NetWeaver RFC SDK 7.50
Programming Guide

Release 7.50

Copyright

© 2018-2023 SAP SE or an SAP affiliate company. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP SE or an SAP affiliate company.

The information contained herein may be changed without prior notice. Some software products marketed by SAP SE and its distributors contain proprietary software components of other software vendors. National product specifications may vary.

These materials are provided by SAP SE or an SAP affiliate company for informational purposes only, without representation or warranty of any kind, and SAP or its affiliated companies shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP or SAP affiliate company products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

In particular, SAP SE or its affiliated companies have no obligation to pursue any course of business outlined in this document or any related presentation, or to develop or release any functionality mentioned therein. This document, or any related presentation, and SAP SE's or its affiliated companies' strategy and possible future developments, products, and/or platforms, directions, and functionality are all subject to change and may be changed by SAP SE or its affiliated companies at any time for any reason without notice. The information in this document is not a commitment, promise, or legal obligation to deliver any material, code, or functionality. All forward-looking statements are subject to various risks and uncertainties that could cause actual results to differ materially from expectations. Readers are cautioned not to place undue reliance on these forward-looking statements, and they should not be relied upon in making purchasing decisions.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP SE (or an SAP affiliate company) in Germany and other countries. All other product and service names mentioned are the trademarks of their respective companies.

See www.sap.com/copyright for additional trademark information and notices.

Icons in Body Text

Icon	Meaning
	Caution
	Example
	Note
	Recommendation
	Syntax

Additional icons are used in SAP Library documentation to help you identify different types of information at a glance. For more information, see *Help on Help* → *General Information Classes and Information Classes for Business Information Warehouse* on the first page of any version of *SAP Library*.

Typographic Conventions

Type Style	Description
Example text	Words or characters quoted from the screen. These include field names, screen titles, pushbuttons labels, menu names, menu paths, and menu options.
	Cross-references to other documentation.
`Example text`	Emphasized words or phrases in body text, graphic titles, and table titles.
EXAMPLE TEXT	Technical names of system objects. These include report names, program names, transaction codes, table names, and key concepts of a programming language when they are surrounded by body text, for example, SELECT and INCLUDE.
`Example text`	Output on the screen. This includes file and directory names and their paths, messages, names of variables and parameters, source text, and names of installation, upgrade and database tools.
`Example text`	Exact user entry. These are words or characters that you enter in the system exactly as they appear in the documentation.
`<Example text>`	Variable user entry. Angle brackets indicate that you replace these words and characters with appropriate entries to make entries in the system.
`EXAMPLE TEXT`	Keys on the keyboard, for example, `F2` or `ENTER`.

Contents

1 Content of the NetWeaver RFC SDK 5

2 Obtaining Metadata Descriptions 8

2.1 Automatic Metadata Lookup 9

2.2 Hardcoding Metadata Descriptions 14

2.3 Saving and Restoring Metadata from a File 20

3 Creating and Working with Data Containers 21

3.1 Creating and Destroying Data Containers 22

3.2 Accessing the Values of a Data Container 22

3.3 Data Flow in RFC Client Programs 24

3.4 Data Flow in RFC Server Programs 25

3.5 Working with Tables 25

4 RFC Client Programs 28

4.1 Managing Login Information 28

4.2 Synchronous RFC Client 34

4.3 Transactional RFC Client 39

4.4 Queued and Background RFC Client 45

4.5 WebSocket RFC Client 46

5 RFC Server Programs 47

5.1 Preparing a Server Program for Receiving RFC Requests 47

5.2 Registered RFC Server 55

5.3 Started RFC Server 69

5.4 WebSocket RFC Server 72

5.5 Transactional RFC Server 74

5.6 Queued and Background RFC Server 81

5.7 Error Handling on ABAP Side When Calling an External RFC Server 84

6 Performance Analysis of RFC Calls 86

6.1 Measuring RFC Client Connections 86

6.2 Measuring RFC Server Performance 87

6.3 Tips and Tricks 89

7 RFC Callback 90

7.1 RFC Callback in Client Programs 91

7.2 RFC Callback in Server Programs 92

8 Adding Supportability Features 94

1 Content of the NetWeaver RFC SDK

The NW RFC SDK is available in form of a zip file. See SAP Note 2573790 for instructions on how to download that archive. The archive is available for several different target platforms (note 2573790 provides a complete list of supported platforms) and includes the following files, libraries, and programs:

