Server Programming with the SAP Java Connector (JCo)

Thomas G. Schuessler

Thomas G. Schuessler is the founder of ARAsoft, a company offering products, consulting, custom development, and training to customers worldwide, specializing in integration between SAP and non-SAP components and applications. Thomas is the author of SAP's BIT525 and BIT526 classes. Prior to founding ARAsoft in 1993, he worked with SAP AG and SAP America for seven years.

(complete bio appears on page 46)

It should be observed with what facility mice attach themselves to man.
– Charles Darwin, Notebooks, p. 354

In most integration projects using the SAP Java Connector (JCo), the Java program is the client and the ABAP-based SAP system (e.g., R/3) is the server. But in some cases, you would like an ABAP program to invoke a Java component, because it offers some functionality that would benefit your ABAP program. If that possibility sounds enticing to you, and you are a Java programmer, then this article is for you.

The first part of the article will briefly describe the main usage scenarios that apply to RFC server programming. Then I will show you what you have to do in your SAP system in order to support RFC[1] calls to an external, non-SAP component. This will be followed by an introduction to those aspects of JCo that are relevant for server programming, in other words, experience with JCo is not a prerequisite for mastering the techniques described in this article. Finally, I will introduce and discuss a sample program for the most relevant usage scenario.

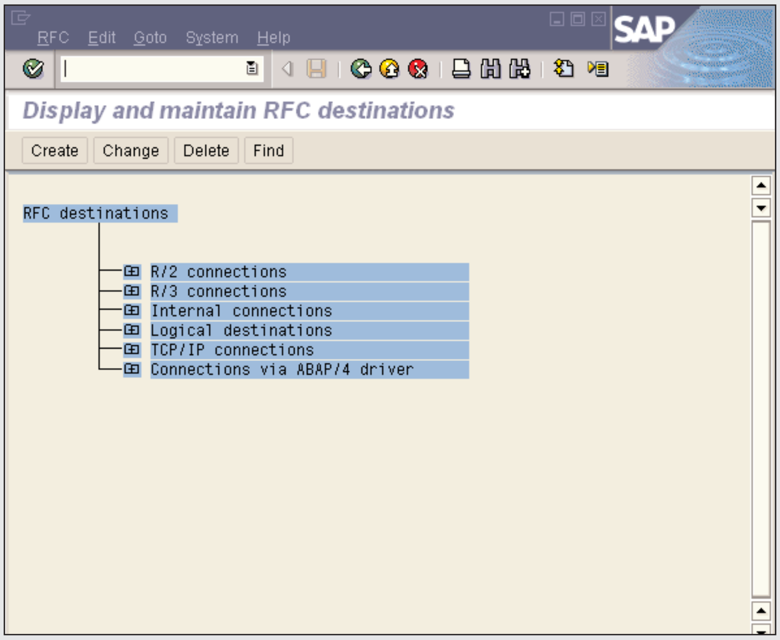
RFC Server Scenarios

Three main scenarios can be distinguished for RFC servers:

- The server provides functionality (e.g., tax calculation) that is accessed synchronously from SAP. The ABAP client usually passes

[1] Remote Function Call (RFC) is the SAP protocol used for communication between SAP, other SAP, and non-SAP systems or components.

Figure 1 *Transaction SM59*


RFC Edit Goto System Help
Display and maintain RFC destinations
Create Change Delete Find

```
RFC destinations
    ├─⊞ R/2 connections
    ├─⊞ R/3 connections
    ├─⊞ Internal connections
    ├─⊞ Logical destinations
    ├─⊞ TCP/IP connections
    └─⊞ Connections via ABAP/4 driver
```

parameters to the server and expects parameters back. This type of server is typically deployed in a data center. When the server is not available, the ABAP client program has to deal with that situation as best it can.

- The server receives Intermediate Documents (IDocs) from SAP. This requires support for tRFC (transactional RFC) in order to provide guaranteed delivery for the IDocs. While this scenario can be implemented with native JCo means, you make your life much easier by using a relatively new add-on to JCo, the SAP Java IDoc Class Library.[2] This type of server is also most often deployed in a data center.

- The server runs on the PC of a SAPGUI user. This user is running an ABAP program that invokes the JCo server, which is required to run on the same machine as SAPGUI. This makes sense if either the server has a GUI and needs to interact with the user or the server needs access to the file system of the SAPGUI user's machine.

All of these scenarios can be realized with JCo. In this article, the focus will be on the first scenario.

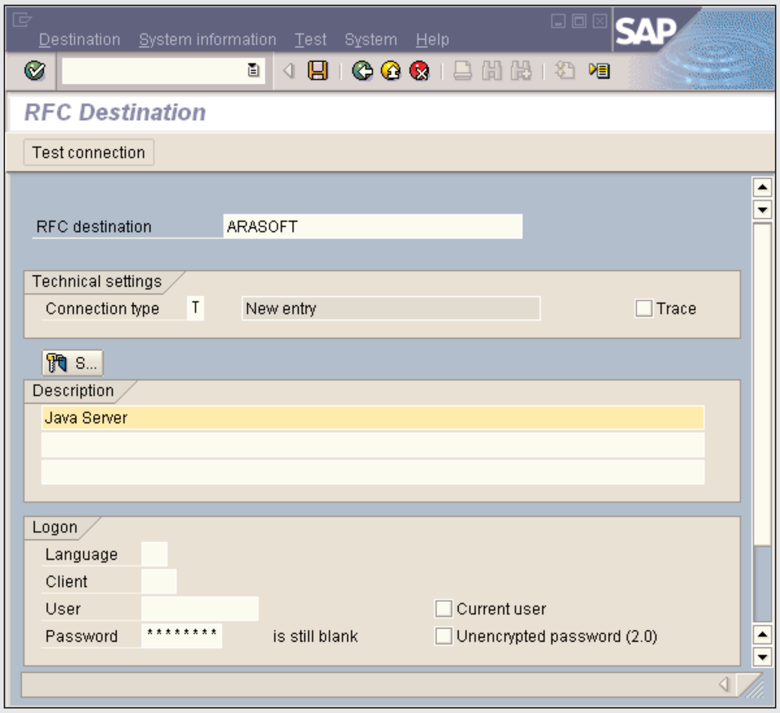
Defining a Destination in SAP

In order for an ABAP program to be able to make a remote call to a server, an RFC destination for this server must be defined in the SAP system. This is accomplished via transaction code SM59 (see **Figure 1**). To create a new destination, click on the *Create* button in Figure 1.

On the next screen (**Figure 2**), enter a name for the destination, specify a connection type of "T" (for TCP/IP), and enter a description. Then press the *Enter* key. Since there are many different types of RFC destinations, the initial screen cannot contain all the fields for all types. Pressing *Enter* allows SAP to interpret the *Connection type* field, and change the screen accordingly. You can see the result in **Figure 3**.

[2] See Robert Chu's excellent articles on pages 49 and 77 in this issue.

Figure 2 *Creating a TCP/IP Destination*



RFC Destination

Test connection

RFC destination ARASOFT

Technical settings

Connection type T New entry ☐ Trace

S...

