i++) {
        // get the data from the group and print.
        System.out.println("Result of Program1; group3[" + i + "] "
            + lib.getGroup3(i).getField01() + ", "
            + lib.getGroup3(i).getField02() + ", "
            + lib.getGroup3(i).getField03());
      }
            broker.logoff();
        } catch (BrokerException excep) {
            excep.printStackTrace ();
        }
```

```
    }

}
```

Client Group (Bean-compliant)

```
import com.softwareag.entirex.aci.Broker;
import com.softwareag.entirex.aci.BrokerException;
import java.math.BigDecimal;

public class ClientGroup {
    public static void main(String[] args) {
        try {
            Broker broker = new Broker(Libgroup.DEFAULT_BROKERID, "User1");
            broker.logon();
            // create the wrapper object.
            Libgroup lib = new Libgroup(broker, Libgroup.DEFAULT_SERVER);
            /* Get the reference for group1 from wrapper object and
             * fill group1 with data. Since group1 is InOut, there exists a
             * reference.
             */
            Group1[] group1 = lib.getGroup1();
            for (int i = 0; i < group1.length; i++) {
                // create a new instance of each array element of group1.
                group1[i] = new Group1();
                // fill the data in each field.
                group1[i].setField01("group1 " + i);
                group1[i].setField02(new BigDecimal(i));
                group1[i].setField03(2 * i);
            }
            /** Create an instance for group2. There is no reference for group2
             *  since this is an In parameter. Fill group2 with data.
             */
            Group1[] group2 = new Group1[1];
            for (int i = 0; i < group2.length; i++) {
                // create a new instance of each array element of group2.
                group2[i] = new Group1();
                // fill the data in each field.
                group2[i].setField01("group2 " + i);
                group2[i].setField02(new BigDecimal(i));
                group2[i].setField03(2 * i);
            }
            // do the RPC.
            lib.program1(group2);
            // We can use the reference group1, it is not modified.
            for (int i = 0; i < group1.length; i++) {
                // get the data from the group and print.
                System.out.println("Result of Program1; group1[" + i + "] "
                   + group1[i].getField01() + ", " + group1[i].getField02() + ", "
                    + group1[i].getField03());
            }
```

```
            // Get the reference for group3. group3 is Out.
            Group1[] group3 = lib.getGroup3();
            for (int i = 0; i < group3.length; i++) {
                // get the data from the group and print.
                System.out.println("Result of Program1; group3[" + i + "] "
                    + group3[i].getField01() + ", " + group3[i].getField02() + ", "
                     + group3[i].getField03());
            }
            broker.logoff();
        } catch (BrokerException excep) {
            excep.printStackTrace ();
        }
    }

}
```

Server

```
public void program1 (LibgroupServer.Program1Group2[] group2) {
        /*
         * Program1Group1 is InOut
         * Program1Group2 is In
         * Program1Group3 is Out
         * Move the values from Program1Group2 to Program1Group1 and move the
         * value from Program1Group1 to Program1Group3.
         */
        int length = Math.min(program1Group1.length, program1Group3.length);
        for (int i = 0; i < length; i++) {
            if (program1Group3[i] == null)
                program1Group3[i] = new Program1Group3();
            program1Group3[i].field01 = program1Group1[i].field01;
            program1Group3[i].field02 = program1Group1[i].field02;
            program1Group3[i].field03 = program1Group1[i].field03;
        }
        for (int i = length; i < program1Group3.length; i++) {
            if (program1Group3[i] == null)
                program1Group3[i] = new Program1Group3();
            program1Group3[i].field01 = "New Text " + i;
            program1Group3[i].field02 = new BigDecimal(10);
            program1Group3[i].field03 = 100 + i;
        }
        // move the values from Program1Group1 to Program1Group3.
        length = Math.min(group2.length, program1Group1.length);
        for (int i = 0; i < length; i++) {
            if (program1Group1[i] == null)
                program1Group1[i] = new Program1Group1();
            program1Group1[i].field01 = group2[i].field01;
            program1Group1[i].field02 = group2[i].field02;
            program1Group1[i].field03 = group2[i].field03;
        }
        for (int i = length; i < program1Group1.length; i++) {
```

