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

Part 1: RFC client programs

by Ulrich Schmidt and Guangwei Li

Ulrich Schmidt
Senior Developer,
SAP AG

Guangwei Li
Senior Developer,
SAP AG

(full bios appear on page 128)

As part of SAP NetWeaver 7.1, a name that refers to a series of products to be released over the next year or so, SAP is introducing a new software development kit (SDK) for remote function call (RFC) communications: SAP NetWeaver RFC SDK. It is the successor to the well-known "classic" RFC SDK for SAP R/3, and you can use it in C/C++-based applications to communicate with SAP back-end systems ranging from SAP R/3 4.0B to the latest SAP NetWeaver systems. This article takes a deeper look at the design of SAP NetWeaver RFC SDK and explains the ideas behind it.

We found that many problems with RFC communications come from misunderstanding the existing API, therefore, we hope that a clear understanding of the concepts of SAP NetWeaver RFC SDK will help you to write efficient, robust RFC programs and avoid potential pitfalls.

This is the first installment in a series of three articles that will explore the advanced features of SAP NetWeaver RFC SDK, in particular, and RFC communications between an SAP system and an external C program, in general. If you are a developer or consultant who needs to access legacy applications (written in C or in any language with a C interface, such as COBOL or FORTRAN) from within ABAP applications, or to access ABAP functionality from within legacy applications, then this series of articles is for you. If you need to provide adapters or add-on software for SAP systems — even if you have an existing solution based on the classic RFC SDK — then it's for you, too.

However, if your external software is written in Java or in a language from the Microsoft .NET Framework, then you shouldn't use SAP NetWeaver RFC SDK. Instead, you should use SAP Java Connector (JCo) or SAP .NET Connector, which you can download from the

SAP Service Marketplace.[1] If you are using scripting languages, such as Perl, Python, or Ruby, then you will find free third-party connectors for these languages on the SDN.[2]

This article is not a comparison between classic RFC SDK and SAP NetWeaver RFC SDK. However, if you are already familiar with classic RFC SDK, we will point out a few noteworthy differences along the way,[3] including:

- New functionality that was not available in the classic RFC SDK
- Special communication features (e.g., nested structures, Unicode communication, or exceptions) that were difficult to accomplish with the classic RFC SDK, but can now be done easily using SAP NetWeaver RFC SDK
- Areas in which the architecture was changed completely, so that a "rethinking" is necessary to avoid pitfalls

Although the basics of SAP NetWeaver RFC SDK are not covered in this series of articles, this first article provides a detailed explanation of two basic concepts used throughout SAP NetWeaver RFC SDK: *metadata descriptions* and *data containers*. These concepts did not exist in the same form in the classic RFC SDK, so a thorough understanding of them is critical to understanding how SAP NetWeaver RFC SDK works. The rest of this article and the two remaining articles in the series will put this knowledge to use.

Proceeding from this foundation, we will explore the finer points of RFC client programming (i.e., writing C programs that call ABAP function modules in an SAP system). Then we'll build a generic client, step-by-step, that can call any remote-enabled function module in the back-end system. To do this, we'll use nearly every feature in SAP NetWeaver RFC SDK. This way, you can see them in action and learn how to use them in your own programs. These features include retrieving and caching structure information (metadata descriptions), traversing nested input or output parameters, automatic code-page-handling, and working with exceptions (ABAP exceptions, ABAP messages, and system failures).

The second installment of this three-article series (to be published in an upcoming *SAP Professional Journal*) will take a detailed look at RFC server programming (i.e., writing C programs that ABAP programs can call). We will also develop a generic server program that can receive and process any arbitrary function call from the back-end system.

The final article will look at some special features, such as transactional RFC (tRFC) and queued RFC (qRFC), hard-coded metadata descriptions, RFC callbacks, and single sign-on (SSO), or Secure Network Communications (SNC).

You can download the source code examples contained in this article from www.sappro.com/downloads.cfm. See the download section corresponding to this issue of *SAP Professional Journal*. We recommend having the source code available as you read the article, because only portions of the code appear in print.

Also, to compile and run the sample programs, you need to download SAP NetWeaver RFC SDK from the SAP Service Marketplace. SAP Note 1025361 tells you where to find this SDK for various operating-system (OS) platforms. Make sure that you have at least patch level 1 of the SDK because a few of the functions mentioned in this article weren't available at patch level 0.

We have a lot to cover, so let's get started.

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

[2] Logon credentials are required to access the information in the SDN at https://www.sdn.sap.com/irj/sdn.

[3] There is a sidebar at the end of the article on pages 126 and 127, which is divided into sections. I refer to the different sections in the sidebar throughout the article.

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 BAPIs (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 to have previous knowledge of the classic RFC SDK.

Metadata descriptions and data containers

As previously mentioned, there are two new SAP NetWeaver RFC SDK concepts, which are basic ingredients of the programming model used in client and server applications alike: metadata description and data containers. They are complementary to each other; therefore, it may not be easy, at first, to see the differences between them and why they are both necessary. This article will help you to clearly separate the concepts and to know when to use each.

Metadata descriptions

Metadata is a high-level description of data structures (in the way that a city map describes a city). SAP NetWeaver RFC SDK has two types of metadata descriptions: *structure definitions* (C-type RFC_TYPE_DESC_HANDLE) and *function descriptions* (C-type RFC_FUNCTION_DESC_HANDLE).

- A *structure definition* gives detailed information about all the fields of a STRUCTURE or TABLE: their names, data types, and lengths. If one field is also a structure (or even a table), then a structure definition may contain other structure definitions as sub-descriptions.
- A *function description* gives detailed information about all the IMPORTING, EXPORTING, CHANGING, and TABLES parameters of a remote-enabled function module (RFM). It also gives a complete list of the ABAP exceptions that the function module may throw. For scalar parameters, a function description is similar to a structure definition: It simply gives their names, data types, and lengths. However, if a field contains a structure or table, then the function description has a structure definition as a sub-description.

