WIS
Wellesley Information Services

Improve communication between your C/C++ applications and SAP systems with SAP NetWeaver RFC SDK

Part 2: RFC server programs

by Ulrich Schmidt, Senior Developer, SAP AG, and Guangwei Li, Senior Developer, SAP AG

SAP's new software development kit (SDK) for remote function call (RFC) communications, SAP NetWeaver RFC SDK, is the successor to SAP R/3's classic RFC SDK. You can use it in C/C++ based applications to communicate with SAP back-end systems.

This article is the second of a three-part series on RFC communications between an SAP system and an external program written in C or another low-level programming language that has a C interface.[1] In the first article, we discussed the basic data structures — metadata descriptions and data containers — in use throughout SAP NetWeaver RFC SDK. We also discussed RFC client programs (i.e., those cases where an external program issues an RFC into the SAP system). In this article, we talk about RFC server programs (i.e., those cases where the SAP system issues an RFC to an external program). This type of functionality enables you to access any kind of system or functionality from an ABAP application, even assembly-language routines or hardware device drivers, if you need them.

First, we explore the basic design of an RFC server program based on SAP NetWeaver RFC SDK. Then, we'll use most of the SAP NetWeaver RFC library features to build a generic server, which can receive and process any arbitrary function call from the back end. These features include:

- Automatically retrieving and caching structure information (metadata descriptions) for arbitrary function modules
- Receiving calls from multiple SAP systems in a single server program
- Protecting the server from unauthorized access if only certain SAP system users are allowed to execute functions on the server
- Triggering all kinds of exceptions, such as ABAP exceptions, ABAP messages, and system failures, from the server program

If you haven't already done so, download the source code examples mentioned in this article from the November/December 2007 downloads section at http://www.sappro.com/downloads.cfm. We recommend that you have the examples available as you read this article because only portions of the code appear in print. To compile and run the sample programs, you need to download SAP NetWeaver RFC SDK from the SAP Service Marketplace.[2] SAP Note 1025361 describes where to find the SDK for various operating system platforms. Make sure that you have at least patch level 1 of SAP NetWeaver RFC SDK, because a few of the functions mentioned in this article were not available at patch level 0. If you have trouble compiling the sample programs included in this article, please review the tips for compiling and linking in SAP Note 1056696.

[1] "Improve communication between your C/C++ applications and SAP systems with SAP NetWeaver RFC SDK: Part 1 — RFC client programs" (*SAP Professional Journal*, November/December 2007).

[2] Logon credentials are required to access the information in the SAP Service Marketplace at http://service.sap.com/swdc.

>> Prerequisites

We assume that you have a basic understanding of how RFC communications work from the ABAP point of view. That is, you should be familiar with the concepts of a remote-enabled function module and BAPI (to call ABAP functionality from external C/C++ programs) and with the `CALL FUNCTION ... DESTINATION` statement (to call external functionality from ABAP programs). If necessary, refer to the corresponding chapters of the online SAP Library. You should also take a detailed look at the API documentation that comes with SAP NetWeaver RFC SDK: the sapnwrfc.h header file and a programming guide in PDF form. You don't need previous knowledge of the classic RFC SDK, but we assume you've read the first article of this series.

A generic server

When writing an RFC server program, you use a number of the features that were explained in the first article: most notably, *metadata descriptions* and *data containers*. The basic concepts of a server program, however, differ substantially from those of a client program.

The main reason is that there are usually only a fixed number of function modules that the application needs to call, and you know all of them in advance when you write the client application. You hard code these function modules into the program. As the demoClient example in the previous article shows, however, you can still write client programs that react to external (e.g., user) input and you can determine dynamically at runtime what function modules to call.

On the other hand, a server program doesn't decide which function modules to execute at runtime; the SAP back-end system on the other side of the network connection makes this decision. So, even though there are still cases in which a server that can handle a small, predefined selection of function modules is sufficient, the need for a server program that reacts to incoming calls of an unexpected type is much greater than it is for a client program.

Let's start with the server example in SAP NetWeaver RFC SDK so that you can see how a server with a predefined set of function modules works behind the scenes. However, because you need to design a generic server quite differently, that discussion will make up the remainder of this article.

Designing a simple server

Before the server begins to listen for incoming requests via RfcListenAndDispatch(), you need to take a few preliminary steps. (Please open the file demo\stfcDeepTableServer.c from SAP NetWeaver RFC SDK.) To receive a certain function call, the RFC library needs two pieces of information that the developer must provide:*A metadata description for the function module.* When a request arrives, the RFC library uses it to create a data container for the input data that comes with the call.

- *A C function implementing the function module's "business logic."* When a request arrives, the RFC library calls the C function, passes it the new data container filled with the input data, and expects that the C function will fill the data container's output parameters.

>> Tip!