```
            if (program1Group1[i] == null)
                program1Group1[i] = new Program1Group1();
            program1Group1[i].field01 = "New Text " + i;
            program1Group1[i].field02 = new BigDecimal(10);
            program1Group1[i].field03 = 100 + i;
        }
    }
```

Mapping Structures

Structures are mapped like Groups. See *Mapping Groups and Periodic Groups*.

Example

The following example shows how to program with structures in a Java client and server. The structures are mapped to inner classes of the interface objects; if Bean-compliant generation is used, the structures are mapped to normal classes in their own file. The IDL program consists of one structure that is used with different directions. In the example above for the groups we have the same fields in each group. This example shows how to simplify this by using a structure. The structure is defined outside the program an references to the structure can be used several times in different programs. The client shows how to initialize the fields in the references of the structure for the In and InOut parameters and how to get the results from the Out and InOut parameters. The server part shows only the implemented server method, not the other parts of the generated server skeleton. The server just moves the data from the `In` parameters to the `Out` parameters and fills the gaps. We assume that `ClientStrct.class` and the client interface object `Libstrct.class` are in the same folder. To compile and run the client and the server you need the `entirex.jar`. For the server we assume that `LibstrctServer.class` and `LibstrctStub.class` are in the same folder and this folder is in the classpath of the EntireX Java RPC Server.

IDL

```
library 'LibStrct' is
    struct 'Struct1' is
        define data parameter
        1 Field01   (A10)
        1 Field02   (N2)
        1 Field03   (I4)
        end-define

   program 'Program1' is
        define data parameter
        1 Ref1  ('Struct1'/3)   In Out
        1 Ref2  ('Struct1'/1)   In
        1 Ref3  ('Struct1'/2)   Out
        end-define
```

Client

```
import com.softwareag.entirex.aci.Broker;
import com.softwareag.entirex.aci.BrokerException;
import java.math.BigDecimal;

public class ClientStrct {
    public static void main(String[] args) {
        try {
            Broker broker = new Broker(Libstrct.DEFAULT_BROKERID, "User1");
            broker.logon();
            // create the wrapper object.
            Libstrct lib = new Libstrct(broker, Libstrct.DEFAULT_SERVER);
            /* create a struct object (as defined in the wrapper object) for the
             * InOut parameter struct1.
             */
            Struct1[] struct1 = new Struct1[3];
            // /*
            //  * Using the old style:
            //  *  fill the struct object with data.
            //  */
            // for (int i = 0; i < struct1.length; i++) {
            //     // create a new array element.
            //     struct1[i] = new Struct1();
            //     struct1[i].setField01("struct1 ");
            //     struct1[i].setField02(new BigDecimal(4 + i));
            //     struct1[i].setField03(i);
            // }
            // // set the struct object in the wrapper object
            // lib.setRef1 (struct1);
      /*
       * Fill the struct1 parameters, using the new methods for indexed access.
       */
            for (int i = 0; i < struct1.length; i++) {
              Struct1 struct = new Struct1();
                struct.setField01("struct1 ");
                struct.setField02(new BigDecimal(4 + i));
                struct.setField03(i);
                lib.setRef1(i, struct);
            }
            /* create a struct object (as defined in the wrapper object) for the
             * In parameter struct2.
             */
            Struct1[] struct2 = new Struct1[1];
            for (int i = 0; i < struct2.length; i++) {
                // create a new array element.
                struct2[i] = new Struct1();
                struct2[i].setField01("struct2 ");
                struct2[i].setField02(new BigDecimal(4 + i));
                struct2[i].setField03(i);
            }
```

```
            // do the RPC.
            lib.program1(struct2);

            // /*
            //  * Using the old style:
            //  * get the data from the InOut parameter struct1.
            //  */
            // for (int i = 0; i < struct1.length; i++) {
            //     // get the data from the struct and print.
            //     System.out.println("Result of Program1, struct1[" + i + "] "
           //        + struct1[i].getField01() + ", "   + struct1[i].getField02() ↩
+ ", "
            //         + struct1[i].getField03());
            // }
      /*
       * Retrieve the ref1 elements, using the new indexed access method.
       */
      for (int i = 0; i < 3; i++) {
        // get the data from the struct and print.
        System.out.println("Result of Program1, struct1[" + i + "] "
            + lib.getRef1(i).getField01() + ", "
            + lib.getRef1(i).getField02() + ", "
            + lib.getRef1(i).getField03());
      }
      // /*
      //  * Using the old style:
      //  * get the struct object for the Out parameter struct3.
      //  */
      // Struct1[] struct3 = lib.getRef3();
      // // get the data from the Out parameter struct3.
      // for (int i = 0; i < struct3.length; i++) {
      //     // get the data from the struct and print.
      //     System.out.println("Result of Program1, struct3[" + i + "] "
      //        + struct3[i].getField01() + ", "   + struct3[i].getField02() + ", "
      //         + struct3[i].getField03());
      // }
      /*
       * Retrieve the ref3 elements, using the new indexed access method.
       */
      for (int i = 0; i < 2; i++) {
        // get the data from the struct and print.
        System.out.println("Result of Program1, struct3[" + i + "] "
            + lib.getRef3(i).getField01() + ", "
            + lib.getRef3(i).getField02() + ", "
            + lib.getRef3(i).getField03());
      }
            broker.logoff();
        } catch (BrokerException excep) {
            excep.printStackTrace ();
        }
```

```
    }
}
```

ClientStrct (Bean-compliant)