To illustrate this concept, let's look at an example, the function description of `STFC_STRUCTURE`. **Figure 1** shows the essential portions of the metadata

Name	Type	Direction	NUC* length	UC length	Decimals	Type description
IMPORTSTRUCT	STRUCTURE	IMPORT	144	264	0	Pointer to RFCTEST
ECHOSTRUCT	STRUCTURE	EXPORT	144	264	0	Pointer to RFCTEST
RESPTEXT	CHAR	EXPORT	255	510	0	NULL
RFCTABLE	TABLE	TABLES	144	264	0	Pointer to RFCTEST
* "NUC" stands for Non-Unicode and "UC" stands for Unicode.						

Figure 1 Metadata description for function module STFC_STRUCTURE

description for this function module. In your programs, this would correspond to an RFC_FUNCTION_DESC_HANDLE.

In this metadata description, three of the four parameters are of type STRUCTURE or TABLE and have the same structure definition: RFCTEST. But the description of RFCTEST is kept in memory only once and is referred to by pointer from within the function description.

Figure 2 shows the metadata description for the structure RFCTEST, to which the function module STFC_STRUCTURE (and possibly other function modules using the same structure in their importing/exporting or table parameters) refers. In your programs, this would correspond to an RFC_STRUCTURE_DESC_HANDLE.

Note that the offset isn't just the sum of the lengths of the previous fields because integer (INT) and floating-point (FLOAT) types have restrictions on possible start addresses (e.g., an INT4 needs to start at an address divisible by 4 and a FLOAT must start at an address divisible by 8); the field RFCINT4 has an offset of 20 (24 in a Unicode system) instead of the expected 17 (22 in Unicode). Character (CHAR) types, on the other hand, may start at any address in Non-Unicode systems and at any even address in Unicode systems.

All this should demonstrate that it might be a bit tricky to get complicated structure definitions correct. Therefore, use SAP NetWeaver RFC SDK's automatic lookup features (described below) whenever possible. If you construct your metadata descriptions manually and get them wrong, you will corrupt your data. Our third article will explore the only scenarios where manual construction of metadata descriptions may be unavoidable.

Metadata descriptions have two purposes:

- As tools to dynamically explore the parameters of a function module — e.g., for a generic user interface (UI) that automatically construct forms to collect the inputs and display the outputs
- As blueprints to create the corresponding data containers needed to process the actual function calls (see the section "Data containers" on page 116)

SAP NetWeaver RFC SDK offers two convenient functions for looking up metadata descriptions: `RfcGetTypeDesc()` and

Name	Type	NUC length	NUC offset	UC length	UC offset	Decimals	Type description
RFCFLOAT	FLOAT	8	0	8	0	16	NULL
RFCCHAR1	CHAR	1	8	2	8	0	NULL
RFCINT2	INT2	2	10	2	10	0	NULL
RFCINT1	INT1	1	12	1	12	0	NULL
RFCCHAR4	CHAR	4	13	8	14	0	NULL
RFCINT4	INT	4	20	4	24	0	NULL
RFCHEX3	BYTE	3	24	3	28	0	NULL
RFCCHAR2	CHAR	2	27	4	32	0	NULL
RFCTIME	TIME	6	29	12	36	0	NULL
RFCDATE	DATE	8	35	16	48	0	NULL
RFCDATA1	CHAR	50	43	100	64	0	NULL
RFCDATA2	CHAR	50	93	100	164	0	NULL

Figure 2 Metadata description for structure RFCTEST

`RfcGetFunctionDesc()`. You only need to give them an open RFC connection into the back-end system and the name of the function module or structure you want, and they will look up the complete function module or structure definition in the back-end system's data dictionary (DDIC) and return the information in a format you can use — i.e., as C structures of type RFC_STRUCTURE_DESC_HANDLE and RFC_FUNCTION_DESC_HANDLE.

This DDIC lookup is rather expensive performance-wise, so the SAP NetWeaver RFC library has a built-in cache for these metadata descriptions. The actual lookup is only done the first time. If you need the same function module or structure again, you will get the cached result.

One important feature of this cache is that it uses the system ID of the corresponding back-end system as a key. The idea is: If your program communicates with two different back-end systems during its lifetime, let's say system ABC of release 4.6C and system XYZ of release 6.20, then the function module HUGO may have additional fields in the newer back-end release. So, to process function calls to both systems correctly without losing data or causing data corruption, you need two different metadata descriptions: HUGO/ABC and HUGO/XYZ.

The two structures RFC_PARAMETER_DESC (used in function descriptions) and RFC_FIELD_DESC (used in structure definitions) contain one more member variable: void* extendedDescription. The RFC library doesn't use this parameter. It's for application programmers who want to store additional information about various fields within the metadata. For example, for structures used in UIs, the application could retrieve a list of allowed input values (F4-Help values) and store it within the structure definition.

A scalar importing parameter of a function module may also have a default value, a value that the function module works with if the caller doesn't specify a value. `RfcGetFunctionDesc()` retrieves this default value and stores it in RFC_PARAMETER_DESC.defaultValue. The field RFC_PARAMETER_DESC.parameterText will contain the documentation for that parameter, as defined in the back-end system at SE37.

For information on some of the differences between the classic RFC SDK and SAP NetWeaver RFC SDK regarding the concepts of metadata representation, see the section "Metadata descriptions" in the sidebar on page 126.