- Several C header files that you can use to compile your C/C++ programs against the RFC API.
- For platforms that have separate link libraries and runtime libraries (for example, Windows or IBM z/OS): link libraries that you use to link programs against the shared libraries.
- Shared libraries needed by your programs at runtime (and at link time on those platforms, that don't have separate link and runtime libraries).
- Sample programs to get you started.

Additionally, a zip file with Doxygen documentation describing each function, structure and enumeration of the RFC library in detail, is available for download from https://support.sap.com/nwrfcsdk.

An important source of information is also the file *sapnwrfc.ini* contained in the demo directory of the NW RFC SDK archive: it provides a complete list of all configuration parameters that a program can use with the NetWeaver RFC library.

The following files are platform independent:

Directory	File	Description
include	**sapnwrfc.h**	This header file contains all function, structure and *enum* declarations that make up the RFC API. The implementing library is *[lib]sapnwrfc*.
	sapucrfc.h	Header file containing platform-independent Unicode specifications and functions for processing Unicode data. The implementing library is *libsapucum*.
	sapdecf.h	Header file containing functions and typedefs for working with the ABAP DECFLOAT data types (DF16 and DF34). The implementing library is *libicudecnumber*. Since Patch Level 11, the libicudecnumber is linked statically and replaces the dynamically loaded library.
demo	**companyClient.c, sflightClient.c, sso2sample.c, stfcDeepTableServer.c, ThroughputSample.c**	Sample programs: a simple RFC client, a more complex client, a sample of how to use a ticket to log into the backend, an RFC server processing a nested table in the IMPORT/EXPORT parameters, and an example how to analyze performance using an RfcThrouput object.
	rfcexec.h, rfcexec.cpp	Source code of the *rfcexec* sample program that you can modify according to your needs.
	startrfc.h, startrfc.cpp	Source code of the *startrfc* sample program, that you can modify according to your needs.
	readme.txt	Some hints to additional info sources.
	sapnwrfc.ini	A sample *sapnwrfc.ini* file that contains detailed documentation of every logon and customizing parameter that a program can use with the NW RFC library. **Note:** the application can use almost every parameter in the *sapnwrfc.ini* configuration file as well as in an

		RFC_LOGON_PARAMETER array that is being passed into an RFC library function call (like `RfcOpenConnection()` or `RfcRegisterServer()`, and so on). Exceptions are a few “global” parameters, which are clearly marked as such.
doc	**licenses.txt**	This file informs about the Open Source libraries shipped with NW RFC SDK, and which licenses those libraries use.
	release_notes.txt	Accumulated information about enhancements and fixes in the patch levels up to the one contained in the archive.

The following files are platform-dependent and may vary from platform to platform (important ones are listed):

Directory	File	Description
bin	**rfcexec.exe** (on Windows) **rfcexec** (on the remaining platforms)	Executable file needed by some ALE scenarios for sending IDocs via file port to EDI subsystems. The source code of this program is contained in the demo directory, so you can modify it according to your needs, add further security checks, and so on.
	startrfc.exe (on Windows) **startrfc** (on the remaining platforms)	Executable file needed by some ALE scenarios for sending IDocs via file port into an SAP System. The source code of this program is contained in the demo directory, so you can modify it according to your needs, add further security checks, and so on.
lib	**icudt50.dll, icuin50.dll, icuuc50.dll** (on Windows) **libicudata.so.50, libicui18n.so.50, libicuuc.so.50** (on Linux, zLinux and Solaris) **libicudata50.a, libicui18n50.a, libicuuc50.a** (on AS/400 and AIX) **libicudata.50.dylib, libicui18n.50.dylib, libicuuc.50.dylib** (on MacOS X) **libicudata.sl.50, libicui18n.sl.50, libicuuc.sl.50** (on HP-UX) **libicudata50.1.so, libicui18n50.1.so, libicuuc50.1.so** (on z/OS)	IBM’s open source library “International Components for Unicode”. Used internally by the RFC SDK. Do not use directly.
	sapnwrfc.dll (Windows only) **libsapnwrfc.dylib (MacOS only)**	Shared library exporting the RFC API.