The RFC library needs a pointer to a C function to which it can jump. If a user wants to implement the business logic in a different language, he or she still needs to provide the C function and make the jump to the other language from that function.

Before the stfcDeepTableServer.c program enters the dispatch loop in line 167, it creates a metadata description for the function module STFC_DEEP_TABLE using RfcGetFunctionDesc(). For this, the program needs the user ID and password information for the back-end system (stored in repoCon[]). This may seem a bit strange at first because you wouldn't expect a server program to need logon credentials. But if you want to look up the metadata in the back-end system's data dictionary (DDIC), which is a very convenient feature, first you need to log on via a client connection. The back-end user for this step doesn't need many authorizations. The authorization objects that SAP Note 460089 lists for a repository user are sufficient.[3] As an alternative to using RfcGetFunctionDesc(), you can construct a hard coded metadata description by hand, but this task is tedious and error-prone and should only be used when it's absolutely necessary.

Next, the server program needs a C function of type RFC_SERVER_FUNCTION that implements the logic for TFC_DEEP_TABLE. The C implementation of this function module does something completely different from what the function's ABAP implementation does in the back end. It prints the import parameters it received and asks the user for the values of the export parameters. Upon receiving a certain value in the import parameters, it shuts itself down. In contrast, the ABAP version of this function module just copies the import parameter values into the export parameters. The stfcDeepTableServer program implements the function module logic in the function stfcDeepTableImplementation() at the beginning of the file. This function is a callback function because from a C/C++ perspective it is a function pointer being passed around as an argument. (Don't confuse this with the concept of RFC callback, which we'll discuss in the third article in

[3] See the discussion of this topic in the section "Log on to the back-end system" in the November/December 2007 article.

How the registration process works

It's easy to get unexpected results if you do the registration process incorrectly, so let's discuss how it works.

In an SAP system with several application servers, each one typically has its own RFC gateway. If, in the definition of the RFC destination in SM59 (Connection Establishment Using Destination), you leave the two fields *gateway host* and *gateway service* empty, then each application server will look to its own local gateway to find registrations for that particular program ID. If the application server can't find any registration information on its local gateway, the RFC will issue the error message "program not registered" and fail. In short, your external server program is visible only on the application server whose host name you specified as gateway host in the `RfcRegisterServer()` call.

>> Note

The load-balancing algorithm of the RFC gateway has been changed (and fixed) a number of times. If you want to make sure your system uses that load-balancing algorithm, which is considered the most efficient, you should upgrade your system to the most current kernel-patch level for the SAP system release in use at your site.

You can use this feature if you want to do load balancing and you expect every application server to generate approximately the same load. You start your external server program *n* times, once for each application server. Then, every application server has its own dedicated instance of the server program to which it can send requests. However, you typically want to start the external program once and have it accessible from every application server that's part of the SAP R/3 system. In that case, you pick one particular application server — perhaps the most powerful one, or the one that doesn't already have too many other programs registered — and enter its host name as the gateway host in SM59. (The gateway service usually needs to be sapgw*XX*, where *XX* is the system number.) Then, all the application servers know that the external program registers at that particular gateway. So, they send their requests for this destination to the specified gateway, which forwards it to the registered program. You only need to make sure that your program starts with the correct gateway host parameter.

You can still use load balancing. You just need to start the program several times or start several threads in it, each of which issues its own `RfcRegisterServer()` using the same gateway parameters. The gateway automatically balances the load over those multiple registrations.

>> Note

There is a profile parameter, gw/monitor, which prevents remote gateway monitoring. If this parameter is set to a value other than "2" in your system, then RSGWREGP only lists the registrations on the application server's local gateway. In this case, you need to execute that report once on each application server to get a complete list.

This load-balancing feature of the RFC gateway may be confusing at first. Often, after a scenario has gone live, a forgotten test or development server still registers its program. In that case, half the RFC calls (or IDocs) go to the production instance and process correctly, while the other half are sent to the forgotten test instance and either appear to be lost without a trace or generate strange error messages that no one can explain. In most cases, everything works as designed and the administrators just need to turn off the forgotten test server to prevent it from receiving any calls.

If you ever need to investigate such mysterious phenomena, the report RSGWREGP will come in handy. It connects to each RFC gateway in the SAP system and lists all currently registered RFC programs, together with the status and IP address from which the registration came.

Finally, a word about security: There are a number of gateway parameters that control who can register an external server at an RFC destination. Please see the gateway documentation in the online SAP Library. We recommend using at least the secinfo file. If you search the SAP Library with the keywords "gateway" and "secinfo," your search should turn up an entry called "Authorizations for Registering External Programs with the SAP Gateway." The secinfo file on the gateway allows you to define from which IP addresses a registration for a certain program ID (RFC destination) may come. The gateway then rejects any registration attempts for that program ID from other hosts. This limits the danger of an unwanted program registering at the destination and intercepting RFC calls, either accidentally (a forgotten test server) or intentionally (an intruder).

this series.) In the rest of this article, the terms *callback function* and *implementing function* refer to this special kind of function, depending on whether the current focus is on the RFC library's use of the function as a callback or on the function's implementation of the business logic.