```
import com.softwareag.entirex.aci.Broker;
import com.softwareag.entirex.aci.BrokerException;
import java.math.BigDecimal;

public class ClientStrct {
    public static void main(String[] args) {
        try {
            Broker broker = new Broker(Libstrct.DEFAULT_BROKERID, "User1");
            broker.logon();
            // create the wrapper object.
            Libstrct lib = new Libstrct(broker, Libstrct.DEFAULT_SERVER);
            /* create a struct object (as defined in the wrapper object) for the
             * InOut parameter struct1.
             */
            Struct1[] struct1 = new Struct1[3];
            // fill the struct object with data.
            for (int i = 0; i < struct1.length; i++) {
                // create a new array element.
                struct1[i] = new Struct1();
                struct1[i].setField01("struct1 ");
                struct1[i].setField02(new BigDecimal(4 + i));
                struct1[i].setField03(i);
            }
            /* create a struct object (as defined in the wrapper object) for the
             * In parameter struct2.
             */
            Struct1[] struct2 = new Struct1[1];
            for (int i = 0; i < struct2.length; i++) {
                // create a new array element.
                struct2[i] = new Struct1();
                struct2[i].setField01("struct2 ");
                struct2[i].setField02(new BigDecimal(4 + i));
                struct2[i].setField03(i);
            }
            // set the struct object in the wrapper object
            lib.setRef1 (struct1);
            // do the RPC.
            lib.program1(struct2);
            // get the struct object for the Out parameter struct3.
            Struct1[] struct3 = lib.getRef3();
            // get the data from the InOut parameter struct1.
            for (int i = 0; i < struct1.length; i++) {
                // get the data from the struct and print.
                System.out.println("Result of Program1, struct1[" + i + "] "
                   + struct1[i].getField01() + ", "   + struct1[i].getField02() + ↩
", "
```

```
                    + struct1[i].getField03());
            }
            // get the data from the Out parameter struct3.
            for (int i = 0; i < struct3.length; i++) {
                // get the data from the struct and print.
                System.out.println("Result of Program1, struct3[" + i + "] "
                  + struct3[i].getField01() + ", "   + struct3[i].getField02() + ↩
", "
                    + struct3[i].getField03());
            }
            broker.logoff();
        } catch (BrokerException excep) {
            excep.printStackTrace ();
        }
    }
}
```

Server

```
public void program1 (Struct1[] ref2) {
        /*
         * Program1Group1 is InOut
         * Program1Group2 is In
         * Program1Group3 is Out
         * Move the values from Program1Group2 to Program1Group1 and move the
         * value from Program1Group1 to Program1Group3.
         */
        int length = Math.min(program1Ref1.length, program1Ref3.length);
        for (int i = 0; i < length; i++) {
            if (program1Ref3[i] == null)
                program1Ref3[i] = new Struct1();
            program1Ref3[i].field01 = program1Ref1[i].field01;
            program1Ref3[i].field02 = program1Ref1[i].field02;
            program1Ref3[i].field03 = program1Ref1[i].field03;
        }
        for (int i = length; i < program1Ref3.length; i++) {
            if (program1Ref3[i] == null)
                program1Ref3[i] = new Struct1();
            program1Ref3[i].field01 = "New Text " + i;
            program1Ref3[i].field02 = new BigDecimal(10);
            program1Ref3[i].field03 = 100 + i;
        }

       length = Math.min(ref2.length, program1Ref1.length);
        for (int i = 0; i < length; i++) {
            if (program1Ref1[i] == null)
                program1Ref1[i] = new Struct1();
            program1Ref1[i].field01 = ref2[i].field01;
            program1Ref1[i].field02 = ref2[i].field02;
            program1Ref1[i].field03 = ref2[i].field03;
        }
```