	libsapnwrfc.so (other platforms)	
	sapnwrfc.lib (Windows only) libsapnwrfc.x libsapnwrfc.a (z/OS only)	Import library for the above shared library needed on these platforms as linker input.
	libsapucum.dll (Windows only) libsapucum.dylib (MacOS only) libsapucum.so (other platforms)	Shared library exporting the Unicode API (some string manipulation functions needed for handling Unicode string data).
	libsapucum.lib (Windows only) libsapucum.x libsapucum.a (z/OS only)	Import library for the above shared library needed on these platforms as linker input.

2 Obtaining Metadata Descriptions

The main task of the NetWeaver RFC SDK is to exchange data between the ABAP world and the C/C++ world. In order to do that, both sides must have a consistent agreement about how a certain piece of data looks like. For example, a certain field is of type “character” (CHAR in ABAP, SAP_UC in C) and has the length “10”. Or a certain variable may be a structured type with 17 subfields, some of which are character fields of various lengths, others are integer fields of length 4 bytes and the final one may be an 8-byte floating point number. This means we are assembling “data about how data looks like”, which is called *metadata*.

In the SAP system, all metadata is defined centrally in the ABAP DDIC (Data Dictionary). In order to interpret the ABAP data correctly in a C/C++ program, a way is needed to pull these metadata descriptions about fields, structures, tables and function modules from the ABAP DDIC into the C/C++ program.

Another important point to consider is that the data types available in the ABAP language differ slightly from the data types available in the C programming language, so a mapping is required between the two corresponding types from each environment.

The NW RFC API supports all elementary and complex ABAP data types. The following list shows the supported data types and their specifications:

Table 2-A: ABAP Data Types Supported by the RFC API

ABAP type	C typedef	Length (in Bytes)	Description
c	RFC_CHAR	1-65535	Characters, blank padded at the end
n	RFC_NUM	1-65535	Digits, fixed size, leading '0' padded.
x	RFC_BYTE	1-65535	Binary data
p	RFC_BCD	1-16	BCD numbers (Binary Coded Decimals)
i	RFC_INT	4	Integer
b	RFC_INT1	1	1-byte integer, not directly supported by ABAP
s	RFC_INT2	2	2-byte integer, not directly supported by ABAP
8	RFC_INT8	8	8-byte integer
p	RFC_UTCLONG		Timestamp with high precision - 8 bytes
f	RFC_FLOAT	8	Floating point
d	RFC_DATE	8 or 16	Date ("YYYYMMDD")
t	RFC_TIME	6 or 12	Time ("HHMMSS")
a	RFC_DECF16	8	Decimal floating point 8 bytes (IEEE 754r)

e	RFC_DECF34	16	Decimal floating point 16 bytes (IEEE 754r)
g	RFC_CHAR*		Variable-length, zero terminated string
y	RFC_BYTE*		Variable-length raw string, length in bytes

The necessary typedefs for working with these types in a C program are defined in the *sapnwrfc.h* header file.

Let's now look at the two main options for getting the necessary metadata descriptions into your program.