Description

Java Server

Logon

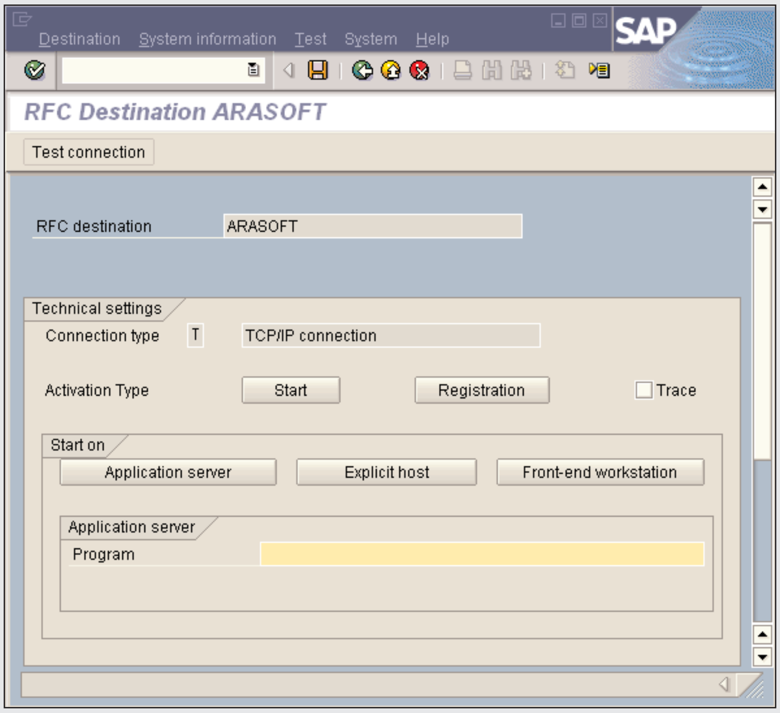
Language

Client

User ☐ Current user

Password ******** is still blank ☐ Unencrypted password (2.0)

Figure 3 *Activation Type*



RFC Destination ARASOFT

Test connection

RFC destination ARASOFT

Technical settings

Connection type T TCP/IP connection

Activation Type Start Registration ☐ Trace

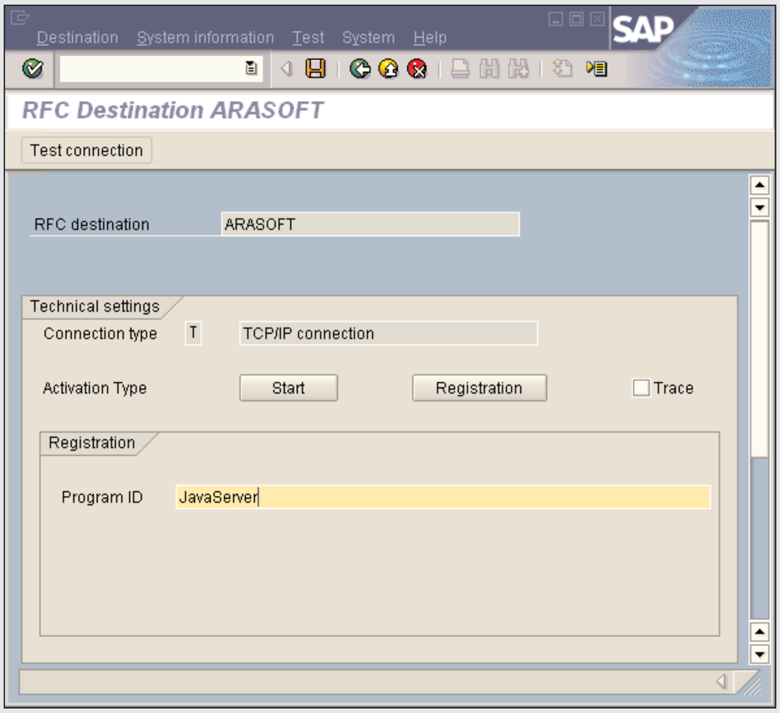
Start on

Application server Explicit host Front-end workstation

Application server

Program

Figure 4 *Specifying the Program ID*



RFC Destination ARASOFT

RFC destination: ARASOFT

Technical settings

Connection type: T TCP/IP connection

Activation Type

Registration

Program ID: JavaServer

A TCP/IP destination has one of two activation types, *Start* or *Registration*. JCo supports the latter, where the external server registers itself with the gateway process of an application server. Clicking on the *Registration* button in Figure 3 will make the screen reconfigure itself one more time, as shown in **Figure 4**.

Now enter a *Program ID* for your destination. This Program ID is case-sensitive and will have to be passed by the Java server program when it registers itself with the gateway.

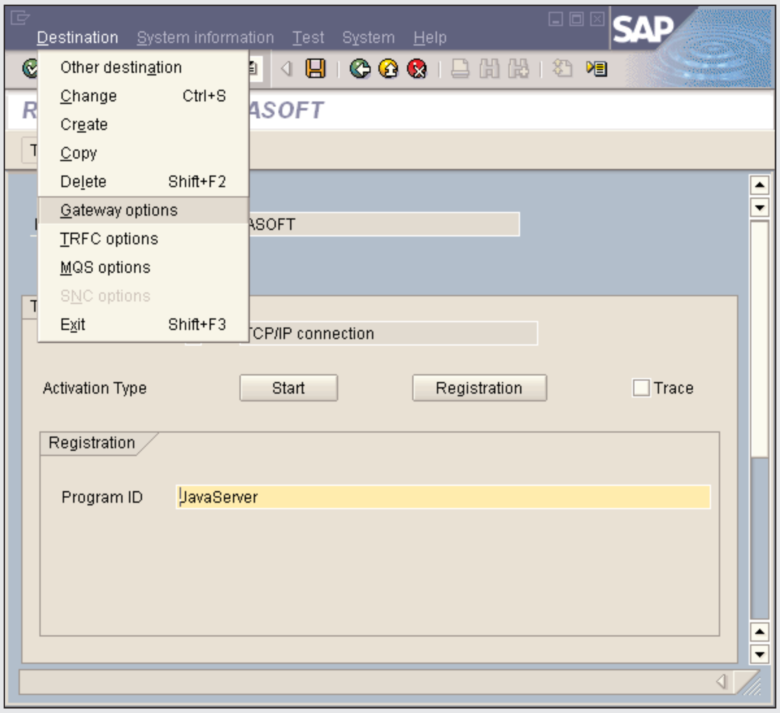
Speaking of which: unless you only have one application server for your SAP system, you need to specify at which gateway your Java server will register. Select *Destination* → *Gateway options* from the menu shown in **Figure 5**. In the resulting pop-up window (**Figure 6**), enter the host name of the application server you want to use when your Java server registers, and the gateway service string. This string always starts with "sapgw" and ends in the two-digit system number of your SAP system. Click on the *O.K.* button of the pop-up window and then on the *Save* icon of the main screen to save your new destination.

Make sure that you follow these steps exactly when you define a new destination. It is very easy to make a mistake, due to the dynamic screen changes. You might, for example, forget to click on the *Registration* button in Figure 3, and enter the *Program ID* in the field *Program*. As a result, your Java server would never be able to be invoked from ABAP, and you would see some interesting error messages.

Also, for the time being, resist the temptation to use the *Test connection* button. For this test to be successful, your Java server must be running and registered. We will see the result of the *Test connection* button later when our Java server is ready.

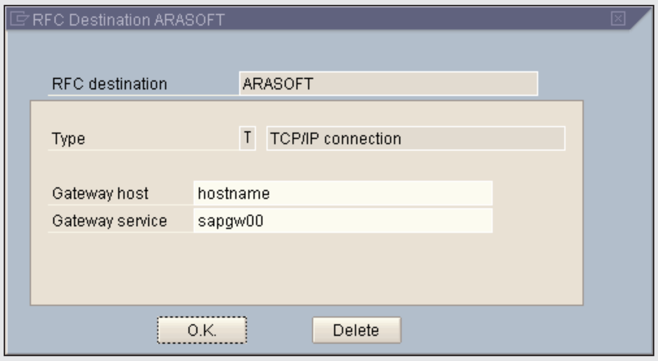
The destination that you just created can be used