```
        for (int i = length; i < program1Ref1.length; i++) {
            if (program1Ref1[i] == null)
                program1Ref1[i] = new Struct1();
            program1Ref1[i].field01 = "New Text " + i;
            program1Ref1[i].field02 = new BigDecimal(10);
            program1Ref1[i].field03 = 100 + i;
        }
    }
```

Mapping the Direction Attributes IN, OUT, and INOUT

The IDL syntax allows you to define parameters as `IN` parameters, `OUT` parameters, or `IN OUT` parameters (which is the default if nothing is specified). This direction specification is reflected in the generated Java interface object as follows:

- `IN` parameters are sent from the RPC client to the RPC server. `IN` parameters are implemented as parameters of the generated method.
- `OUT` parameters are sent from the RPC server to the RPC client. `OUT` parameters are implemented as read-only properties. A `getMethod` is generated for each `OUT` parameter.
- `INOUT` parameters are sent from the RPC client to the RPC server and then back to the RPC client. `INOUT` parameters are implemented as properties. A `setMethod` and a corresponding `getMethod` is generated for each `INOUT` parameter.

Note that only the direction information of the top-level fields (level 1) is relevant. Group fields always inherit the specification from their parent. A different specification is ignored.

See the `attribute-list` under *Software AG IDL Grammar* in the *IDL Editor* documentation for the syntax on how to describe attributes in the IDL file and refer to the `direction` attribute.

Mapping the aligned Attribute

The `aligned` attribute is not relevant for the programming language Java. However, a Java client can send the `aligned` attribute to an EntireX RPC server, where it might be needed.

See the `attribute-list` under *Software AG IDL Grammar* in the *IDL Editor* documentation for the syntax on how to describe attributes in the IDL file and refer to the `aligned` attribute.

Calling Servers as Procedures or Functions

The IDL syntax allows definition of procedures only. It does not have the concept of a function. A function is a procedure which, in addition to the parameters, returns a value. Procedures and functions are transparent between clients and server, i.e. a client using a function can call a server implemented as a procedure and vice versa. In Java a procedure corresponds to a method with result type void, a function returns a value of some type.

It is possible to treat the `OUT` parameter of a procedure as the return value of a function. The Java Wrapper generates a method with a non-void result type when the following conditions are met:

- the last parameter of the procedure definition is of type `OUT`;
- this last parameter of the procedure definition has the name `Function_Result`. The name `Function_Result` is not case-sensitive.

Of course, in this case `getMethod` is not generated for this `OUT` parameter.

As an example, see the Java Wrapper example that comes with EntireX.

II Writing Applications with the Java Wrapper

Writing Simple Applications with the Java Wrapper

- *Required Steps*
- *Java Wrapper Constructors*
- *Generated Java Wrapper Methods*

Writing Advanced Applications - Java Wrapper

- *Natural Logon or Changing the Library Name*
- *Customizing the Generated Java Classes*
- *Using RPC Compression*
- *Using Conversational RPC*
- *Using Natural Security*
- *Support of DVIPA*

Writing RPC Clients for the RPC-ACI Bridge in Java

5 Writing Simple Applications with the Java Wrapper

Required Steps

Interaction with the Java Wrapper occurs through instantiating objects of different classes, invoking their methods and manipulating their inner state. The basic steps for writing a client are listed below. For details, see the examples delivered with EntireX (*Delivered Java Wrapper Examples*). Methods and properties to interact with the EntireX Broker are completely inherited from the EntireX Java ACI. The EntireX Java ACI also contains the class `RPCService` used as the superclass by the generated Java Wrapper classes.

Basic Steps:

- Instantiate a Broker object.

  One object instance represents one session to an EntireX Broker on your network. If you want to work with multiple EntireX Brokers or with multiple sessions, create one object for each session to an EntireX Broker.
- Use the Broker object to log the application on to EntireX Broker.
- Instantiate the generated Java Wrapper object (see *Java Wrapper Constructors*).
- Use the Java Wrapper methods (see *Generated Java Wrapper Methods*) to call the server programs and access their parameters.

Java Wrapper Constructors

Two constructors are available for the generated Java Wrapper class:

- public Example (Broker broker)
- public Example (Broker broker, String serverAddr)

public Example (Broker broker)

This constructor requires an instantiated Broker object only. The server address used is specified in the properties of the IDL file. Each generated Java Wrapper class has two public static String constants which contain the default values of the Broker and the server as set in the properties of the IDL file. For example:

```
public static final String DEFAULT_BROKERID = "localhost";
public static final String DEFAULT_SERVER = "RPC/SRV1/CALLNAT";
```

A Java Wrapper object using the default settings may be instantiated with the following coding:

```
Broker broker = new Broker(Example.DEFAULT_BROKERID, "UserId");
Example myExample = new Example(broker);
```

public Example (Broker broker, String serverAddr)

This constructor requires an instantiated Broker object and the server address. A Java Wrapper object can be instantiated with the following coding:

```
Broker broker = new Broker("localhost", "UserId");
Example myExample = new Example(broker, "RPC/MYRPC/CALLNAT");
```

Generated Java Wrapper Methods

EntireX Interface Object Version Information

To get the version information of the generated interface object, use the method `getStubVersion()`. It is implemented in the RPC client and server interface objects. The method returns a version string.

Example:

```
"EntireX RPC for Java Interface Object Version=8.2.0, Patch Level=0"
```

Application Identification

The application identification is sent from the application to the Broker. It is visible with Broker Command and Info Services. The identification consists of four parts: name, node, type, and version. These four parts are sent with each Broker call and are visible in trace information.

For the Java Wrapper these values are:

- **Application name**
  `ANAME=Java Runtime`
- **Node name**
  `ANODE=<host name>`

- **Application type**
  `ATYPE=Java`
- **Version**
  `AVERS=8.2.0.0`

The application is allowed to set the application name with the method `Broker.setApplicationName(String)`.

See `setApplicationName` of class `Broker` in the Javadoc documentation of the Java ACI for more information.

6 Writing Advanced Applications - Java Wrapper

Each generated Java Wrapper class inherits methods from the EntireX Java ACI.

This section describes what can be performed with the methods inherited from the class RPCService.

Natural Logon or Changing the Library Name

The library name sent with the RPC request to the EntireX RPC or the Natural RPC Server is specified in the Software AG IDL file (see `library-definition` under *Software AG IDL Grammar* in the *IDL Editor* documentation). When the RPC is executed, this library name can be overwritten.

To overwrite the library, a C Wrapper client must

- Call the `setLibraryName` method of the generated Java Wrapper class with the new library name as a parameter.

To force the library to be considered by Natural RPC Server

- Call the `setNaturalLogon` method of the generated Java Wrapper class with the parameter set to True.

Caution: Natural and EntireX RPC servers behave differently regarding the library name.

See *Natural Logon or Changing the Library Name*.

Customizing the Generated Java Classes

You can extend the generated Java Wrapper Class of the client. By default, the generated client class is a subclass of `com.softwareag.entirex.aci.RPCService`. The customization component allows you to specify a class used as the superclass of the generated client class. This user-defined class (customization class) must be a subclass of `com.softwareag.entirex.aci.RPCService`.

When a customization class is specified, the calls to the user-exit methods `onEnter`, `onLeave`, `onException` and `onRetry` are generated.

To generate a customized Java Wrapper client

1 Implement your customization class. If you use a package for your customization class, specify package and class in the following step. Place the source for the customization class in the package folder, using the folder of the IDL file as package-root. The customization class needs a default constructor and one additional constructor with 4 arguments. See the example below.

2 Specify the name of your customization class in the *EntireX Workbench*, under **Tools**, **Options**, **Java**. This name is stored in the *entirex.properties* file (which is in your home directory) using the key `entirex.wrapper.custom.class`.

3 Generate the wrapper client classes

To use the customized Java Wrapper client

- Add (public) arbitrary methods and fields to your customization class. These methods and fields are inherited by the generated client class. Add your own processing instructions to these methods.

To perform all Broker-related processing in the generated Java Wrapper client

1 Overwrite the constructors of `RPCService`. You can instantiate wrapper classes without specifying a Broker object and server address as a parameter. Use the method `setbroker()` to set or change the reference to the Broker object, and the method `setServerAddress()` to set or change the server address.

2 Use the four user exit methods `onEnter`, `onLeave`, `onException` and `onRetry`. These methods have default implementations in `RPCService` and can be overwritten in your customization class. These exits are called at the beginning and the end of each generated method of the Java Wrapper class and when a broker exception is thrown. See `RPCService` in the Javadoc documentation of the Java ACI.

Example of a Customization Class

```
package ExamplePackage;

import com.softwareag.entirex.aci.Broker;
import com.softwareag.entirex.aci.BrokerException;
import com.softwareag.entirex.aci.RPCService;

public class ExampleCustomization extends RPCService {
   public ExampleCustomization ()
   {
      super();
   }
   public ExampleCustomization (Broker broker, String serverAddr, String
libName, boolean compress)
   {
      super(broker, serverAddr, libName, compress);
   }
   protected void onEnter(String progname) throws BrokerException {
      // insert your implementation here.
   }

   protected void onLeave(String progname, int sendLen, int receiveLen) throws
```

```
BrokerException {
      // insert your implementation here.
   }

   protected void onException(String progname, BrokerException exception) throws
BrokerException {
      // insert your implementation here.
   }

   protected boolean onRetry(String progname, BrokerException exception) throws
BrokerException {
      // insert your implementation here.
      return false;
   }
}
```

Using RPC Compression

EntireX and Natural RPC support a feature called RPC compression to reduce network traffic. The default for compression is on. See *RPC Compression*.

To switch compression on and off

- Use the `setCompression` method of the class `RPCService`.

To check the current compression setting

- Use the `getCompression` method of the class `RPCService`.

Using Conversational RPC

It is assumed that you are familiar with the concepts of conversational and non-conversational RPC. See *Conversational RPC*.

To enable conversational RPC

1. Create a Conversation object and set this with `setConversation` on the wrapper object.
2. Different wrapper objects can participate in the same conversation if they use the same instance of a conversation object.

To abort a conversational RPC communication

- Abort an RPC conversation by calling the `closeConversation` method

To close and commit a conversational RPC communication

- Commit the RPC conversation by calling the `closeConversationCommit` method.

Caution: Natural RPC Servers and EntireX RPC Servers behave differently when ending an RPC convsersation.

See *Conversational RPC*.

Using Natural Security

A Natural RPC Server may run under Natural Security to protect RPC requests. See *Natural Security*.

To authenticate a Java Wrapper client against Natural Security

- Specify a user ID and password in the `logon` method of class `Broker`.

  If different user IDs and/or passwords are used for EntireX Security and Natural Security, use the methods `setRPCUserId` or `setRPCPassword` to set the user IDs and/or passwords for Natural Security.

To force a Java Wrapper client to log on to a specific Natural library

1. Call the `setLibraryName` method of the generated wrapper objects with the new library name as a parameter.
2. Call the `setNaturalLogon` method of the generated wrapper objects with the parameter set to true.

See also ***Natural Logon or Changing the Library Name***.

Example:

Assume that library is a wrapper object that is generated from an IDL library. This object extends `com.softwareag.entirex.aci.RPCService`. For this object, call the methods as shown:

```
library.setRPCUserId("testuser");
library.setRPCPassword("password");
library.setLibraryName("NATLIB");  // this is necessary only if the Natural Library
                                   // name is different from the library name in ↩
the IDL.
library.setNaturalLogon(true);
```

The order of the four methods is arbitrary.

Support of DVIPA

A TCP/IP connection established between stub and broker is not exclusively assigned to a particular thread. With multi-threaded applications, two or more threads may use the same connection. On the other hand, if a connection is busy, another new one is created to exchange data.

In order to access the same z/OS broker instance in a DVIPA-controlled environment, an affinity between application thread and TCP/IP connection is needed to always use the same connection within an application thread. Therefore, an environment variable is evaluated to control the handling of TCP/IP connections.

If broker ID contains the parameter "`poolsize=0`" (e.g. `ETB001?poolsize=0`), an affinity between threads and TCP/IP connections is established. All requests to one particular broker will use the same TCP/IP connection.

See also *Support of Clustering in a High Availability Scenario* under *Administration of Broker Stubs* in the platform-specific administration documentation.

7 Writing RPC Clients for the RPC-ACI Bridge in Java

The RPC-ACI Bridge enables RPC-based client applications to be used with ACI servers.

The EntireX RPC-ACI Bridge reports errors from the RPC server side and the ACI side to the RPC clients. Errors from the ACI side include errors by the Broker for ACI. The RPC-ACI Bridge reports the same error classes and error codes for the RPC server side as the XML/SOAP RPC Server. The RPC-ACI Bridge reports errors of the ACI side in a client-specific way as described below.

To write a Java client

1 Generate the Java RPC client stub from the IDL file as described in ***Using the Java Wrapper***.

2 Implement the client with this stub.

All errors are reported as `BrokerExceptions`. Errors on the ACI side of the RPC-ACI Bridge are `BrokerExceptions` in class 1018. See *Message Class 1018 - EntireX RPC-ACI Bridge* under *Error Messages and Codes*.

III

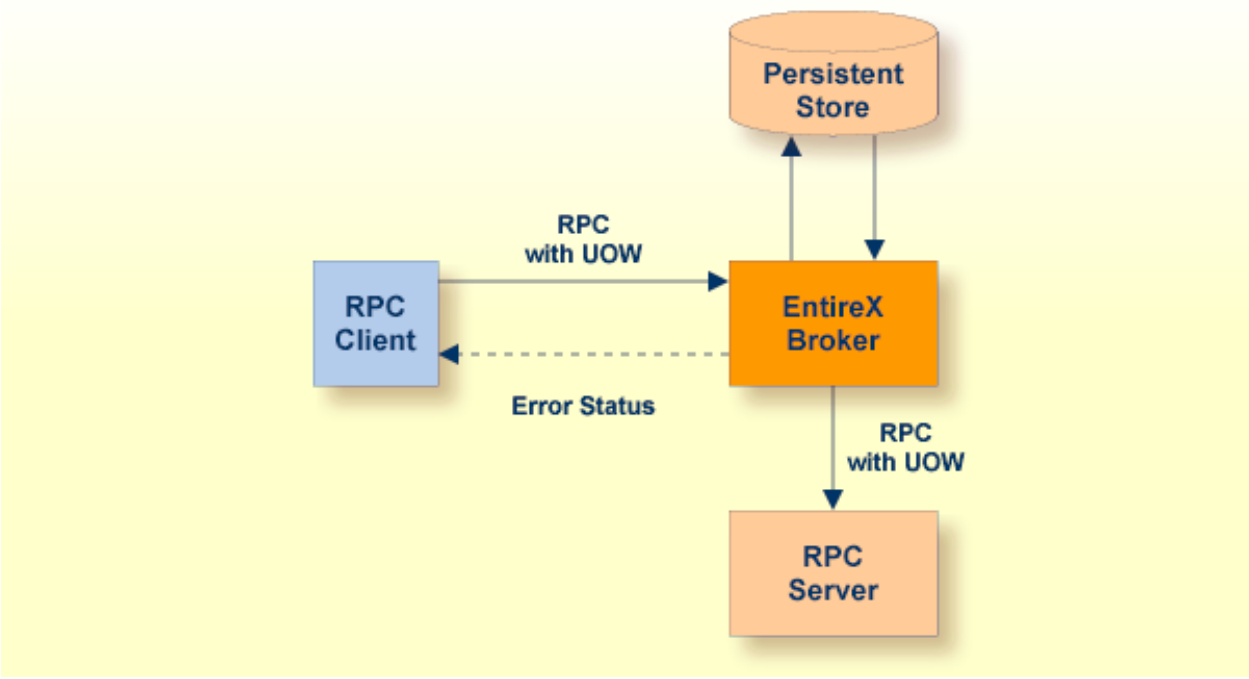
8 Reliable RPC for Java Wrapper

Introduction to Reliable RPC

In the architecture of modern e-business applications (such as SOA), loosely coupled systems are becoming more and more important. Reliable messaging is one important technology for this type of system.

Reliable RPC is the EntireX implementation of a reliable messaging system. It combines EntireX RPC technology and persistence, which is implemented with units of work (UOWs).

- Reliable RPC allows asynchronous calls (“fire and forget”)
- Reliable RPC is supported by most EntireX wrappers
- Reliable RPC messages are stored in the Broker's persistent store until a server is available
- Reliable RPC clients are able to request the status of the messages they have sent



Reliable RPC is used to send messages to a persisted Broker service. The messages are described by an IDL program that contains only `IN` parameters. The client interface object and the server interface object are generated from this IDL file, using the EntireX Java Wrapper.

Reliable RPC is enabled at runtime. The client has to set one of two different modes before issuing a reliable RPC request:

- `AUTO_COMMIT`
- `CLIENT_COMMIT`

While AUTO_COMMIT commits each RPC message implicitly after sending it, a series of RPC messages sent in a unit of work (UOW) can be committed or rolled back explicitly using CLIENT_COMMIT mode.

The server is implemented and configured in the same way as for normal RPC.

Writing a Client

All methods for reliable RPC are available on the interface object. See RPCService in the Javadoc documentation of the Java ACI for details. The methods are:

- RPCService.setReliable
- RPCService.getReliable
- RPCService.reliableCommit
- RPCService.reliableRollback
- RPCService.getMessageId
- RPCService.getStatusOfMessage

Example (this example is included as source in the *examples/RPC/reliable/JavaClient* folder):

Create Broker object and interface object.