Let's look at a practical example. Open the file printDescription.c from the samples you downloaded. This program opens a connection to the back end, (see the section "A generic RFC client program" on page 119 for a more detailed discussion of how this can be done). It looks up the description of the function module `W3_GET_MINIAPP_TEXTS` and prints out the result. You need at least an SAP R/3 4.6C back end to run this example, since the function module didn't exist in earlier releases. Just change the logon parameters in lines 14-19 of printDescription.c, compile the program, and then run it.

Note!

The concept of using the system ID of the back-end system as a key has been employed in the SAP Business Connector, and we have seen the following tricky problem occur at a number of customer locations. If two different SAP systems within a system landscape share the same system ID, the DDIC cache can't distinguish between the two. The system to which the SAP Business Connector connected first wins, and calls to the other system will be done with cached structure information from the first system. This may result in a possible loss of data or data corruption for those function modules whose metadata differs across the two systems. This is why SAP requires that no two systems within the same landscape have the same system ID.

Tip!

On Linux and Windows, certain minimum releases of the C runtime are required to execute the programs. See SAP Notes 1021236 (for Linux) and 684106 (for Windows). The compiler and linker options needed to compile programs using the SAP NetWeaver RFC SDK are listed in SAP Note 1056696.

In addition, for Windows users using Microsoft Visual Studio, this is a description of how to set the Visual Studio project properties:

- **General section:** Make sure the CharacterSet field is set to "Use Unicode Character Set."
- **Debugging section:** Under Environment, add something like Path=%Path%;nwrfcsdk\lib.
- **C/C++ section:**
  - **General:** Add the nwrfcsdk\include directory under Additional Include Directories.
  - **Preprocessor:** Add the two preprocessor definitions SAPonNT and SAPwithUNICODE.
  - **Code Generation:** Choose "Multi-threaded DLL (/MD)" as the Runtime Library; selecting "Multi-threaded Debug DLL (/MDd)" may lead to strange problems.
- **Linker section:**
  - **General:** Add the nwrfcsdk\lib directory under Additional Library Directories.
  - **Input:** Under Additional Dependencies, add libsapucum.lib, sapnwrfc.lib, and sapdecfICUlib.lib (if you are using the DecFloat data types).

Data containers

A data container is an actual piece of memory into which you can write the values of the parameters and fields to be sent to the back-end system, and from which you can read the values that the back end sends to you. Both client and server use the same data containers. The only differences are:

- **When your program acts as a client**, you create the container, fill in the imports, and submit the container to the RFC library. The RFC library serializes the data, sends it to the back end, receives the response from the back end, deserializes the data, and fills in the export parameters of the data container. Afterwards, you can read the exports from the container. In the end, you have to destroy the container to release the memory.
- **When your application plays the role of server**, the RFC library receives the call from the back-end system, creates a data container, fills in the import parameters, and passes it to you. You can then read the imports and set the values for the exports. When you are finished, you simply return from your implementation of the server function. The RFC library then takes care of sending the exports to the back-end system and destroying the data container.

A data container is always created from a metadata description, either via `RfcCreateFunction()` for a complete function module (including the necessary sub-containers for structures and tables) or via `RfcCreateStructure()` or `RfcCreateTable()` for a single structure or table. The second possibility is almost never necessary. We can think of one example: When you need to keep some of the function module's data after its corresponding data container has been destroyed. In that case, you should create a separate container for the structure or table of interest and copy the data into it before destroying the container for the function module. However, you can also conveniently achieve this result via the function `RfcCloneStructure()` or `RfcCloneTable()`.

At this point, we should mention one feature that might be good to know for memory-intensive

applications: The RFC library uses a kind of "lazy" memory allocation. For example, sometimes a function module has a lot of large import structures, but you only want to fill a few of them; the others are optional and remain empty so it would be a waste of memory if the RFC library allocated space for all of them. For this reason, sub-containers are allocated only when you actually access them. (The same rule applies to tables.) This is possible because the data container keeps a pointer to the metadata description — the blueprint — from which it was originally created.

To illustrate, **Figure 3** is a code snippet for the function module `STFC_STRUCTURE`.

A similar concept can also reduce the amount of data in the RFC response. Let's assume a function module or a BAPI has several tables that the back-end system usually fills with thousands of lines of data, but you're only interested in one of these tables. You can use the following code to deactivate the table A_TABLE for the function module funcHandle. The back-end system is notified that the client isn't interested in this particular parameter, and it won't even return the data for A_TABLE. This not only prevents your program from allocating memory for this unnecessary table, but also reduces the amount of data sent over the network.

```
RfcSetParameterActive(funcHandle,
    cU("A_TABLE"), 0, NULL);
```

The fields of a data container are read and set, respectively, via the family of `RfcGet<type>()`/`RfcSet<type>()` functions. We won't elaborate on them — because they're self-explanatory — but we just want to point out two useful features:

- You can use the same functions for all three types of data containers: function handles (type RFC_FUNCTION_HANDLE), structure handles (type RFC_STRUCTURE_HANDLE), and table handles (type RFC_TABLE_HANDLE). For the table handles, you only need to make sure that you position the row cursor to the appropriate line before accessing the fields:

```
RfcMoveTo(tableHandle, 42, &errorInfo);
RfcGetString(tableHandle,
    cU("A_STRING_FIELD"), buffer,
    bufLen, &strLen, &errorInfo);
```

```
 1 RFC_FUNCTION_DESC_HANDLE aFunctionDesc;
 2 RFC_FUNCTION_HANDLE aFunction;
 3 RFC_STRUCTURE_HANDLE aStructure;
 4
 5 aFunctionDesc = RfcGetFunctionDesc(connectionHandle, cU("STFC_STRUCTURE"), NULL);
 6 aFunction = RfcCreateFunction(aFunctionDesc, NULL);
 7 /* The previous line only allocated the following memory (on a 32bit platform):
 8    4 bytes to hold a pointer to a future IMPORTSTRUCT (=NULL)
 9    4 bytes to hold a pointer to a future ECHOSTRUCT   (=NULL)
10  510 bytes to hold the value of the export parameter RESPTEXT
11    4 bytes to hold a pointer to a future RFCTABLE     (=NULL) */
12
13 RfcGetStructure(aFunction, cU("IMPORTSTRUCT"), &aStructure, NULL);
14 /* Now the 144 bytes for the 12 fields of the structure IMPORTSTRUCT have
15    been allocated, and the corresponding pointer in aFunction has been
16    initialized to point to that structure. */
```

Figure 3 Code snippet to illustrate when memory is allocated

This code fills the buffer with the value that table field A_STRING_FIELD has in line 42. (Of course, you should check the return codes of `RfcMoveTo()` and `RfcGetString()` to make sure the table really has a line 42 and a field named A_STRING_FIELD.)