Then, all of the individual pieces of the previous two paragraphs are put together in line 158 of stfcDeepTableServer.c, where the server program installs the metadata description and callback function pair using RfcInstallServerFunction(). The metadata description contains the function module name, so the RFC library can identify this pairing when a request for STFC_DEEP_TABLE arrives. Usually, the first parameter of RfcInstallServerFunction() is the system ID of the back-end system from which you want to receive calls. In our example, the demo program passes a NULL reference here instead of a system ID.

Next, the program needs the three parameters *gateway host*, *gateway service*, and *program ID* (stored in serverCon[]) to register at the back-end system's RFC gateway using RfcRegisterServer(). Registering makes the RFC server program known to the SAP system, so when the SAP system wants to send an RFC call to the RFC destination corresponding to that program ID, the SAP system knows where to send the call. See the sidebar below for more on how the registration process works.

The RFC server program is now ready to start listening for incoming requests via the RfcListenAndDispatch() function. When you call it, the following happens internally:

1. The RFC library waits for an incoming request from the back-end system. While the default is set to wait forever (called a "blocking wait"), you can also specify a timeout in seconds if your program (or the current thread) needs to perform other tasks on occasion. If no request arrives within the timeout period, RfcListenAndDispatch() returns with the code RFC_RETRY.

2. When a request arrives, RfcListenAndDispatch() first searches for a pairing of metadata description and C function that matches the function module name, which it can use to process the call. Here, it applies the following search order:

   – It checks whether, for the combination of system ID and function module name, a pair of metadata descriptions has been installed via the following code:

   ```
   RfcInstallServerFunction(
      cU("SysID"), metadataHandle,
      callbackFunction, &errorInfo)
   ```

   – If that doesn't turn up anything, it checks to see whether a pair of metadata description and callback function for that function module name was installed using this code:

   ```
   RfcInstallServerFunction(
      NULL, metadataHandle,
      callbackFunction, &errorInfo)
   ```

   Passing NULL as the system ID in RfcInstallServerFunction() is convenient if your server program connects to several different back-end systems and you want to provide a single implementing function for them all. A word of warning, however: Different back-end releases may have different descriptions for the same function module. For example, in a newer release, a function module may have an additional import parameter, or perhaps it has enlarged one of the parameters (e.g., CHAR30 to CHAR50).

   Let's say you fetch the metadata description from the older back end, register it globally via sysID=NULL, and then a call comes in from the newer back end. This call would fail because the RFC library couldn't fit the data received from the newer back end into the data container created from the old metadata description. If you're planning to use this feature, make sure you obtain your metadata descriptions from the back end using the latest release. A few extra fields or enlarged fields won't hurt when receiving calls from the older system; the extra memory in the data container will just remain empty.

   – If the RFC library still can't find a metadata/callback pair, it checks whether RfcInstallGenericServerFunction() has installed a global function handler. This mechanism is a bit more complicated than the one using RfcInstallServerFunction() (for more on this topic see "Setting up automatic metadata retrieval" on the following page).

   – If a global handler doesn't exist, the RFC library aborts the call and sends a SYSTEM_FAILURE to the back end with an error saying that the function module could not be found.

3. Let's assume that a matching metadata/callback pair was found for the function module. Then, RfcListenAndDispatch() creates a data container from the metadata description, fills it with the data from the request, and calls the callback function, passing the data container to it. At this point, the control goes back to you. You can do whatever you like in that callback function: Read the import parameters from the data container, set values for the export parameters, or some other action.

4. When the callback function returns, RfcListenAndDispatch() performs one of two actions, depending on the return code: Either it sends some type of error message back to the SAP system, or if the callback function returns RFC_OK, which it should most of the time, it reads the export parameter values from the data container and returns them to the back end. (The mechanism showing how the callback function triggers different error types is described in "Throwing exceptions" on page 6.)

5. After the response to the function call has been sent back to the SAP system, RfcListenAndDispatch() frees the data container used for the call and returns the control back to you. In most cases, the return code will be the code that RfcListenAndDispatch() got from the callback function (i.e., RFC_OK or an error code). The only exception to this is when a technical problem occurs while processing the callback function's output. For example, if the network connection to the back-end system breaks down while RfcListenAndDispatch() is trying to return the response, the return code will be RFC_COMMUNICATION_FAILURE.

Now, the control is back in the dispatch loop. The correct way to react to the different return codes of RfcListenAndDispatch() is closely intertwined with the way the callback function triggers the different error types. The usual reaction is to loop over to the beginning of the dispatch loop and wait for the next incoming call. However, if the error code tells you that the back end is down or if some global flag tells you to stop because the program is shutting down, then the natural reaction would be to break out of the loop and do the necessary cleanup. This completes the server program.