```
Broker broker = new Broker(Mail.DEFAULT_BROKERID, userID);
Mail mail = new Mail(broker);
broker.logon();
```

Enable reliable RPC with CLIENT_COMMIT

```
  mail.setReliable(RPCService.RELIABLE_CLIENT_COMMIT);
```

The first RPC message.

```
  mail.sendmail("mail receiver", "Subject 1", "Text 1");
```

Check the status: get the message ID first and use it to retrieve the status.

```
String messageID = mail.getMessageID();
String messageStatus = mail.getStatusOfMessage(messageID);
System.out.println("Status: " + messageStatus + ", id: " + messageID);
```

The second RPC message.

```
mail.sendmail("mail receiver", "Subject 2", "Text 2");
```

Commit the two messages.

```
mail.reliableCommit();
```

Check the status again for the same message ID.

```
messageStatus = mail.getStatusOfMessage(messageID);
System.out.println("Status: " + messageStatus + ", id: " + messageID);
```

The third RPC message.

```
mail.sendmail("mail receiver", "Subject 3", "Text 3");
```

Check the status: get the new message ID and use it to retrieve the status.

```
messageID = mail.getMessageID();
messageStatus = mail.getStatusOfMessage(messageID);
System.out.println("Status: " + messageStatus + ", id: " + messageID);
```

Roll back the third message and check status.