Figure 5 *Setting the Gateway Options*



Program ID: JavaServer

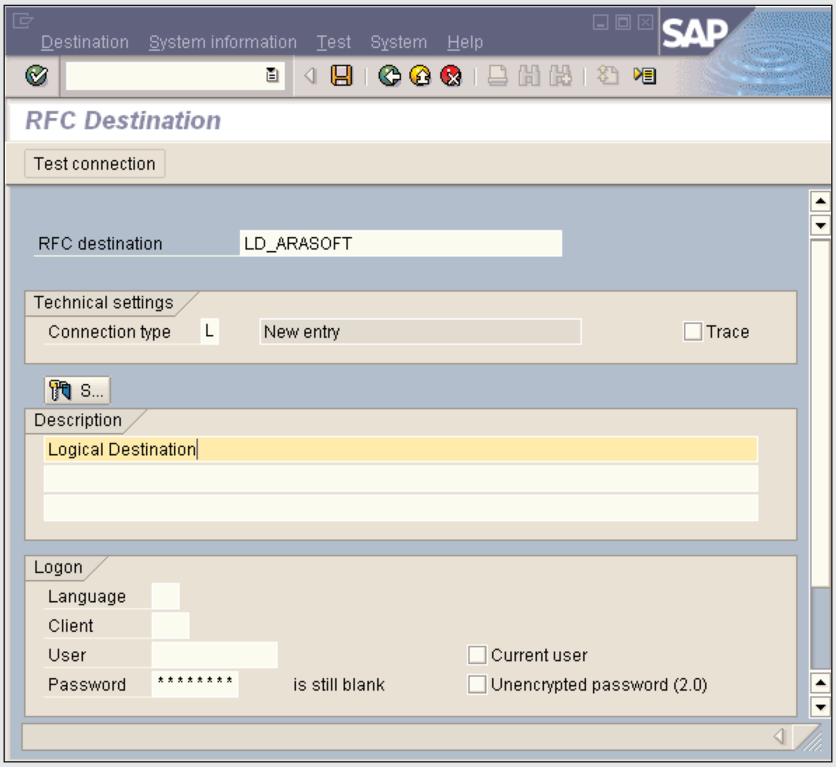
Figure 6 *Gateway Host and Service*



RFC destination	ARASOFT
Type	T TCP/IP connection
Gateway host	hostname
Gateway service	sapgw00

directly in your ABAP program, but this would result in the hard-coding of a destination name in your source code, something clearly to be discommended. Instead, the ABAP program should use a logical destination, which then references a physical destination (like the one you just created) or another logical destination. Obviously, at the end of a chain of logical destinations, there must always be a physical one, at least if you are interested in actually calling your server.

Figure 7 *Creating a Logical Destination*



RFC Destination

RFC destination LD_ARASOFT

Technical settings
Connection type L New entry ☐ Trace

Description
Logical Destination

Logon
Language
Client
User ☐ Current user
Password ******** is still blank ☐ Unencrypted password (2.0)

Figure 7 shows how a logical destination is created in SM59. The only difference from the definition of a TCP/IP destination is that here you select *Connection type* "L". Press the *Enter* key, and you get the screen in **Figure 8**. Now you simply enter the name of the physical destination to which a call to the logical one will be forwarded. Using only logical destinations in your ABAP code will prevent expensive maintenance if the name of a physical destination needs to be changed for whatever reason. Choose the names of your logical destinations wisely, though, because you will have to live with them forever unless you are willing to change your ABAP source code.

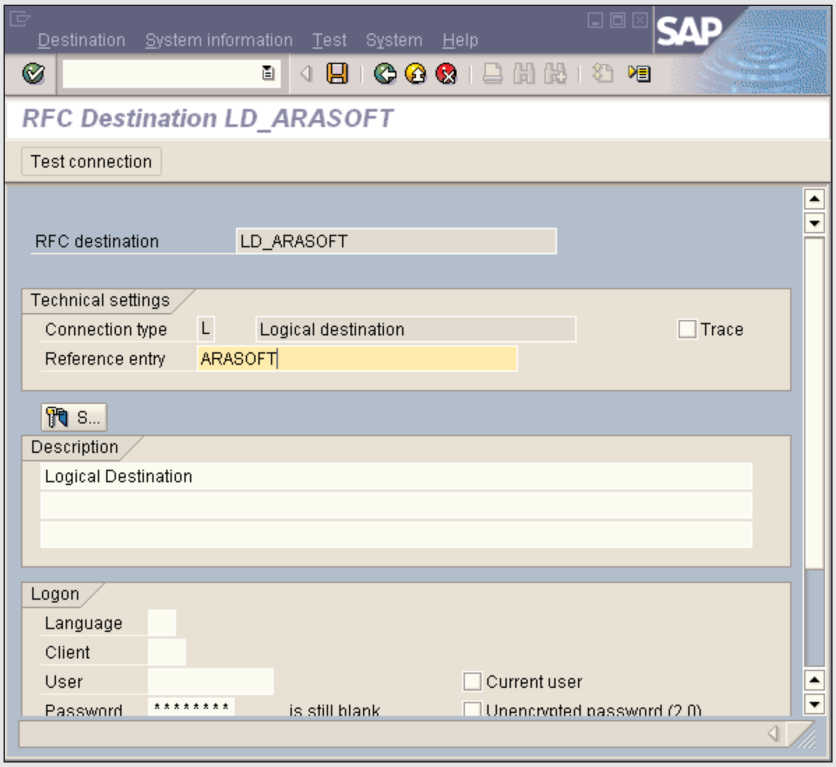
You may find it hard to believe, but that's all you need to do in the SAP system to prepare for RFC calls to your Java component.

Downloading and Installing the SAP Java Connector

SAP customers and partners can download JCo for free from **http://service.sap.com/connectors**. You should use the latest version, which at the time of this writing was 2.1.0. The following installation instructions apply to Windows. See the JCo documentation for the installation procedures for different platforms.

Extract the contents of the downloaded zip file into a folder. This folder now contains — amongst other things — a copy of the RFC library, librfc32.dll. Look up its version number and compare it to the version number of the librfc32.dll in WINNT\SYSTEM32. If the library supplied with JCo is newer than the one in WINNT\SYSTEM32, copy JCo's version there.

Figure 8 *Referencing the Physical Destination*



RFC Destination LD_ARASOFT

RFC destination: LD_ARASOFT

Technical settings
Connection type: L Logical destination
Trace
Reference entry: ARASOFT

Description
Logical Destination

Logon
Language
Client
User
Current user
Password: ******** is still blank
Unencrypted password (2.0)

Now go to your favorite Java IDE and make sure that file sapjco.jar (also in the main JCo installation folder) is in the classpath. EOI: end of installation.

JCo Basics for Server Programming

When your Java server is invoked, a *JCO.Function* object is passed. This object contains both the metadata and the actual parameter data of the invoked function. To access the parameters, the *JCO.Function* class offers three methods, *getExportParameterList()*, *getImportParameterList()*, and *getTableParameterList()*. All three methods return an object of type *JCO.ParameterList*, or *null*, if no parameters of the requested type exist.

Import parameters are those that the ABAP client sends to you, export parameters are the ones your Java server sends back. Import and export parameters are usually simple fields or structures consisting of simple fields.

Table parameters are intelligent arrays that represent ABAP internal tables. A table parameter has zero or more rows and each row has fields just like a structure parameter.

In other words, each parameter is a field or consists of fields. Time to look at the data type mapping between ABAP and Java.

The ABAP client will call a function in your Java server, blissfully unaware that some conversion magic takes place. You, on the other hand, have to

Figure 9 ***ABAP and Java Data Types***

ABAP	Description	Java	Type Constant
b	1-byte integer	int	JCO.TYPE_INT1
s	2-byte integer	int	JCO.TYPE_INT2
I	4-byte integer	int	JCO.TYPE_INT
C	Character	String	JCO.TYPE_CHAR
N	Numerical character	String	JCO.TYPE_NUM
P	Binary Coded Decimal	BigDecimal	JCO.TYPE_BCD
D	Date	Date	JCO.TYPE_DATE
T	Time	Date	JCO.TYPE_TIME
F	Float	double	JCO.TYPE_FLOAT
X	Raw data	byte[]	JCO.TYPE_BYTE
g	String (variable-length)	String	JCO.TYPE_STRING
y	Raw data (variable-length)	byte[]	JCO.TYPE_XSTRING

understand some of the magic. Take a look at **Figure 9**. It shows the mapping between the ABAP data types and the Java data types used by JCo. The most important detail about these conversions is that a fixed-length ABAP string loses its trailing blanks when converted to a Java *String* object.

Let us now look at how we access a simple field parameter in a *JCO.ParameterList* object. The first six methods in **Figure 10** correspond to the six different Java data types that JCo uses according to Figure 9. Usually, you will use the access method that fits the data type of the field, but JCo will try to convert to a different type if you use a different method. If the field is a date, for example, you can use *getString()* to get a string representation of that date. If the conversion fails, JCo will throw a *JCO.ConversionException*.