Tip!

In the SAP NetWeaver RFC SDK, table lines are numbered from 0 to *n*-1, as is customary in C. In ABAP and in the classic RFC SDK, lines are numbered from 1 to *n*.

- The `RfcGet<type>()/RfcSet<type>()` functions perform an automatic data-type conversion, where possible. For example, if a structure has an INT field and you have a value for it in string form (from a UI, for example), you can simply use:

```
RfcSetString(structHandle,
    cU("AN_INT_FIELD"), stringValue,
    &errorInfo);
```

The RFC library will then convert the string value (stringValue) to an integer value before setting the field. Or if you want to get the value of a FLOAT field in string form, you can use:

```
RfcGetString(structHandle,
    cU("A_FLOAT_FIELD"), buffer, bufLen,
    &strLen, &errorInfo);
```

If you attempt an impossible conversion (such as trying to assign the value "XYZ" to an integer field), you get the return code `RFC_CONVERSION_FAILURE` and the field remains unchanged.

You need to be careful when you release memory. You can free the memory allocated for functions, structures, and tables via `RfcDestroyFunction()`, `RfcDestroyStructure()`, and `RfcDestroyTable()`.

However, first let's continue the above code snippet for `STFC_STRUCTURE` (**Figure 3**). If you call:

```
RfcDestroyStructure(aStructure, NULL);
```

and then continue working with aFunction, you risk getting a segmentation fault, because the pointer for IMPORTSTRUCT within aFunction now points to an invalid address. The following code will definitely lead to a crash:

```
RfcDestroyStructure(aStructure, NULL);
RfcDestroyFunction(aFunction, NULL);
```

because `RfcDestroyFunction()` frees the memory for all associated sub-containers. Since the pointer to

Tip!

You can reuse an RFC_FUNCTION_HANDLE for several calls to the back-end system. But you should be extremely careful to avoid the following two pitfalls:

- An importing field doesn't get a new value before a second call is executed using the same function handle. The value from the previous call is still present in the function handle and will be sent to the back end again in the second call.
- In the second call, the back end doesn't return a value for an exporting parameter for which it returned a value in the previous call. The value from the previous call is still stored in that field, and you may mistakenly believe that it is a part of the current call's response.

When in doubt as to whether a certain parameter is correctly overwritten with a new value, it's better to manually set it to the initial value before reusing the function handle. For table parameters, use `RfcDeleteAllRows()` to clear the table before the next call.

IMPORTSTRUCT within aFunction is not `NULL`, the code will try again to free aStructure, which was already freed in the previous line.

In general, you need to call `RfcDestroyX()` for those containers that you created yourself via `RfcCreateX()`. You should never delete the containers that you got from `RfcGetStructure()/ RfcGetTable()` or from a table via `RfcGetCurrentRow()/RfcAppendNewRow()/ RfcInsertNewRow()`. Nor should you ever delete the function handle (RFC_FUNCTION_HANDLE), which the RFC library passes to you when your application plays the role of server.

For a summary of the differences between the classic RFC SDK and SAP NetWeaver RFC SDK in the way function module parameters are represented and exchanged between your program and the RFC library, refer to the section "Data containers" in the sidebar on page 126.

After so much theory, next we'll put the two concepts we learned — metadata descriptions and data containers — into action. The data containers are used for executing function modules, and the metadata description is used for doing this dynamically — i.e., without knowing at compile time which function modules are going to be executed.

A generic RFC client program

Let's develop an RFC client program to dynamically execute any given function module. The program will perform the following steps:

1. Log on to the back-end system.
2. Ask the user for the name of a remote-enabled function module.
3. Look up the function description in the back-end system's DDIC, and then ask the user for input values for the function module's IMPORTING, CHANGING, and TABLES parameters.
4. Execute the function module in the back end.
5. Display either of the following:
   - The values of the EXPORTING, CHANGING, and TABLES parameters if the call ended successfully.
   - The error information if the call ended with an ABAP exception, an ABAP E-, A-, or X-message, or a system failure.

> ***Note!***
>
> For the rest of the article, you should have the two sample files demoClient.c and helperFunctions.c open in your favorite C editor.

To begin, we need to consider how our program is going to get the necessary logon parameters for accessing the back-end system. This is an important point to think about, because the logon credentials should be both easy to maintain during the lifetime of the program and at the same time securely protected against unauthorized access.

Log on to the back-end system

In our first example (printDescription.c) the logon parameters had been given to `RfcOpenConnection()` in an RFC_LOGON_PARAMS array. Of course, hard-coding the logon parameters into the program is a terrible idea. Direct usage of the parameters "user," "ashost," "mshost," and so on, only makes sense if you get them dynamically (i.e., from a UI or from a Lightweight Directory Access Protocol — LDAP — directory). If they are hard-coded into the program, then the program will have to be recompiled whenever one of the parameters changes. This may be quite often because some security policies require the password to be changed regularly. In those cases where the logon data remains more or less static (except for periodical password changes), using the parameter "dest" with the sapnwrfc.ini file is a convenient way

to organize logon data. We'll do that for the rest of the examples. We'll program against the destination "SPJ," and you only have to fill the logon parameters of your back-end system into the sapnwrfc.ini file included in the examples. If you put the completed file in the current working directory of the demoClient program, it should work at your site without any code changes. In a real-life scenario you would use the operating system's file system protection and encryption mechanisms to ensure that the sapnwrfc.ini file can only be accessed by the client program and an administrator.