2.1 Automatic Metadata Lookup

The most convenient way of obtaining metadata descriptions, is to use the NW RFC library's DDIC lookup functions. With just a few lines of code it is possible to retrieve the metadata for all the function modules and their structures and tables that a program may need during its lifetime. Here are the steps that the RFC program needs to perform for this:

1. Open an RFC client connection. (The guide will get to this in more detail in chapter 4 *RFC Client Programs*.)
2. Use one of the DDIC lookup functions to let the NW RFC library read all the required field, structure/table and function module descriptions from the ABAP DDIC over this open RFC connection.
3. Close the connection again, if it is no longer needed (for example, if the program is an RFC server program).
4. Use the metadata objects for creating the corresponding memory areas which are going to hold the data. (We will call these *data containers* in the following.) This will be explained in more detail in chapter 3 *Creating and Working with Data Containers*.

Step 2 will now be elaborated in more detail. The first important point to know is that the NW RFC library automatically keeps a so-called *DDIC cache*. Every function description and structure description that you look up from the DDIC, will be put into this cache, so when you need that description a second time, the library will no longer make a call to the backend, but return the already cached information. In long-running programs, this usually results in a big performance improvement.

This cache, however, uses the SAP system ID as a key, meaning: if your program executes the same function calls against two (or more) different SAP systems, then the description for each function module is kept in memory twice (or more). The reason for this is, that the same function module may look different in two different backend systems. For example, if one backend is an older 4.6C system, while the other one already has release 7.40, then a certain function module parameter may be of type CHAR20 in the older release, while it has been enlarged to CHAR30 in the newer one. Or a certain structure may have additional fields in the newer system, which did not yet exist in the older one. So, if a structure definition that was retrieved from system A, is used for a function call to system B, the data may get truncated or corrupted during the RFC call.

In most cases, the application will probably need just one function module description, and this can be achieved with a few lines of code like this:

```
RFC_CONNECTION_PARAMETER loginParams[6];
RFC_ERROR_INFO errorInfo;
RFC_CONNECTION_HANDLE connection;
RFC_FUNCTION_DESC_HANDLE bapiCompanyDesc;

// Set login parameters here

connection = RfcOpenConnection(loginParams, 6, &errorInfo);
// Error handling omitted

bapiCompanyDesc = RfcGetFunctionDesc(connection,
                    cU("BAPI_COMPANY_GETDETAIL"), &errorInfo);
// Error handling omitted
```

The API `RfcGetFunctionDesc()` first looks into the cache corresponding to the system ID taken from the connection handle. If the function description has not yet been cached previously, the API makes the necessary DDIC lookups over the provided connection and afterwards puts the complete function description into the cache.

An alternative design to the above way of obtaining the metadata when it's first needed, is to fill the cache with all required metadata somewhere in the start-up/initialization routine of the application. A code snippet for this approach would look like this:

```
RFC_CONNECTION_PARAMETER loginParams[7];
RFC_ERROR_INFO errorInfo;
RFC_CONNECTION_HANDLE connection;
RFC_METADATA_QUERY_RESULT_HANDLE queryResult;

// Set login parameters here
// Note that the parameter USE_REPOSITORY_ROUNDTRIP_OPTIMIZATION = 1 needs
// to be set and that the backend must have at least the Support Package
// level listed in SAP note 1456826.
connection = RfcOpenConnection(loginParams, 7, &errorInfo);
// Error handling omitted


queryResult = RfcCreateMetadataQueryResult(&errorInfo);
// Error handling omitted
SAP_UC* functionNames[5] = {
             cU("BAPI_COMPANY_GETLIST"),
             cU("BAPI_COMPANY_GETDETAIL"),
             cU("BAPI_CUSTOMER_GETLIST"),
             cU("BAPI_CUSTOMER_GETDETAIL"),
             cU("BAPI_DOES_NOT_EXIST")
};
RfcMetadataBatchQuery(connection, functionNames, 5, NULL, 0,
                       NULL, 0, queryResult, &errorInfo);
if (errorInfo.code != RFC_OK) {
   unsigned lookupSucceeded, lookupFailed;

   printfU(cU("DDIC Lookup failed for the following functions:\n"));
   RfcDescribeMetadataQueryResult(queryResult,
                RFC_METADATA_FUNCTION, &lookupSucceeded,
                &lookupFailed, &errorInfo);
   for (unsigned i=0; i<lookupFailed; ++i) {
      RFC_METADATA_QUERY_RESULT_ENTRY resultEntry;

      RfcGetMetadataQueryFailedEntry(queryResult,
            RFC_METADATA_FUNCTION, i, &resultEntry, &errorInfo);
      printfU(cU("%s: %s\n"), resultEntry.name,  resultEntry.errorMessage);
```

```
    }
}

RfcDestroyMetadataQueryResult(queryResult, NULL);
```