The final method in Figure 10, *getValue()*, can be used as a generic way to access a field and will return an object. Primitive Java data types, like *int*, will be wrapped into the appropriate class, like *Integer*.

The methods in Figure 10 let you access an individual simple field parameter by passing either the parameter name or a zero-based index. Using the former approach makes your source code a lot easier to read and less likely to be affected by changes in SAP.

Figure 11 contains additional access methods provided for your convenience. JCo will try to convert the field to the desired type and throw a *JCO.ConversionException* if the conversion fails.

To change the value of a field, you utilize the *setValue()* method. This method is overloaded for the data types listed in **Figure 12**. Again, JCo will try to convert and throw an exception if you pass an unconvertible value.

Figure 10 Basic Field Access Methods

public java.math.BigDecimal getBigDecimal(int index/String field_name)
public byte[] getByteArray(int index/String field_name)
public java.util.Date getDate(int index/String field_name)
public double getDouble(int index/String field_name)
public int getInt(int index/String field_name)
public java.lang.String getString(int index/String field_name)
public java.lang.Object getValue(int index/String field_name)

Figure 11 Convenience Field Access Methods

public java.math.BigInteger getBigInteger(int index/String field_name)
public java.io.InputStream getBinaryStream(int index/String field_name)
public char getChar(int index/String field_name)
public java.io.Reader getCharacterStream(int index/String field_name)
public short getShort(int index/String field_name)
public long getLong(int index/String field_name)
public java.util.Date getTime(int index/String field_name)

Figure 12 Data Types Allowed in setValue()

byte[]
char
int
long
double
short
java.lang.Object
java.lang.String

If the parameter is a structure, you use the *getStructure()* method of the *JCO.ParameterList* class, passing the name of the parameter. This method returns an object of type *JCO.Structure*. How do you access the fields in the structure? Easy: the methods of Figures 10 and 11 as well as the *setValue()* method are available for *JCO.Structure* objects, too.

One more parameter type to go: table parameters. Method *getTable()* on *JCO.ParameterList* gets passed the name of the table parameter and returns an object of type *JCO.Table*. This class is based on the notion of a current row, the fields of which can be accessed in the same fashion as in a *JCO.Structure* object. The

Figure 13 ***JCO.Table Navigation Methods***

Name	Description
void setRow(int pos)	Sets the current row index to the desired value.
int getNumRows()	Returns the number of rows in the table.
int getRow()	Returns the current row index.
void firstRow()	Equivalent to setRow(0).
void lastRow()	Equivalent to setRow(getNumRows() - 1).
boolean nextRow()	Equivalent to setRow(getRow() + 1), returns false if the current row is the last row.
boolean previousRow()	Equivalent to setRow(getRow() - 1), returns false if the current row is the first row.
boolean isEmpty()	Checks whether getNumRows() returns zero.
boolean isFirstRow()	Checks whether the current row is the first row.
boolean isLastRow()	Checks whether the current row is the last row.

methods shown in **Figure 13** are used to navigate a table.

Of course we also need to be able to add and remove rows in the table. The methods shown in **Figure 14** take care of that.

A Demo Server Class

Now that you know how to access and change the parameters of a *JCO.Function* object, you are ready to write your first server class. A JCo server class is derived from class *JCO.Server*, which takes care of all the hard work and only leaves the application logic to you. **Figure 15** contains a sample server class. Note that you have to provide a constructor that takes certain parameters and calls the appropriate constructor of its superclass. The constructor used in Figure 15 is the one that I always use, but there are some alternative formats described in the JCo documentation. The meaning of the constructor parameters will be discussed later. The only method your server class has to override is *handleRequest()*. JCo passes a *JCO.Function* object to this method. Using the *getName()* method of *JCO.Function*, you can distinguish which particular function was called from ABAP.

The sample server supports three functions: BAPI_COMPANYCODE_GETLIST (BAPI CompanyCode.GetList), BAPI_COMPANYCODE_GETDETAIL (BAPI CompanyCode.GetDetail), and DDIF_FIELDINFO_GET.

When the client calls BAPI_COMPANYCODE_GETLIST, we first delete all rows from the COMPANYCODE_LIST table parameter (in order to get rid of any rows the client has sent us), and then add two new rows. The COMP_CODE and COMP_NAME fields are filled for each row. Finally, we set the TYPE and

Figure 14 *JCO.Table Manipulation Methods*

Name	Description
void appendRow()	Adds a row at the end of the table. The current row index points to the new row.
void appendRows(int num_rows)	Adds multiple rows at the end of the table. The current row index points to the first of the new rows.
public void clear()	Deletes all rows.
void deleteAllRows()	Same as clear().
void deleteRow()	Deletes the current row.
void deleteRow(int pos)	Deletes the specified row.
void insertRow(int pos)	Inserts a row at the specified index.

Figure 15 *Class DemoServer*

```
import com.sap.mw.jco.*;

public class DemoServer extends JCO.Server {

  public DemoServer(java.util.Properties properties,
                    IRepository repository) {
    super(properties, repository);
  }

  protected void handleRequest(JCO.Function function) {
    if (function.getName().equals("BAPI_COMPANYCODE_GETLIST")) {
      JCO.Table codes =
        function.getTableParameterList().getTable("COMPANYCODE_LIST");
      codes.deleteAllRows();
      codes.appendRows(2);
      codes.setValue("0001", "COMP_CODE");
      codes.setValue("ARAsoft Germany", "COMP_NAME");
      codes.nextRow();
      codes.setValue("0002", "COMP_CODE");
      codes.setValue("ARAsoft Venezuela", "COMP_NAME");
      JCO.Structure returnCode =
        function.getExportParameterList().getStructure("RETURN");
      returnCode.setValue("S", "TYPE");
      returnCode.setValue("All is well.", "MESSAGE");
    }
    if (function.getName().equals("BAPI_COMPANYCODE_GETDETAIL")) {
      JCO.Structure returnCode =
        function.getExportParameterList().getStructure("RETURN");
```

(continued on next page)

Figure 15 *(continued)*

```
        String code =
          function.getImportParameterList().getString("COMPANYCODEID");
        if (code.equals("0001") || code.equals("0002")) {
          JCO.Structure detail =
            function.getExportParameterList()
                    .getStructure("COMPANYCODE_DETAIL");
          if (code.equals("0001")) {
            detail.setValue("Wiesloch", "CITY");
          } else {
            detail.setValue("Caracas", "CITY");
          }
          returnCode.setValue("S", "TYPE");
          returnCode.setValue("All is well.", "MESSAGE");
        }
        else {
          returnCode.setValue("E", "TYPE");
          returnCode.setValue("FN020", "CODE");
          returnCode.setValue
            ("Company code '" + code + "' not defined.", "MESSAGE");
          returnCode.setValue(code, "MESSAGE_V1");
        }
      }
      else if (function.getName().equals("DDIF_FIELDINFO_GET")) {
        throw new JCO.AbapException
          ("NOT_FOUND", "I know nothing (Manuel from FT)");
      }
      else {
        throw new JCO.AbapException
          ("FUNCTION_NOT_SUPPORTED",
           "Requested function not supported.");
      }
    }
  }
```

MESSAGE fields of the RETURN structure export parameter to indicate that the call was successful.

For BAPI_COMPANYCODE_GETDETAIL, we first look at the simple field import parameter COMPANYCODEID. If it contains one of the values returned by BAPI_COMPANYCODE_GETLIST ("0001" or "0002"), we set the CITY field of the COMPANYCODE_DETAIL structure export parameter accordingly. And we also provide a positive return message.

Otherwise, if the passed company code is different and hence unknown to us, we indicate an error by setting the TYPE, CODE, MESSAGE, and MESSAGE_V1 fields of the RETURN parameter.

The implementation of DDIF_FIELDINFO_GET in our server does not do anything particularly useful, but it is an example of how to throw an exception that is defined for the function in SAP. The constructor of the *JCO.AbapException* object is passed two parameters. The first one contains the exception name as defined in SAP, the second one a description text that we can set freely.