```
mail.reliableRollback();
messageStatus = mail.getStatusOfMessage(messageID);
System.out.println("Status: " + messageStatus + ", id: " + messageID);
broker.logoff();
```

Limitations

1. All program calls that are called in the same transaction (`CLIENT_COMMIT`) must be in the same IDL library.
2. It is not allowed to switch from `CLIENT_COMMIT` to `AUTO_COMMIT` in a transaction.
3. Messages (IDL programs) have `IN` parameters only.

Writing a Server

The server implementation consist of the four classes:

- `Abstract<IDL library name>Server`
- `<IDL library name>`
- `<IDL library name>Server`
- `<IDL library name>Stub`

Add your implementation to the class `<IDL library name>Server`. There are no server-side methods for reliable RPC. The server does not send back a message to the client. The server can run deferred, thus client and server do not necessarily run at the same time. If the server fails, it throws an exception. This causes a cancel of the transaction (unit of work inside the Broker) and the error code is written to the user status field of the unit of work.

Broker Configuration

A Broker configuration with `PSTORE` is recommended. This enables the Broker to store the messages for more than one Broker session. These messages are still available after Broker restart. The attributes `STORE`, `PSTORE`, and `PSTORE-TYPE` in the Broker attribute file can be used to configure this feature. The lifetime of the messages and the status information can be configured with the attributes `UWTIME` and `UWSTAT-LIFETIME`. Other attributes such as `MAX-MESSAGES-IN-UOW`, `MAX-UOWS` and `MAX-UOW-MESSAGE-LENGTH` may be used in addition to configure the units of work. See *Broker Attributes* in the administration documentation.

The result of the method `RPCService.getStatusOfMessage` depends on the configuration of the unit of work status lifetime. If the status is not stored longer than the message, the method returns `(not available)`.

9 Java Wrapper Examples

Delivered Java Wrapper Examples

This section describes the examples for the Java Wrapper folder *examples/java_wrapper* of the Developer's Kit.

Scope

This folder *examples/RPC/basic/example/JavaClient* contains an example of a standalone application (*MyClient.java*) that calls remote procedures CALC and SQUARE with its associated IDL file *example.idl*. An example implementation of a server interface object for the Java RPC Server is available too (*ExampleServer.java*). This server interface object runs with the generic Java RPC Server, which is part of the Java Runtime.

To run the examples from within the Eclipse IDE, import the folder *examples/RPC/basic/example/JavaClient* into a Java project, then add *entirex.jar* to the build path of this project.

This example consists of the following programs, which may be called remotely using the EntireX Broker:

- **CALC**
  Uses two operands and one operator (+-*/) to return the result. A Java client will be able to call our sample method like this:

```
Example myExample = new Example(broker);
  int op1 = 1234;
  int op2 = 5678;
  int y = myExample.calc ("+", op1, op2)
```

- **SQUARE**
  Uses one input value to return its square.

Software AG IDL File and Interface Object Generation Process for the Example

The IDL file describes the interface. See *Software AG IDL File* in the IDL Editor documentation. For the mapping between IDL data types and types of Java see ***Mapping IDL Data Types to Java Data Types***.

This IDL file *example.idl* of the Java Wrapper example is part of the EntireX examples.

```
Library 'EXAMPLE' Is
   Program 'CALC' Is
      Define Data Parameter
         1 Operation       (A1)  In
         1 Operand_1       (I4)  In
         1 Operand_2       (I4)  In
         1 Function_result (I4)  Out
      End-Define

   Program 'SQUARE' Is
      Define Data Parameter
         1 Operand         (I4)  In
         1 Result          (I4)  Out
      End-Define
```

Generating a client interface object with the Java Wrapper creates the following file in the source folder of the IDL file: *Example.java*.

Generating a server interface object with the Java Wrapper produces the following files in the source folder of the IDL file: *ExampleServer.java*, *ExampleStub.java*, *AbstractExampleServer.java*.

Other EntireX Developer's Kit Wrapper Examples

Other Wrappers of the Developer's Kit provide the same examples (CALC and SQUARE). The examples can be mixed, meaning any client can call any server. Natural RPC is also fully compatible with EntireX RPC. A Natural client can call any EntireX RPC server and vice versa.

For examples of other programming language bindings see:

- *Delivered Examples for the C Wrapper* in the C Wrapper documentation
- *Delivered Examples for the COBOL Wrapper*
- Delivered Examples for Natural in subdirectory Java Wrapper, Broker RPC/Client and Broker RPC/Server.

Running the Delivered Examples

Prerequisites for Running the Examples

1. Verify that the Java classpath contains both the *entirex.jar* file, which is located in the classes directory, and an entry for the directory containing the generated classes.
2. To run the client programs, an RPC server is needed. You may use the example RPC server under CICS, UNIX and Windows. If you want to use a Natural RPC Server copy the **.nsn* files to a Natural library. You can also use the Java RPC Server under UNIX and Windows.

To run the client example

1. Generate the client interface object.
2. Compile the *MyClient.java* file.
3. Run *MyClient.class*.
4. Select one of the examples with the option button and press **Call()**.
5. If you want to use EntireX Security, uncomment the line

```
//broker.setSecurity(new EntireXSecurity(), false);
```

To run the server example

1. Generate the RPC server as described under ***Generating a Java Server Interface Object***.
2. Implement the methods in <*library name*>*Server.java.*
3. Run the Java RPC Server. The classpath must contain the directory of the server interface object classes. If you start the Java RPC Server in the current directory, then add "." to the classpath. If the Java RPC Server is started in a different directory, the complete pathname of the Java Wrapper example has to be part of the classpath. Otherwise the Java RPC Server cannot load the server interface object classes.
4. To shut down the Java RPC Server, use the System Management Hub. (Note that this stops all RPC servers that register the same service.)