For information regarding some of the differences between the classic RFC SDK and SAP NetWeaver RFC SDK's concepts for writing an RFC server program, see the section "Basic functionality" in the sidebar that begins on page 9.

Setting up automatic metadata retrieval

The SAP NetWeaver RFC library needs the metadata description of a function module before it can process that function module. In the simple RFC server, the metadata description for the function module that the server is supposed to process is simply handed over to the RFC library before the server is started. But what do you do if you don't know in advance which function modules to expect?

In a simple RFC server, a C function dedicated to handling one particular function module is installed with the metadata object for that function module. In the generic server, the C function to handle all incoming requests is installed with another callback function, which the RFC library will call whenever it needs some function module's description. You need to implement this function, and in it you can do whatever you like, as long as it ends up with a usable metadata description for the function module at hand. For example, you could simply return a hard coded description for some function modules and open a client connection into the back-end system to use RfcGetFunctionDesc() for the rest.

Please take a look at demoServer.c now. The function for providing metadata is implemented in line 171 of repositoryLookup(), a function of type RFC_FUNC_DESC_CALLBACK. It gets the system ID from the attributes of the current call, opens a client connection into that system, and looks up the function module there using RfcGetFunctionDesc(). The result is put in the output parameter of repositoryLookup(). If anything goes wrong, the program returns the error RFC_NOT_FOUND to indicate that it couldn't handle that function module. Together with genericRequestHandler() (the callback function responsible for processing the RFC calls), repositoryLookup() is installed in line 290 of demoServer.c before starting the server.

>> Note

In contrast to the demoClient example, where we could use any name we liked for DEST, here we actually need to use the system ID of the back end. Otherwise, the repositoryLookup() function won't be able to find the correct logon data.

The way the complete mechanism works from end to end is as follows: In the sapnwrfc.ini file (see the Samples.zip file) we defined two sections for each back-end system (i.e., MBS_REG, which contains the necessary parameters to register the server, and MBS, which contains the necessary parameters to log on to the back end and access the DDIC).

In the current example, we use two back-end systems to make things more interesting. Substitute all values with values corresponding to your environment. If you only have one SAP system at your disposal, let both registered servers point to different RFC destinations on the same system.

After installing the two callback functions, the program uses the two _REG entries in sapnwrfc.ini to start listening at the two RFC destinations. This is similar to how it's executed in stfcDeepTableServer. Despite our earlier recommendation, we use the same loop to dispatch calls from both systems. Setting the timeout value to 2 seconds in RfcListenAndDispatch() makes this possible. The only side effect is that in the worst case, an incoming call may have to wait for 2 seconds before it is processed. Setting up a multithreaded program here is beyond the scope of this article, because we would need to provide two different versions of the program for this: one for Windows and one for Unix.

When a request comes in from an SAP system, the RFC library searches for a matching pair of metadata description and implementing function, as outlined in the section "Designing a simple server" on page 2. This time, the process will reach step 3 in that section, and will continue as follows:

- If this is the first time that this function call is received from the back-end system with that particular system ID, the RFC library calls the repositoryLookup() function. If it's not the first time, this step is omitted, and the RFC library simply uses the previous call's result and omits the next step.
- The repositoryLookup() function takes the system ID from the

current call attributes and opens a connection to that system. Since an entry for that system ID is defined in sapnwrfc.ini, this connection attempt should be successful. Then, this connection is used to look up the function description, and the result is returned to the RFC library. If the repositoryLookup() function returns RFC_NOT_FOUND, the RFC library aborts the current call and sends a SYSTEM_FAILURE to the back end. Otherwise, the RFC library uses the returned metadata object to create a data container and fills it with data imported from the back end.

- Next, the RFC library calls the genericRequestHandler() function (the second in the RfcInstallGenericServerFunction() step) and passes the data container to it. Then, genericRequestHandler() processes the RFC call, as described next.
- The rest of this process is much like the process in the simple server case: After genericRequestHandler() returns, the RFC library sends either the export data or an error message back to the SAP system, depending on the return code of genericRequestHandler().
- Finally, the RFC library destroys the data container to free the memory again. Don't keep references to structures or tables from that data container. If you need to use some of the data later on, clone it.

Our discussion of RfcListenAndDispatch() is now complete. So, now let's look at how genericRequestHandler() processes the incoming RFC calls.

Protecting the server against unauthorized access

One of the first things you see in genericRequestHandler() is a call to the checkAuthorization() function. This demonstrates how an RFC server program can implement an access control list (ACL) for the function modules it offers. Basically, you can use the same mechanisms here as you do in the classic RFC SDK; SAP Note 934507 describes them in detail. The only difference is that in the classic RFC SDK, checkAuthorization() is implemented as another callback function, called from RfcDispatch(). SAP NetWeaver RFC SDK has dropped this concept because there's no reason to call it every time an RFC request is received. Developers who want to perform authorization checks can always implement such a function and call it from those server functions they want to protect. Other function calls don't need to be burdened with it.