If the client tries to invoke any other function than the ones supported and just discussed, we throw an exception (FUNCTION_NOT_SUPPORTED) to indicate this.

Your own server will probably do something more interesting and important than the sample, but the way you will deal with the function parameters and throw exceptions will be the same.

Managing a Server

The server class encapsulates all the application logic, but in order to start and monitor the server you need some more code, which is typically kept in a separate class, hereinafter called the "server manager class". This class creates and starts as many instances of the server class as you would like. These instances are separate threads so that multiple requests from different ABAP client programs can be processed concurrently. If there are more concurrent requests than threads, some of the ABAP clients would have to wait and after 10 seconds the request would be terminated with an exception in SAP. So it is very important to provide enough threads to avoid this situation. You can monitor the activity of your server threads by using a *JCO.ServerStateChangedListener*, which will be discussed a little later. JCo makes it very easy to start multiple threads. All you have to do is make sure that your server class is thread-safe. The sample class in Figure 15 fulfills this requirement.

If you look at the constructor of our server class, you see that its second parameter must be of type *IRepository*. This repository provides the metadata of all the functions your server class wants to support. There are three main alternatives for providing this repository:

- Create a *JCO.Repositor*y object that uses a client (yes, our server actually is a client as well) connection to SAP to dynamically retrieve the required metadata from SAP. This is the recommended approach because you always use the correct, up-to-date metadata. All you have to ensure is that the interfaces of all functions that your server class supports are defined in the SAP Function Builder (transaction code SE37).

- Write your own repository class that extends *JCO.BasicRepository* and hard-code the metadata. An example of this approach can be found in the Example5.java sample program provided with JCo.

  There are two disadvantages of this approach:

  - Hard-coding the metadata correctly is far from easy and the documentation JCo offers on this topic is suboptimal.
  - Even more importantly, if a function interface is ever changed in SAP, you may have to change the source code of your server manager class.

- Create a *JCO.Repository* object with a client connection to SAP as in the first approach. Call method *getFunctionTemplate()* once for each function your server class needs to support in order to guarantee that all required metadata is now in the repository cache. Now save the repository to disk by using the *save()* method. In every subsequent start of your server manager class, load the repository from disk by calling the *load()* method.

  While this approach avoids hard-coding the metadata it still suffers from the problem that metadata may change in SAP, thus invalidating the metadata stored in the disk file.

If you want to learn more about repositories in JCo, please refer to my article "Repositories in the SAP Java Connector (JCo)" in the March/April 2003 issue of this publication.

Figure 16 *Creating the Repository*

```
static final int MAX_SERVERS = 3;
static final String POOL_NAME = "ARAsoft";

IRepository repository;
OrderedProperties logonProperties;

  private void createRepository() {
    try {
      logonProperties = OrderedProperties.load("/logon.properties");
      JCO.addClientPool(POOL_NAME,
                        MAX_SERVERS,
                        logonProperties);
      repository = JCO.createRepository("Repository", POOL_NAME);
      repository.getFunctionTemplate("BAPI_COMPANYCODE_GETLIST");
      repository.getFunctionTemplate("BAPI_COMPANYCODE_GETDETAIL");
      repository.getFunctionTemplate("DDIF_FIELDINFO_GET");
    }
    catch (Exception ex) {}
  }
```

The sample code in **Figure 16** employs the first, recommended approach. First, we need to establish a client connection to SAP, using a connection pool with as many connections as there will be instances of the server class (defined in constant MAX_SERVERS). In order to avoid hard-coded connection parameters, the required information is retrieved from a disk file like the one shown in **Figure 17**. The properties starting with “jco.client” are used for setting up the client connection, the others will be discussed in a moment. To actually load the properties into a *Properties* object, Figure 16 uses a utility class called *OrderedProperties* (the appendix to this article contains the complete source code). Of course you can use any other way of creating a *Properties* object in your own solutions.

Now the *addClientPool()* method is invoked to create the client connection pool. The first parameter is the name we want to give the pool, the second one defines the maximum number of connections in the pool. Using the constant MAX_SERVERS guarantees that our server class will never have to wait for metadata access in SAP. The final parameter is the *Properties* object that we created in the previous statement.

The repository can now be instantiated by invoking the *createRepository()* method. The first parameter is an arbitrary name, the second one is the name of the connection pool we just created.

The next three statements, invoking *getFunctionTemplate()* once for each function we want to support in our server class, are not strictly necessary. There are two advantages of doing this, nevertheless. Firstly, with all required metadata loaded already, the ABAP client program will not experience a longer response time when a function in our server class is called for the first time. Secondly, we will not need the client connection to SAP anymore, which will avoid problems when the SAP system becomes unavailable during the lifetime of our server. See the aforementioned article in the March/April 2003 issue for more details on this.

Now we can start our server threads by invoking the *startServers()* method in **Figure 18**. We create an array of server objects with MAX_SERVER entries. Then for each entry, an instance of *DemoServer* is created.

The first constructor parameter is a *Properties*

Figure 17 *A Sample Connection Properties File*

```
jco.client.client=001
jco.client.user=userid
jco.client.passwd=***
jco.client.lang=EN
jco.client.ashost=hostname
jco.client.sysnr=00

jco.server.gwhost=hostname
jco.server.gwserv=sapgw00
jco.server.progid=JavaServer
jco.server.max_startup_delay=600
```

Figure 18 *Starting and Stopping the Server Threads*

```
DemoServer servers[] = new DemoServer[MAX_SERVERS];

  private void startServers() {
    for (int i = 0; i < MAX_SERVERS; i++) {
      servers[i] = new DemoServer(logonProperties,
                                  repository);
      try {
        servers[i].start();
      }
      catch (Exception ex) {
        System.out.println("Could not start server.\n" + ex);
      }
    }
  }
  private void stopServers() {
    for (int i = 0; i < MAX_SERVERS; i++) {
      servers[i].stop();
    }
  }
```

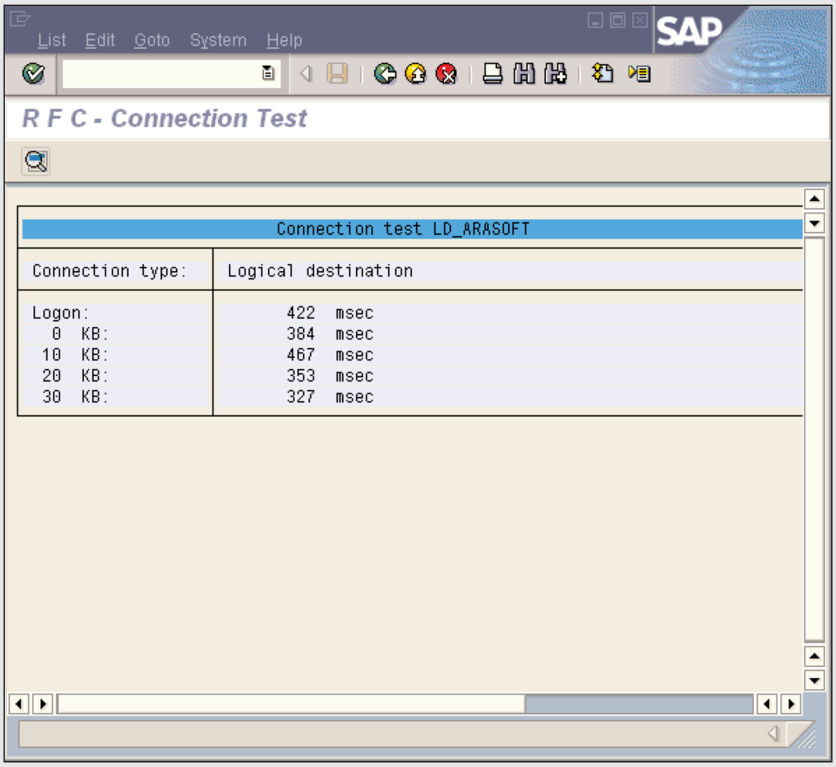
object. The `jco.server.gwhost`, `jco.server.gwserv`, and `jco.server.progid` properties in Figure 17 are required for the correct registration of our server. The values for these properties must be identical to the ones used when we created the physical destination.