After logging on (see demoClient.c, line 90), we perform the first important error-handling routine. A detailed explanation of the four possible error types follows in the discussion of the `RfcInvoke()` step in the "Error-handling" section on page 122. For now, suffice it to say that only two errors can occur at this point in the code:

1. Any kind of problem with network communications (unknown host, back end currently down, etc.), which is indicated by a value of `errorInfo.code = RFC_COMMUNICATION_FAILURE`
2. Any kind of problem with the user (wrong password, user locked, insufficient authorization for executing RFCs, etc.), which is indicated by `errorInfo.code = RFC_LOGON_FAILURE`

In these cases, the field errorInfo.message gives a detailed description of what went wrong.

You may wonder what kind of minimum authorizations a user needs in production environments so the program can log in and execute the function modules it needs. In a production situation, it's not advisable to run programs with a user who has the SAP_ALL authorization. SAP Note 460089 discusses this in much detail. We'd like to add a few more thoughts about the concepts outlined in that note.

To comprehend SAP Note 460089 properly, you need to understand the concept that there are basically four different kinds of authorization objects:

a) The bare technical authorization for performing RFC calls (This is a special authorization necessary to log in to SAP R/3 via RFC. Even if a user has permission to execute function module A and can do so while being logged in via SAPGUI, that user won't be able to execute module A via RFC without the "technical authorization" to use RFC.)

b) The authorization to make metadata lookups in the back-end system's DDIC (This is not needed when using hard-coded metadata.)

c) The permission to call the function module (or function group) that you want to call

d) The authorization object on the application level, which the function module may check internally

Based on these different types of authorization objects, the back-end system performs a complicated sequence of authority checks when an RFC client program tries to execute a function module. Let's illustrate this process with an example of a client program trying to execute the function module `BAPI_USER_CREATE`. In this case, the interaction between back-end system and client program runs through the following phases:

1. At logon time while `RfcOpenConnection()` is still running, the system checks whether the user has the permission to execute RFCs (authorization object S_RFC with a number of technical function groups as outlined in SAP Note 460089). Some authorization checks on the kernel level make sure the user is allowed to use RFC in the first place, though this is more of a "technical" question as opposed to the "business logic" authorizations.
2. The client program looks up the metadata for BAPI_USER_CREATE, and the system checks permissions to access the DDIC (S_RFC with a number of function groups that access the DDIC). This happens during RfcGetFunctionDesc().
3. RfcInvoke() is called and the back-end system's RFC runtime module first checks whether the user has the authorization to execute the function module BAPI_USER_CREATE (again authorization object S_RFC with function group SU_USER). If so, the system starts that function module on behalf of the user; if not, the call is aborted with a SYSTEM_FAILURE.

4. The ABAP code of BAPI_USER_CREATE begins executing, and within the innards of that function module an authority check against the authorization object S_USER_SAS is performed. If the check fails, the BAPI appends a corresponding error message to its RETURN table and exits. From the RFC point of view, the call has ended successfully in that case, but BAPIs are not allowed to throw exceptions. Therefore, the BAPI needs to use its RETURN structure or table to indicate an error situation. (Other function modules may throw an exception if their internal authority check fails.)

In the demo examples accompanying this article and in smaller applications, it is okay to use one user for everything. However, if you are creating more complex systems, you'll probably want to use one designated user with authorizations a and b for the metadata lookups (repository user) and one or more other users with authorizations a, c, and d for the actual function calls (application user).

After `RfcGetFunctionDesc()`, another error check is due as the function module the user requested may not exist in the back-end system.

Now everything is prepared for collecting the values for the function module's input parameters, whatever these may be. In the helper functions of the following section you can see how the metadata descriptions "intertwine" with the data containers in order to achieve this end. These functions use the metadata descriptions for finding out what parameters there actually are, and the data container for storing the values for each of these parameters.

Setting the input parameters

Let's take a look at how the demoClient program asks the user for input. The basic principle for looping over the parameters has already been shown in printDescription.c. In that example we knew that `W3_GET_MINIAPP_TEXTS` only had flat structures, so a simple loop over the fields is sufficient. In general, however, a function module may have "nested" structures and tables, so we need a recursive approach here. This is accomplished via the helper functions `fillStructure()` and `fillTable()`, which recursively call each other.

The entry point into the recursion is the function `fillImports()`, which is called in line 123 of demoClient.c and implemented in helperFunctions.c. It loops over the list of parameters, once each for the IMPORTING, CHANGING, and TABLES parameters, and calls `fillParameter()` for each parameter. This function then decides whether a parameter is a scalar type that can be read directly, or if that parameter is a structured type that needs another loop over its sub-fields. If another loop is required, it is performed in `fillStructure()`, which distinguishes between scalar and structured sub-fields and calls itself recursively for the structured fields.

The function `fillTable()` is just a loop that calls `fillStructure()` once for every line to be appended to the table. The same kind of mechanism is also used when printing the outputs of the function module. The family of functions where `fill` is replaced by `print` basically works in the same manner, except instead of reading a value from the console and setting it in the data container, these functions read the value from the data container and print it to the console.