The demo implementation prints a few key details about the current caller, such as the user name and the back end's system ID, and then asks the controller of the demoServer program whether to allow that function module to execute.

For a comparison of the differences and similarities between the classic RFC SDK and the SAP NetWeaver RFC SDK in protecting against unauthorized access, see the section "Protecting the server against unauthorized access" in the sidebar that begins on page 9.

>> Note

It's important to get the metadata from the correct back-end system because a function module may differ from release to release. The same caveats set out in the section "A generic RFC client program" in the November/December 2007 article apply here.

Processing the input parameters

This is similar to how you processed the output parameters for the RFC client in the previous article. The demo program just prints to the console all the data that it got from the back end. The only difference is that in the client case, the parameters that the program got from the data container were the results of a function call (EXPORTING, CHANGING, or TABLES). Now, in the RFC server case, the parameters are the inputs to a function call (IMPORTING, CHANGING, or TABLES). Other than that, the printImports() function uses the same recursive mechanism when reading parameters from the data container and printing them to the console as printExports() did in the demoClient program.

One more point may be noteworthy: For setting up the recursive traverse through the data structures, we needed a metadata description of these structures. In the demoClient program, we already had the necessary metadata description because we needed it to create the data container. But now, in the server case, the RFC library only passes the data container to the genericRequestHandler() function. Where do you get the necessary metadata description? Do you need to perform another lookup like the one in repositoryLookup()? No! Recall the discussion of the "lazy" memory-allocation mechanism in the first article: For creating necessary subcontainers on demand, a data container must keep a reference to the metadata description from which it was created. You can get that reference through RfcDescribeFunction() and then use it in the recursion.

Throwing exceptions

The SAP NetWeaver RFC library enables an external server program to throw the full range of errors that an ABAP function module can throw. Furthermore, it can issue a SYSTEM_FAILURE, which only the kernel can trigger on the SAP R/3 side. If an implementing function (e.g., genericRequestHandler()) needs to signal a serious error condition, it has to do two things:

- Fill the fields in the errorInfo structure that correspond to the type of error the function wants to throw.
- Exit with the return code corresponding to that type of error.

In addition to transmitting the error message back to the SAP system, triggering an error also has an impact on the state of the RFC connection, depending on the type of error. In other words, a "severe" error will close the connection. Therefore, the error han-

>> Note

A password is not sent from the other side for security reasons. The user must be authenticated when logging on to SAP R/3, so the RFC server program can rely on this. More security is available when using single sign-on (SSO) or secure network communications (SNC).

dling isn't finished just by returning the correct return code from genericRequestHandler(); the dispatch loop in mainU() (or the listen() function in this case) also has a job to do. The complete process works in the following way:

- The server function fills the corresponding subfields of errorInfo and returns one of the error codes.
- RfcListenAndDispatch() takes the error information and sends it to the back-end system in the correct format.
- In case of a system failure or an ABAP message, the gateway cancels the registration, which closes the RFC connection.
- RfcListenAndDispatch() returns the same error code as the server function (or RFC_COMMUNICATION_FAILURE if a network problem occurred in the second step).
- The dispatch loop needs to check the return code. If it is RFC_NOT_FOUND, RFC_EXTERNAL_FAILURE, or RFC_ABAP_MESSAGE, the dispatch loop reopens the connection using RfcRegisterServer() before processing the next iteration of the dispatch loop. If the return code is RFC_COMMUNICATION_FAILURE (or RFC_CLOSED, indicating someone on the back end has manually canceled your registration, by using SMGW, for example), then trying to reconnect may be worthwhile. A firewall timeout may have closed the connection. But if the reconnect also fails, the current demo program will give up. See **Figure 1** for an explanation of the possible error types and their effects. For more information, see the sidebar on page 9.

In an RFC client program, the error handling after RfcInvoke() is the critical point of the program; in the RFC server case, the error handling after RfcListenAndDispatch() is the most important. Let's take a look at the RfcListenAndDispatch() section as shown in **Figure 2** on page 8 (see the listen() function in demoServer.c).

For RFC_OK, RFC_RETRY, and RFC_ABAP_EXCEPTION, there's nothing left to do except to execute the loop again and wait for the next call. We have included these responses here only for the sake of completeness. The difference in error handling after an RfcInvoke() in the RFC client is that, in the server, the registration is lost in three cases:

The implementing function throws an ABAP message (indicated by return code RFC_ABAP_MESSAGE).

- The implementing function throws a system failure (corresponding to return code RFC_EXTERNAL_FAILURE in the server and RFC_ABAP_RUNTIME_FAILURE in the client).