The meaning of the last property in Figure 17, `jco.server.max_startup_delay`, will be discussed later.

The second constructor parameter is the repository object, created in Figure 16.

After instantiating the *DemoServer* threads, their *start()* method is called. Once that has been

Figure 19 Testing a Destination in SM59



```
Connection test LD_ARASOFT

Connection type:    Logical destination

Logon:                  422  msec
   0  KB:               384  msec
  10  KB:               467  msec
  20  KB:               353  msec
  30  KB:               327  msec
```

accomplished the server is ready for function calls from an ABAP client.

You can now test the destination from SM59. **Figure 19** shows the result of such a test. Note that I used the logical destination LD_ARASOFT. It refers to physical destination ARASOFT, which has been defined with Program ID "JavaServer".[3]

Error Handling and Server Monitoring

What if something goes wrong in your server?

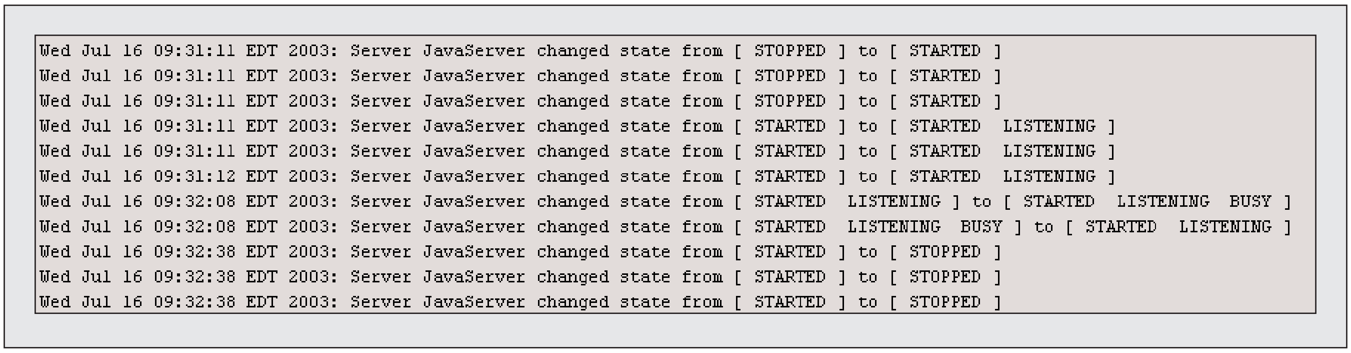
[3] The times shown in Figure 19 are comparatively large. This is due to the fact that the Java server was running on a PC in Venezuela, while the SAP system was in Philadelphia. In a local network with sufficient bandwidth you can expect much smaller numbers.

JCo provides the capability to register listeners that are invoked when a Java exception or a Java error occurs. The class that wants to be the listener (usually the server manager class) must implement the *JCO.ServerExceptionListener* and *JCO.ServerErrorListener* interfaces and register itself as the listener, using the *addServerExceptionListener()* and *addServerErrorListener()* methods, as shown in **Figure 20**. The listener methods invoked by JCo are *serverExceptionOccurred()* and *serverErrorOccurred()* respectively. The code for these methods in Figure 20 simply prints out information about the problem that occurred. Obviously, you could write much more elaborate implementations, notifying a central administration component or even alerting an operator by sending an email.

Figure 20 **Error Handling and Server Monitoring**

```
public class DemoServerManager
                 implements JCO.ServerExceptionListener,
                            JCO.ServerErrorListener,
                            JCO.ServerStateChangedListener {
  public DemoServerManager() {
    createRepository();
    JCO.addServerExceptionListener(this);
    JCO.addServerErrorListener(this);
    JCO.addServerStateChangedListener(this);
    startServers();
  }
  public void serverExceptionOccurred
             (JCO.Server server, Exception ex) {
    System.out.println("Exception in Server " + server.getProgID() +
                       ":\n" + ex);
    ex.printStackTrace();
  }
  public void serverErrorOccurred
             (JCO.Server server, Error error) {
    System.out.println("Error in Server " + server.getProgID() +
                       ":\n" + error);
    error.printStackTrace();
  }
  public void serverStateChangeOccurred
             (JCO.Server server, int old_state, int new_state) {
    System.out.print(new java.util.Date() + ": ");
    System.out.print("Server " + server.getProgID() +
                     " changed state from [");
    if ((old_state & JCO.STATE_STOPPED    ) != 0)
      System.out.print(" STOPPED ");
    if ((old_state & JCO.STATE_STARTED    ) != 0)
      System.out.print(" STARTED ");
    if ((old_state & JCO.STATE_LISTENING  ) != 0)
      System.out.print(" LISTENING ");
    if ((old_state & JCO.STATE_TRANSACTION) != 0)
      System.out.print(" TRANSACTION ");
    if ((old_state & JCO.STATE_BUSY       ) != 0)
      System.out.print(" BUSY ");
    System.out.print("] to [");
    if ((new_state & JCO.STATE_STOPPED    ) != 0)
      System.out.print(" STOPPED ");
    if ((new_state & JCO.STATE_STARTED    ) != 0)
      System.out.print(" STARTED ");
    if ((new_state & JCO.STATE_LISTENING  ) != 0)
      System.out.print(" LISTENING ");
    if ((new_state & JCO.STATE_TRANSACTION) != 0)
      System.out.print(" TRANSACTION ");
    if ((new_state & JCO.STATE_BUSY       ) != 0)
      System.out.print(" BUSY ");
    System.out.println("]");
  }
}
```

Figure 21 *Output from the JCO.ServerStateChangedListener*



```
Wed Jul 16 09:31:11 EDT 2003: Server JavaServer changed state from [ STOPPED ] to [ STARTED ]
Wed Jul 16 09:31:11 EDT 2003: Server JavaServer changed state from [ STOPPED ] to [ STARTED ]
Wed Jul 16 09:31:11 EDT 2003: Server JavaServer changed state from [ STOPPED ] to [ STARTED ]
Wed Jul 16 09:31:11 EDT 2003: Server JavaServer changed state from [ STARTED ] to [ STARTED  LISTENING ]
Wed Jul 16 09:31:11 EDT 2003: Server JavaServer changed state from [ STARTED ] to [ STARTED  LISTENING ]
Wed Jul 16 09:31:12 EDT 2003: Server JavaServer changed state from [ STARTED ] to [ STARTED  LISTENING ]
Wed Jul 16 09:32:08 EDT 2003: Server JavaServer changed state from [ STARTED  LISTENING ] to [ STARTED  LISTENING  BUSY ]
Wed Jul 16 09:32:08 EDT 2003: Server JavaServer changed state from [ STARTED  LISTENING  BUSY ] to [ STARTED  LISTENING ]
Wed Jul 16 09:32:38 EDT 2003: Server JavaServer changed state from [ STARTED ] to [ STOPPED ]
Wed Jul 16 09:32:38 EDT 2003: Server JavaServer changed state from [ STARTED ] to [ STOPPED ]
Wed Jul 16 09:32:38 EDT 2003: Server JavaServer changed state from [ STARTED ] to [ STOPPED ]
```

To provide monitoring of your server you can employ a *JCO.ServerStateChangedListener*, which is invoked whenever the state of one of your server threads changes. The listener is registered by a call to *addServerStateChangedListener()* and the method you must implement is called *serverStateChangeOccurred()*. This method is passed three arguments: the server instance for which the change occurred, the old state of the server, and the new state of the server. The two states are encoded in an integer. The meaning of the individual bits can be found in the JCo documentation.

The code for the *serverStateChangeOccurred()* in Figure 20 prints out the old and new states. **Figure 21** contains sample output. Again, your own implementation could be much more elaborate. You could, for instance, keep track of the percentage of your server threads in use at any time and start additional threads to accommodate more traffic from the SAP system.