The convenient feature of automatic data-type conversion (mentioned in the section "Data containers" on page 116) is used to set values in the data containers. No matter whether the parameter is of type character, integer, floating-point, or raw binary data, you can use `RfcSetString()` to set its value. The RFC library internally converts the string into the correct data type before filling it into the RFC buffers. However, when using this feature, you need to make an error check after every `RfcSetString()`. (Normally, when you are certain that a field exists and that you have the correct data type for it, you can omit the error check.) Here, however, the user might enter non-convertible values. In particular, you need to know that the values must be entered in hexadecimal string format for binary types. You can enter floating-point values as decimal numbers or in scientific notation.

If you already have the value in the correct format in your programs, you can (and should) use the corresponding functions `RfcSetInt()`, `RfcSetFloat()`,

and so on. But `RfcSetString()` is convenient in those cases where you get the values from user input, XML documents, and so on.

Tip!

The SAP NetWeaver RFC library takes all character input in the form of SAP_UC, which is basically a UTF-16LE representation. (If interested, please refer to www.unicode.org for an explanation of the Unicode standard and its various technical representations, such as UTF-8 and UTF-16.) To simplify working with this data type, the sapuc.h header and the libsapucum library provide a set of functions and macros for processing SAP_UC. For example, the macro `cU()` turns a char string literal into an SAP_UC string literal.

Most important for any function from the standard C library dealing with strings, you can obtain a corresponding SAP_UC version by simply adding a "U" to the end of the function name. For example, the sample programs show the use of `printfU()`; there are also functions such as `getenvU()` and `fopenU()`.

In Windows, SAP_UC is equivalent to the wchar_t type, so you can immediately pass the values to any Windows w-functions without problems. On most Unix/Linux platforms, however, the wchar_t type is implemented in a different way. The two functions `RfcSAPUCToUTF8()` and `RfcUTF8ToSAPUC()` may be handy; they convert SAP_UC to and from UTF-8. Functions for processing UTF-8 data should also be available on most platforms.

After the program collected all the values for IMPORTING, CHANGING, and TABLES parameters, it can finally execute the function module on the back-end system using `RfcInvoke()`.

Error-handling

`RfcInvoke()` is the critical point where all kinds of different errors may happen. Basically, the different types of errors correspond to the different locations in the execution stack where an error can occur:

- **An ABAP exception can be thought of as an "error on application level."** An ABAP exception is raised in the application's ABAP code. It usually means that the function module you called doesn't quite know what to do with the inputs you gave it. For example, if you called a (hypothetical) function module READ_ORDER_DETAIL with an invalid or non-existent order number, it would throw the ABAP exception ORDER_NOT_FOUND. From the perspective of RFC communications, this isn't a problem. The RFC connection, as well as the user context in the back end, remain alive and can be used for further function calls.

- **A system failure can be thought of as an "error on kernel level."** A system failure comes from the SAP R/3 kernel and indicates a low-level technical problem. For example, while executing a (hypothetical) function module, the SAP R/3 work process may have run out of memory or an unexpected division by zero may have occurred. In these cases the work process would abort execution and generate an ABAP short dump, which can later be analyzed with transaction ST22. As a consequence, the back-end system closes the RFC connection and you need to open a new one, if you wish to execute further calls.

- **ABAP messages are somewhat in between the previous two.** They can come from the ABAP code (using the keyword MESSAGE) or they can come from the kernel side. However, the keyword "MESSAGE … RAISING …" doesn't trigger an ABAP message. It leads to an ABAP exception (as in "RAISE …") with the difference that in addition to the exception key, the type, ID, and number of a T100 message are sent back to the client. In particular, when using "MESSAGE … RAISING …" the RFC connection remains alive.

ABAP messages can be difficult to analyze because there are six types of them, three of which affect RFC communications: E (Error), A (Abend), and X (Exit). The other three are I (Info), S (Status), and W (Warning), but they have no impact on RFC, so we won't discuss them. The three that affect RFC behave quite differently; the only common feature is that after the message is sent, the RFC connection is closed.

E: This is similar to an ABAP exception via "MESSAGE … RAISING …" except that it ends the user session and closes the connection. In addition, error details are written to the current work-process trace, which doesn't happen with ABAP exceptions.

A: This is a more serious error and generates a system log entry with syslog area D0. Transaction SM21 can tell you what went wrong.

X: This finally aborts the work process the hard way and generates a short dump with key MESSAGE_TYPE_X. The error details can be found in transaction ST22.

- **A communications failure can be thought of as an "error on network level."** This error indicates a problem in the network layer. It can range from a misspelled hostname to incorrectly configured routers/firewalls to real hardware problems.

You also need to be aware of one special error: If you get the standard TCP/IP error message "Connection reset by peer," it usually means that some router or firewall between the RFC program and the communications partner has closed the connection. However, it can also mean that there is no network problem at all, but instead the communications partner has crashed without properly closing the connection and sending a final error message (as it would in the case of a system failure).

Therefore, before immersing yourself in a detailed analysis of your network settings, you should first check the target host to see whether the process on the other side has crashed or been terminated, or if the operating system has been unexpectedly rebooted.

When reacting to these errors, you need to know in which cases the RFC connection is still okay. From a "technical" point of view a successful function module invocation and one that ended in an ABAP exception do not differ, so the connection can immediately be reused. The only difference is the payload being transported: In the first case, it's some return values; in the second case, it's an "application level error message" (as opposed to a technical or network error message). In other words, the RFC layer only cares about technical problems on the system or network level.

In case of an ABAP message or a system failure, the connection has been closed, but the system is okay in principle, so it's worth trying to open a new connection and continuing with that system for the next calls. (There may be rare cases where a system failure has been caused by the back end being in deep trouble on its database or operating system level. Then, logging on again won't be of much use since another function call will result in the same system failure again.) When you get a communications failure, any further attempt is typically in vain, so it's probably best to log the error and give up until an administrator can fix the problem.