The incoming call cannot find the function module description (indicated by return code RFC_NOT_FOUND).

>> Note

We are reusing code here, and therefore, demoServer.c needs to be linked with the same helperFunctions object module as demoClient.c.

Error type	Corresponds to ABAP statement	In C triggered via return code	Fields to fill in error info	Effect on connection	Effect in the back end (ABAP)
ABAP exception	RAISE <exception key>	RFC_ABAP_EXCEPTION	key	Remains open	SY-SUBRC is set corresponding to the exception key in the EXCEPTIONS clause.
ABAP exception with details	MESSAGE ... RAISING <exception key>	RFC_ABAP_EXCEPTION	key abapMsgType abapMsgClass abapMsgNumber abapMsgV1-V4	Remains open	As above. The following fields are filled: SY-MSGTY, SY-MSGID, SY-MSGNO, and SY-MSGV1-V4.
ABAP message	MESSAGE ...	RFC_ABAP_MESSAGE	abapMsgType abapMsgClass abapMsgNumber abapMsgV1-V4	Is closed	SY-SUBRC is set corresponding to the SYSTEM_FAILURE key in the EXCEPTIONS clause and the SY-MSG fields are filled as above.
System failure		RFC_EXTERNAL_FAILURE	message	Is closed	SY-SUBRC is set corresponding to the SYSTEM_FAILURE key in the EXCEPTIONS clause and the parameter specified in the MESSAGE addition is filled.

Figure 1 How to trigger the various error types from a server function

```
01 rc = RfcListenAndDispatch(*pserverHandle, 2, &errorInfo);
02 switch (rc){
03 case RFC_OK:
04 case RFC_RETRY:
05     break;
06 case RFC_ABAP_EXCEPTION:
07     printfU(cU("ABAP_EXCEPTION in implementing function: %s\n"), errorInfo.key);
08     break;
09 case RFC_NOT_FOUND:
10     printfU(cU("Unknown function module: %s\n"), errorInfo.message);
11     refresh = 1;
12     break;
13 case RFC_EXTERNAL_FAILURE:
14     printfU(cU("SYSTEM_FAILURE has been sent to backend: %s\n"),
                errorInfo.message);
15     refresh = 1;
16     break;
17 case RFC_ABAP_MESSAGE:
18     printfU(cU("ABAP Message has been sent to backend: %s %s %s\n"),
                errorInfo.abapMsgType, errorInfo.abapMsgClass,
19              errorInfo.abapMsgNumber);
20     printfU(cU("Variables: V1=%s V2=%s V3=%s V4=%s\n"), errorInfo.abapMsgV1,
21              errorInfo.abapMsgV2, errorInfo.abapMsgV3, errorInfo.abapMsgV4);
22     refresh = 1;
23     break;
24 case RFC_COMMUNICATION_FAILURE:
25 case RFC_CLOSED:
26     printfU(cU("Connection broke down during transmission of return values:
                %s\n"), errorInfo.message);
27     refresh = 1;
28     break;
29 }
30
31 if (refresh){
32     printfU(cU("Trying to reconnect...\n"));
33     *pserverHandle = RfcRegisterServer(params, 1, &errorInfo);
34     if (*pserverHandle == NULL){
35           printfU(cU("Unable to reconnect to %s: %s: %s\n"), params->value,
36                      RfcGetRcAsString(errorInfo.code), errorInfo.message);
37           printfU(cU("Stopping to listen at %s"), params->value);
38     }
39 }
40
41 if (rc != RFC_RETRY && *pserverHandle != NULL && listening)
42     printfU(cU("\nWaiting for request...\n"));
```

Figure 2 Error-handling after an RfcListenAndDispatch()

How an ABAP program can react to error messages from an external RFC server

Normally, you would test the capabilities of the demoServer program from SAP R/3 using transaction code SE37, entering the name of any remote-enabled function module, then entering the name of the RFC destination (e.g., MBS_REG) and a few input parameters, and finally executing the function module. SE37 would then call the function module on the specified RFC destination and display the results.

However, when it comes to catching exceptions, the capabilities of SE37 are very limited. Until SAP R/3 4.6D, SE37 could only catch declared ABAP exceptions. Nothing else was caught and, therefore, resulted in SE37 dumping. In newer releases, SE37 catches SYSTEM_FAILURE but only displays the contents of the MESSAGE parameter. Any other error details are lost.

Thus, you need to call the demoServer from an ABAP report, if you want to see all the available error details. Please see the ABAP sample code in CALL_EXTERNAL_SERVER.abap. This also illustrates the meaning of the last column of the table in **Figure 1**.

In all three cases, as well as when the connection has been interrupted on the network level, the listen() function tries to reestablish registration once before giving up.

For the differences and similarities in throwing exceptions between the classic RFC SDK and the SAP NetWeaver RFC SDK, see the section "Throwing exceptions" in the sidebar below.