Handling Connection Problems

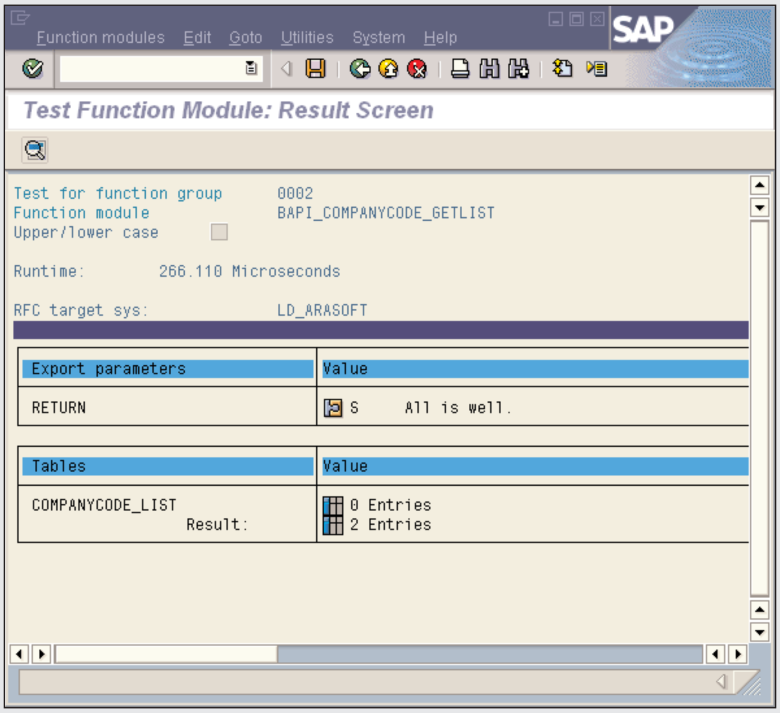
Things break. Networks fail. Our server, on the other hand, should be available whenever the ABAP clients need it. If we weren't using JCo for our server, this section of the article could be quite long. But fortunately JCo has taken appropriate precautions in this area. When one of our server threads fails, because of a network problem, an exception in our code, or for any other reason, JCo waits one second and then attempts to register this server thread again. If that attempt fails (e.g., due to network problems or the SAP system being down for maintenance) the interval is doubled to two seconds, and so on, until the limit defined in property `jco.server.max_startup_delay` is reached (Figure 17 sets this value to 600 seconds, or 10 minutes, for our server). The default for this property is defined by property `jco.middleware.max_startup_delay` and is 3,600 seconds (one hour). To change the default, use method *setMiddlewareProperty()* of class *JCO*. Use the *getMiddlewareProperty()* method to get a string containing the current value for this property.

In addition to this automatic restart behavior, you can extend your server manager class to provide a GUI so that an operator can supervise the status of the server threads and manually restart them immediately after a network or other problem has been resolved.

Invoking the Server from ABAP

We have now built a full-blown server and should take a look at the ABAP client side. The easiest way to invoke our server is from the SAP Function Builder (SE37). **Figure 22** shows the result of calling BAPI_COMPANYCODE_GETLIST. Note that I used the logical destination LD_ARASOFT as the RFC target system. To view the contents of the COMPANYCODE_LIST table parameter, click on the table icon in the row that says "Result". **Figure 23** proves that we successfully populated the table.

Figure 22 Calling BAPI_COMPANYCODE_GETLIST from SE37



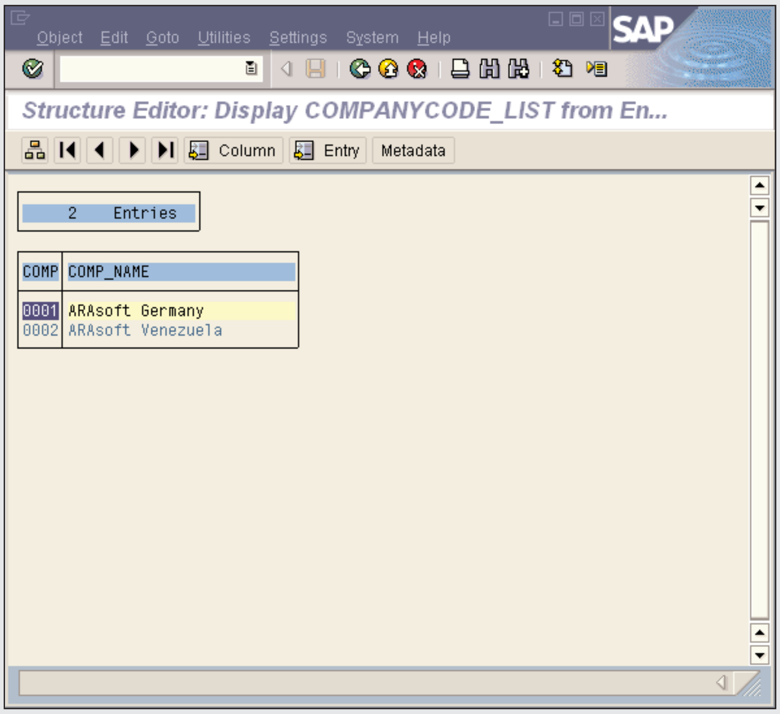
Test Function Module: Result Screen

```
Test for function group      0002
Function module              BAPI_COMPANYCODE_GETLIST
Upper/lower case

Runtime:        266.110 Microseconds

RFC target sys:              LD_ARASOFT
```

Export parameters	Value
RETURN	S All is well.

Tables	Value
COMPANYCODE_LIST Result:	0 Entries 2 Entries

Figure 23 Displaying Parameter COMPANYCODE_LIST

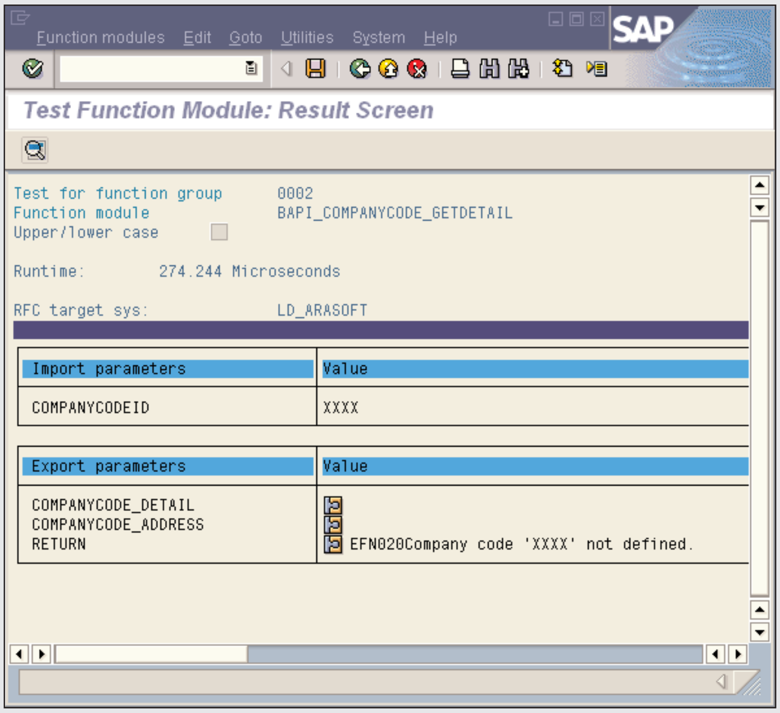


Structure Editor: Display COMPANYCODE_LIST from En...