The switch-block after `RfcInvoke()` in demoClient.c shows what such an error-handling — including re-logon, if necessary — could look like. This is a critical piece of code, as shown in **Figure 4** (on the next page). Because it appears in a similar fashion in most client applications, let's take a moment to discuss it.

First, the network failure needs to be dealt with separately from the other error conditions outside the switch-block. In two of the other error conditions (`RFC_ABAP_MESSAGE` and `RFC_ABAP_RUNTIME_FAILURE`) the connection is being closed normally, not by a network problem. In those cases, you should try to reopen the connection for further calls. You should also try to prepare for situations in which a network failure occurs during an attempt to reopen the connection. In the demo code this is achieved in lines 35 and 41 by overwriting the return code with the result of the reconnect attempts. As a consequence, the if-block in line 49 will now handle

```
01 while(1){
02    //... The necessary preparation for the function call goes here.
03    rc = RfcInvoke(connection, functionContainer, &errorInfo);
04    switch(rc){
05            case RFC_OK:
06                   printExports(functionDescription, functionContainer);
07                   checkForReset(connection);
08                   break;
09            case RFC_ABAP_EXCEPTION:
10                   printfU(cU("%s threw an ABAP Exception: %s\n"),
                         functionName, errorInfo.key);
11                   if (strlenU(errorInfo.abapMsgClass) > 0)
12                          printfU(cU("Message Details: TYPE=%s ID=%s NO=%s V1=%s
                            V2=%s V3=%s V4=%s\n"),
13                                                 errorInfo.abapMsgType,
14                                                 errorInfo.abapMsgClass,
15                                                 errorInfo.abapMsgNumber,
16                                                 errorInfo.abapMsgV1,
17                                                 errorInfo.abapMsgV2,
18                                                 errorInfo.abapMsgV3,
19                                                 errorInfo.abapMsgV4
20                       );
21                   checkForReset(connection);
22                   break;
23            case RFC_ABAP_MESSAGE:
24                   printfU(cU("%s threw an ABAP Message: %s\n"), functionName,
                         errorInfo.message);
25                   printfU(cU("Message Details: TYPE=%s ID=%s NO=%s V1=%s V2=%s
                         V3=%s V4=%s\n"),
26                                        errorInfo.abapMsgType,
27                                        errorInfo.abapMsgClass,
28                                        errorInfo.abapMsgNumber,
29                                        errorInfo.abapMsgV1,
30                                        errorInfo.abapMsgV2,
31                                        errorInfo.abapMsgV3,
32                                        errorInfo.abapMsgV4
33                );
34                connection = RfcOpenConnection(loginParams, 1, &errorInfo);
35                rc = errorInfo.code;
36                break;
```

Continues on next page

Figure 4 Error-handling after an RfcInvoke()


```
37          case RFC_ABAP_RUNTIME_FAILURE:
38                printfU(cU("%s aborted with a SYSTEM_FAILURE: %s: %s\n"),
                      functionName,
39                                        errorInfo.key, errorInfo.message);
40                connection = RfcOpenConnection(loginParams, 1, &errorInfo);
41                rc = errorInfo.code;
42                break;
43    }
44    RfcDestroyFunction(functionContainer, NULL);
45    /* The following if-block handles both cases: when the connection broke
         down during RfcInvoke
46       as well as when we were unable to reconnect after an RFC_ABAP_MESSAGE
         or RFC_ABAP_RUNTIME_FAILURE.
47       That's why we didn't include the case in the switch-block.
48    */
49    if (rc == RFC_COMMUNICATION_FAILURE){
50          printfU(cU("A communication/network problem occurred: %s\n"),
                errorInfo.message);
51          printfU(cU("Please check the connection to %s and try again.\n"),
                attributes.sysId);
52          break;
53    }
54 }
```

Figure 4 (continued)

these two cases as well, avoiding the tripling of the error-handling code for network problems.

The check for resetting the ABAP session context is only necessary in the RFC_OK and RFC_ABAP_EXCEPTION cases (lines 07 and 21). In the other cases (RFC_ABAP_MESSAGE, RFC_ABAP_RUNTIME_FAILURE, and RFC_COMMUNICATION_FAILURE), the ABAP session context has already been destroyed, so there's no need to do it again.

To avoid memory leaks, you should release the functionContainer in any case, independent of the error code. For this reason that's done in line 44 outside the switch-block. The exception is that, if your application can reuse the container for further calls, a good place to destroy it would be the if-block for communication failures.

Tip!

Depending on the back-end release, some error information may not be available. In particular, for some of the ABAP message cases, SAP R/3 4.6C and R/3 4.6D have not yet returned the detail values (message type, ID, number, and v1–v4 parameters). Therefore, the corresponding fields in errorInfo remain empty. This is a shortcoming of the SAP R/3 kernel, so don't blame the SAP NetWeaver RFC library.

Let's proceed to the final step of our RFC client program now: how the results returned by the back end can be displayed to the user.

Comparing SAP NetWeaver RFC SDK to the classic RFC SDK

This article discusses three aspects of RFC programming:

- Metadata descriptions
- Data containers
- Invocation of ABAP function modules

When comparing these to the classic RFC SDK, observe the following differences and similarities.

Metadata descriptions

In the classic RFC SDK, the concept of metadata descriptions exists only in a rudimentary form. You can use the genh tool to generate header files for ABAP structures, but it only supports flat structures (ABAP type 1). Also, two functions, `RfcGetStructureInfoFlat()` and `RfcGetStructureInfoAsTable()`, can be considered forerunners of `RfcGetTypeDesc()`. However, the information these functions return isn't easy to use and works only for flat structures. You can't get the description of a complete function module in one step.