Setting the output parameters

Now let's assume that the user of demoServer doesn't want to throw an error but wants to let the function call end successfully. Then, the program collects values for the export parameters to be returned to the back end in much the same way as the demoClient program collects import parameters to send with the RFC request. The only difference between the functions fillExports() in demoServer.c and fillImports() in demoClient.c is that in the server you fill the EXPORTING, CHANGING, and TABLES parameters instead of the IMPORTING, CHANGING, and TABLES parameters, as you do in the client.

One more point, however, deserves some extra attention. An RFC client can reduce the amount of network traffic and memory consumption by deactivating unwanted return structures and tables via RfcSetParameterActive(), as described in the first article.

Similarly, an SAP system that is playing the role of an RFC client can deactivate a parameter before sending the request to the server program. (Just omit the parameter from the CALL FUNCTION statement in ABAP.) So, typically, the RFC server program should check

Comparing SAP NetWeaver RFC SDK to the classic RFC SDK

This article discusses four aspects of RFC server programming:

- Basic functionality
- Protecting the server against unauthorized access
- Throwing exceptions
- General considerations

When comparing these to the classic RFC SDK, please note the following differences and similarities:

Basic functionality

When comparing how to write RFC server programs for the classic RFC SDK and the SAP NetWeaver RFC SDK, let's look at two aspects: First, how to wait for and receive the next incoming request, and, second, how to process a request.

1. In RFC server programming, the main difference between the two SDKs is this: The SAP NetWeaver RFC SDK provides only one way to design a server program; the classic SDK had the following four basic designs (plus a number of variations):

Continues on next page

Continued from previous page

a. You can install a C callback function and then wait for it to execute using the blocking call RfcDispatch(). This technique most closely resembles the design of the SAP NetWeaver RFC SDK.

b. You can install a C callback function and then periodically check for incoming requests via the non-blocking call RfcListen(). Once RfcListen() indicates that a new request is waiting to be dispatched, you process it with RfcDispatch().

The next two ways to design a server program work without an installed callback function:

c. You can use the blocking call RfcGetName() to wait for the next incoming request. RfcGetName() gives you the name of the function module, and you can process the request within the same function that called RfcGetName().

d. You can use the non-blocking call RfcListen() to periodically check for an incoming request, and once one is available, you can get its name via RfcGetName() and process it in the same function.

In addition to those four basic principles, a number of variations were possible, which increased the total number of possible designs to 15:

– In b and d, above, instead of RfcListen(), you could use the semi-blocking call RfcWaitForRequest(), which waits for a specified number of seconds and then returns, unlike RfcListen(), which returns immediately.

– In c and d, above, instead of RfcGetName(), you could use one of the family of calls, RfcGetNameEx(), RfcGetLongName(), or RfcGetLongNameEx().

The two functions with "Long" in their name allow you to process ABAP function modules whose names are longer than 30 characters, and the two functions ending in "Ex" will notify you whenever the RFC library has automatically processed one of the system calls (RFC_PING, RFC_SYSTEM_INFO, RFC_DOCU, and so on). The two functions without "Ex" don't return after a system call has been processed; they just continue waiting for the next non-system call.

In our opinion, all these different ways of achieving almost the same thing added quite a bit of complexity and confusion to the topic. The flexibility gained by this variety isn't very important if you consider the following:

– Whether a request is processed within the same function or within an extra callback function is of no significance. When you want to handle more than one function module, you can put the logic for each module into a separate function and call the corresponding function depending on the function module name. Things are a bit more convenient if the RFC library does this for you.

– All the features that allow waiting for requests via the non-blocking calls were added mainly for single-threaded programs that need to perform other tasks in parallel. Today, however, there's no magic to writing multi-threaded programs, so these features are no longer necessary. Just put RFC dispatching into a separate thread. In a single-threaded program that is performing multiple tasks in parallel, any severe problem with the RFC connection (or any serious problem in another task, for that matter), such as a deadlocking call, affects all the tasks in that program. But when putting each task into a separate thread, a problem in one task won't affect the other tasks (unless it's a severe bug, such as a segmentation fault, which kills the entire process). This clean separation of different tasks will result in more robust programs.

The SAP NetWeaver RFC SDK now offers only one possible design: basically, an optional timeout for those who absolutely need to do other tasks within the dispatch loop. This timeout comes in handy because it allows you to shut down the RFC server in a clean, controlled manner.

2. There is only one way to process an incoming call. This hasn't changed from the classic SDK to the SAP NetWeaver RFC SDK. However, the necessary steps for processing a function call are completely different. In the classic SDK you had to use RfcGetData() to obtain a request's input data, interpret it the right way, build up the output data in the correct format, and then send it off using RfcSendData(). In the SAP NetWeaver RFC SDK, the RFC library performs all these tasks internally. You only need to work on the data container, which is handed to the processing function.