2 Entries

COMP	COMP_NAME
0001	ARAsoft Germany
0002	ARAsoft Venezuela

Figure 24 *Calling BAPI_COMPANYCODE_GETDETAIL from SE37*

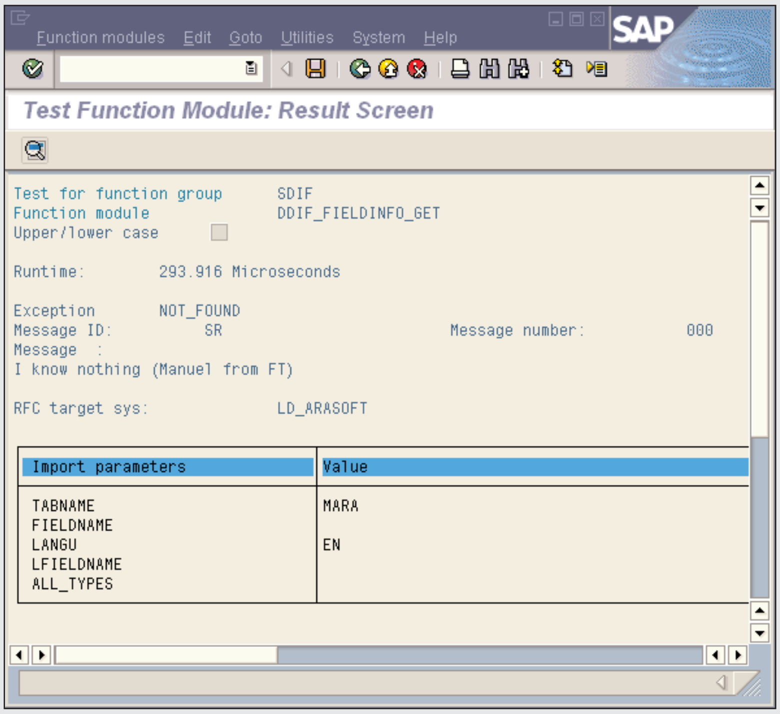

Function modules Edit Goto Utilities System Help
Test Function Module: Result Screen

```
Test for function group      0002
Function module              BAPI_COMPANYCODE_GETDETAIL
Upper/lower case

Runtime:         274.244 Microseconds

RFC target sys:              LD_ARASOFT
```

Import parameters	Value
COMPANYCODEID	XXXX

Export parameters	Value
COMPANYCODE_DETAIL	
COMPANYCODE_ADDRESS	
RETURN	EFN020Company code 'XXXX' not defined.

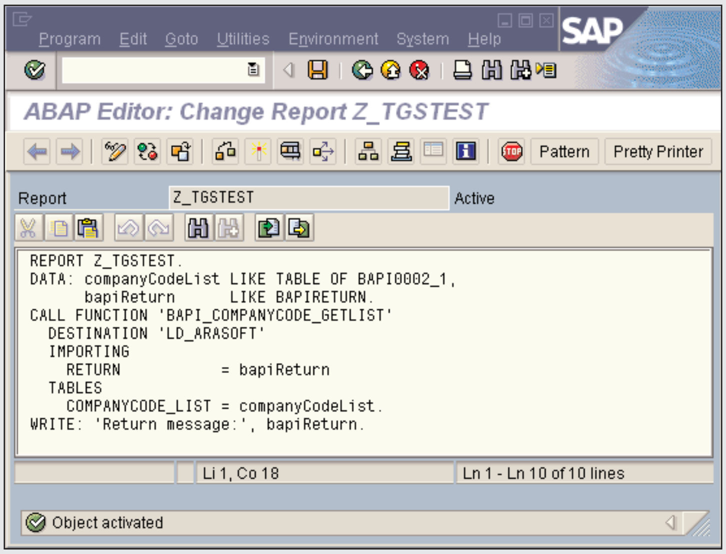
Figure 25 *Calling DDIF_FIELDINFO_GET from SE37*


Function modules Edit Goto Utilities System Help
Test Function Module: Result Screen

```
Test for function group      SDIF
Function module              DDIF_FIELDINFO_GET
Upper/lower case

Runtime:         293.916 Microseconds

Exception        NOT_FOUND
Message ID:           SR                           Message number:          000
Message   :
I know nothing (Manuel from FT)

RFC target sys:              LD_ARASOFT
```

Import parameters	Value
TABNAME	MARA
FIELDNAME	
LANGU	EN
LFIELDNAME	
ALL_TYPES	

Figure 26 *ABAP Code Calling BAPI_COMPANYCODE_GETLIST*

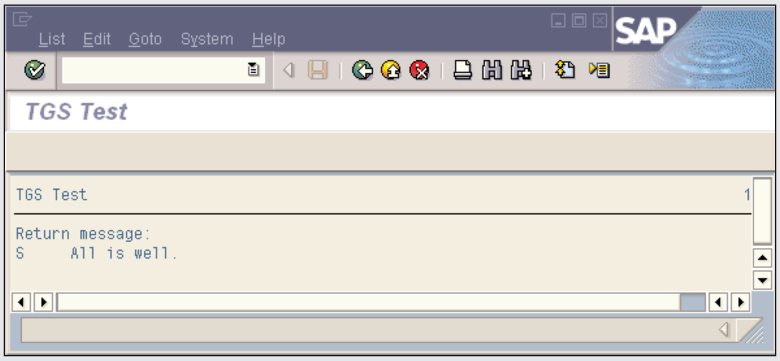

Program Edit Goto Utilities Environment System Help
ABAP Editor: Change Report Z_TGSTEST
Pattern
Pretty Printer
Report
Z_TGSTEST
Active

```
REPORT Z_TGSTEST.
DATA: companyCodeList LIKE TABLE OF BAPI0002_1,
      bapiReturn      LIKE BAPIRETURN.
CALL FUNCTION 'BAPI_COMPANYCODE_GETLIST'
  DESTINATION 'LD_ARASOFT'
  IMPORTING
    RETURN          = bapiReturn
  TABLES
    COMPANYCODE_LIST = companyCodeList.
WRITE: 'Return message:', bapiReturn.
```

Li 1, Co 18   Ln 1 - Ln 10 of 10 lines

Object activated

Figure 27 *Output from the ABAP Report*



```
TGS Test                                                          1

Return message:
S     All is well.
```

In **Figure 24**, I deliberately passed an invalid company code to BAPI_COMPANYCODE_GETDETAIL in order to test the error handling of our server.

Figure 25 contains the result of invoking DDIF_FIELDINFO_GET. As expected, an exception was thrown by our server. Note that the text that was passed as the second parameter into the exception's constructor is actually transferred back to SAP.

For those of you not that familiar with the ABAP syntax, **Figure 26** shows a function call to BAPI_COMPANYCODE_GETLIST in logical destination LD_ARASOFT. **Figure 27** contains the result of running the report.

Conclusion

RFC servers are a powerful tool for your integration projects. JCo has made it very simple to write your own servers in Java, taking care of multi-threading and connection restart for you. Just add some application logic and you have a winner.

Thomas G. Schuessler is the founder of ARAsoft (www.arasoft.de), a company offering products, consulting, custom development, and training to a worldwide base of customers. The company specializes in integration between SAP and non-SAP components and applications. ARAsoft offers various products for BAPI-enabled programs on the Windows and Java platforms. These products facilitate the development of desktop and Internet applications that communicate with R/3. Thomas is the author of SAP's BIT525 "Developing BAPI-enabled Web Applications with Visual Basic" and BIT526 "Developing BAPI-enabled Web Applications with Java" classes, which he teaches in Germany and in English-speaking countries. Thomas is a regularly featured speaker at SAP TechEd and SAPPHIRE conferences. Prior to founding ARAsoft in 1993, he worked with SAP AG and SAP America for seven years. Thomas can be contacted at thomas.schuessler@sap.com or at tgs@arasoft.de.

Appendix: Class OrderedProperties

```
import java.util.*;
import java.io.*;

public class OrderedProperties extends java.util.Properties {

  ArrayList orderedKeys = new ArrayList();

  public OrderedProperties() {
    super();
  }
  public OrderedProperties(java.util.Properties defaults) {
    super(defaults);
  }

  public synchronized Iterator getKeysIterator() {
    return orderedKeys.iterator();
  }

  public static OrderedProperties load(String name) throws Exception {
    OrderedProperties props = null;
    java.io.InputStream is =
      OrderedProperties.class.getResourceAsStream(name);
    props = new OrderedProperties();
    if (is != null) {
      props.load(is);
      return props;
    }
    else {
      throw new IOException("Properties could not be loaded.");
    }
  }
```

```
    public synchronized Object put(Object key, Object value) {
      Object obj = super.put(key, value);
      orderedKeys.add(key);
      return obj;
    }

    public synchronized Object remove(Object key) {
      Object obj = super.remove(key);
      orderedKeys.remove(key);
      return obj;
    }
  }
```