`RfcGetFunctionInfoAsTable()` works only with simple function modules. The classic RFC SDK user basically has to build up memory areas for his or her function modules manually from single pieces. To define the layout of single structures and tables, you can use the functions `RfcInstallStructure()` (plus genh information) and `RfcInstallStructure2()/RfcInstallUnicodeStructure()` (plus information from `RfcGetStructureInfoFlat()`) within their limitations.

Data containers

The concept of data containers doesn't exist at all in the classic RFC SDK. There, it is basically the application developer's responsibility to provide structures with the correct layout to hold the data. In the classic RFC SDK, you always have to provide values in the correct data type, whereas SAP NetWeaver RFC SDK has convenient automatic-conversion features. The only conversion available in the classic RFC SDK is from the BCD data type to STRING and vice versa using `RfcConvertBcdToChar()` and `RfcConvertCharToBcd()`.

Also, in the classic RFC SDK you have to assemble the list of function-module parameters manually, a tedious and error-prone task. In the server case, the programmer must send the response data to the back-end system, in contrast to SAP NetWeaver RFC SDK, which performs this task automatically.

Invocation of ABAP function modules

SAP NetWeaver RFC SDK client connectivity has retained a measure of similarity to the way it worked in the classic RFC SDK. Ignoring changes in how import and export parameters are exchanged with the RFC library, you can think of the functions shown in the table as being equivalent.

Continues on next page

Continued from previous page

Classic RFC SDK	SAP NetWeaver RFC SDK
RfcOpenEx()	RfcOpenConnection()
RfcCallReceiveEx()	RfcInvoke()
RfcCleanupContext()	RfcResetServerContext()
RfcClose()	RfcCloseConnection()

Besides the handling of import/export data, three other concepts have changed quite substantially:

- Error-handling has evolved to support all error types (as long as the back end does, too) while the classic RFC library was only able to handle the standard ABAP exceptions. Everything else was mapped to a SYSTEM_FAILURE, thereby losing the server context's detail information, such as the contents of the fields SY-MSGID, SY-MSGTY, SY-MSGNO, and so on.
- The way "RFC callbacks" are handled within an RFC client program has been simplified. We haven't covered callbacks yet, since there is a separate section dedicated to that topic in the third article.
- Experience has shown that trying to interrupt an ongoing RFC call is quite risky and seldom yields the desired effects. Before something like this can work satisfactorily, a lot needs to be changed on SAP R/3 kernel side. Therefore, the split into two separate RfcCallEx() and RfcReceiveEx() functions, as well as the function RfcCancel(), have been removed from the SAP NetWeaver RFC library.

Processing the output parameters

When `RfcInvoke()` ends successfully, the SAP NetWeaver RFC library has already read the back end's return parameters from the network connection and written them into the RFC_FUNCTION_HANDLE container. The program then uses a similar recursive mechanism to traverse through the output parameters (CHANGING, EXPORTING, and TABLES) via the function `printExports()`, which is symmetrical to the `fillImports()` function. Take a look at the helperFunctions.c file again.

As when setting the input values via `RfcSetString()`, you can use the convenient function `RfcGetString()` to convert all kinds of data types into a string representation. Otherwise, nothing much is new in this section.

One point deserves some extra attention, however: Before trying to execute the next function module, the program asks the user whether to reset the ABAP session context in the back-end system. A reset will clean up any leftover memory or stale information from the previous function call and, thus, make sure that there won't be any unwanted side effects on the next call. It is almost equivalent to closing and then reopening a fresh connection, but it has a much more efficient performance, because it saves the time that would be needed to establish a new network connection and handle an expensive logon procedure. (When SNC is involved, the logon handshake is quite expensive.) A reset is a must, especially when implementing a connection pool (used by different parts of the application) before returning the connection to the pool for others to use.

On the other hand, always resetting the connection after every call isn't necessarily a good idea either. Some families of function modules store important state information in the ABAP session memory, which the next function module in the sequence uses. If that memory is deleted, a follow-up function module that depends on it will get an error. Examples of this are function modules that use the keyword "EXPORT TO

MEMORY" or global parameters defined in the top include of the function group.

Another case in which an untimely reset can hurt is when a BAPI writes to the database (Update-BAPIs). These BAPI changes are only persisted in the database if the BAPI call is followed by a call to `BAPI_TRANSACTION_COMMIT` *within the same user session*. If a reset is done between the two `RfcInvoke()`s for the BAPI and the `BAPI_TRANSACTION_COMMIT`, then the temporary buffer that the BAPI created is discarded and not written to the database.

But note that there is no rule without an exception. There are some old 3.1I BAPIs that have their internal COMMIT WORK. They don't need a `BAPI_TRANSACTION_COMMIT`, so it's always useful to consult the documentation when using BAPIs.

By asking the user whether a connection reset should be done, the demoClient program can correctly call chains of "stateful" function modules, as well as Update-BAPIs.

Conclusion

SAP NetWeaver RFC SDK allows you to write RFC client programs quickly, even if the function modules involved are complex or use advanced features. This makes SAP NetWeaver RFC SDK usable not only for "static" approaches (where the function modules are hard-coded), but also for generic approaches (where the programs need to react dynamically).

The simplicity of the examples given here shows you that a lot of work is being done "under the hood" in the SAP NetWeaver RFC library (i.e., the library internally performs a lot of tasks with which the application developer was formerly burdened). In particular, the parsing and interpreting of complex data types, and the correct conversion of non-ASCII character data were difficult in the past. The enhanced capabilities for reacting to error conditions outside your control (i.e., inside the SAP system or ABAP code, or in the network layer) should make your programs more robust, and the possibility of getting more detailed error messages from the ABAP side should make the task of finding and fixing the cause of a problem easier.

In the second installment of our series of articles, we'll explore the intricacies and advanced features of the "opposite direction" of RFC communications: external programs that can be called from within an SAP system.

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.