Continues on next page

Continued from previous page

This eliminates one common type of problem: If your processing function ended before it called RfcSendData(), it would leave the underlying RFC connection in an inconsistent state, sometimes leading to a crash when the next call came in from the back end. The SAP NetWeaver RFC library now makes sure that the connection state is always set correctly.

>> Note

The new mechanism isn't really a loss of flexibility: Implementing different server functionality for different RFC destinations of the same back end is still possible by evaluating the program ID field of RFC_ATTRIBUTES and then dispatching the call further to one implementation or the other based on the value of that field.

Protecting the server against unauthorized access

The concepts in this section are pretty much identical for both SDKs. There is one minor difference, though: In the RfcDispatch() model, you can install the checkAuthorization() function as another callback function, which the RFC library will execute before executing a server function. In the RfcGetName() model, however, you must call the checkAuthorization() function explicitly when the function module is processed. The SAP NetWeaver RFC SDK follows the second approach for the following reason: In our opinion the authorization check should always be encapsulated into a reusable function. Then, it doesn't make much difference whether the RFC library calls it or you do.

Having the RFC library use checkAuthorization() as a callback function even has two slight disadvantages here: It makes the design a bit more complex, and, once it is installed, the callback function is always executed. If you have function modules that anyone can invoke (such as an anonymous access), executing the callback function is superfluous. You need to ensure that the callback function always allows access, which may unnecessarily complicate the checkAuthorization() function. When you call the function yourself, you can simply omit the call in those function module implementations that don't need protecting.

Throwing exceptions

Throwing an exception from a server function is completely different in the classic RFC SDK. First, it only supports plain ABAP exceptions and system failures. Instead of just returning the corresponding error code from the server function, the developer has to call a function: either RfcRaise() or RfcRaiseTables(), depending on whether the function module being processed has a TABLES parameter. This strange process has been streamlined considerably in SAP NetWeaver RFC SDK.

General considerations

In principle, it's possible to set up a generic RFC server with the classic RFC SDK. You can install a server function for the function module name %%USER_GLOBAL_SERVER, which the RFC library then calls whenever it can't find another handler function for an incoming call. This is similar to how the SAP NetWeaver RFC SDK does it. Dynamically determining the necessary metadata information to interpret the call is quite a complicated task, but SAP Java Connector (JCo) is proof that it can be done.

At this point we should mention one more difference in how you install server functions: In the classic SDK, you can install all server functions either for a particular RFC_HANDLE or globally. In the SAP NetWeaver RFC SDK, you can install them either for a particular system ID or globally. The handle-based installation turned out to be much too fine-grained. It shouldn't be necessary to have different function implementations for different registrations on the same back end. But different implementations for different back ends may be occasionally necessary, and this is simplified in SAP NetWeaver RFC SDK.

to see whether the RFC client has requested a particular parameter before creating and sending the data for it. Perhaps you don't need to do this for every exporting parameter — it's less effort to just set that parameter than to do all the checking, and the RFC library won't send the data for unrequested parameters anyway. But when you need to create a large table with several thousands of lines of information, it may be worth checking to see whether the client needs it all. You can perform this parameter request check with the `RfcIsParameterActive()` function.

To learn the differences between how the classic RFC SDK and the SAP NetWeaver RFC SDK handle output parameters, see the section "General considerations" in the sidebar that begins on the previous page.

Conclusion

The SAP NetWeaver RFC SDK makes it easy for RFC server applications to react flexibly to all kinds of situations — including errors — and to give the ABAP side meaningful details about any error situations. It's also easy to set up a server that can receive calls from different SAP systems simultaneously. Just be careful if the interface of the involved function modules differs between SAP systems.

The first two articles have discussed standard synchronous RFC communications. In our final article, we will delve into the details of a number of special features. These include transactional RFC (server and client side), hard-coded metadata descriptions, RFC callbacks, SSO, and SNC.

Ulrich Schmidt joined SAP in 1998 after working in the field of Computational Algebra at the Department of Mathematics, University of Heidelberg. Initially, he was involved in the development of various products used for the communication between SAP R/3 systems and external components. These products include the SAP Business Connector, which translates SAP's own communications protocol RFC into the standard Internet communications protocols HTTP, HTTPS, FTP, and SMTP, as well as pure RFC-based tools, such as the SAP Java Connector and RFC SDK. Ulrich gained insight into the requirements of real-world communications scenarios by assisting in the setup and maintenance of various customer projects using the above products for RFC and IDoc communications.

Guangwei Li joined SAP in 1997 after working in the fields of CAD/CAM, Production Planning and Control, as well as Internet Messaging. Since then his work has been focused on the communications and integration between SAP systems and external systems, especially the external systems running on Microsoft Windows platforms. He has been involved in the development of the SAP DCOM Connector, the SAP Connector for Microsoft .NET, and the RFC SDK.