Once the metadata descriptions are in the cache, the program can use them from anywhere with a call like

```
RFC_FUNCTION_DESC_HANDLE bapiCompanyGetListDesc =
    RfcGetCachedFunctionDesc(cU("ABC"), cU("BAPI_COMPANY_GETLIST"),
                             &errorInfo);
```

where “ABC” is the system ID of the backend system from which you looked up the metadata.

The advantages of this approach are:

- The NW RFC library can perform the necessary DDIC lookups in a much more “compact” form. Especially when using the connection parameter `USE_REPOSITORY_ROUNDTRIP_OPTIMIZATION` (see below), it may be possible that the metadata for all function modules can be collected from the DDIC in one single roundtrip to the backend, instead of possibly hundreds of roundtrips.
- The part of the coding that deals with metadata does not need to have access to an open RFC connection. Just knowing the system ID of the ABAP system it works with is enough.

One disadvantage, however, is that the start-up phase of the application can take quite a bit longer compared to the “just-in-time metadata retrieval approach”.

In the previous couple of paragraphs, we discussed the API functions available for the metadata lookup of *remote-enabled function modules* (RFMs). These APIs automatically also lookup and cache the metadata descriptions of all structures and tables used by these function modules, so it should almost never be necessary to make extra DDIC lookups for structures and tables. A handle to a structure description can be obtained from the “parent” (the RFM description) using the APIs `RfcGetParameterDescByIndex()` or `RfcGetParameterDescByName()`. For example, if you need to know the layout of the table `COMPANY_LIST` used by `BAPI_COMPANY_GETLIST`, you can proceed as follows:

```
RFC_FUNCTION_DESC_HANDLE bapiCompanyGetListDesc; // Obtained as
                                                 // outlined above
RFC_PARAMETER_DESC parameterDescription;
RFC_TYPE_DESC_HANDLE companyListDesc;

RfcGetParameterDescByName(bapiCompanyGetListDesc,
          cU("COMPANY_LIST"), &parameterDescription, &errorInfo);

if (errorInfo.code == RFC_OK) {
    companyListDesc = parameterDescription.typeDescHandle;
}
```

Or, if you happen to know the DDIC name of the structure, from which the function module parameter `COMPANY_LIST` is derived, you can simply get it from the cache via

```
RFC_TYPE_DESC_HANDLE companyListDesc =
       RfcGetCachedTypeDesc(cU("ABC"), cU("BAPI0014_1"), &errorInfo);
```

And a third way of obtaining the description of a certain structure is by getting it from a structure or table, which has been created from that description. Data containers, like structures and tables, will be discussed in the next chapter, so you don’t have to fully understand the following yet, but for completeness, here is a code snippet illustrating that way as well. Assume that the program is an RFC server, it received a call for

BAPI_COMPANY_GETLIST, and the NW RFC library provided to the call handler (among other things) the table COMPANY_LIST:

```
RFC_TABLE_HANDLE companyList = ...; // Got it from ABAP system
RFC_TYPE_DESC_HANDLE companyListDesc =
       RfcDescribeType(companyList, &errorInfo);
```

So in any case, even though it should never be necessary to perform a DDIC lookup for a structure or table (because the program can usually get it from the description of the function modules you use or from the actual data container), the NW RFC library nevertheless provides DDIC lookup functions for reading structure descriptions (RFC_TYPE_DESC_HANDLE) as well as ABAP class descriptions (RFC_CLASS_DESC_HANDLE) from the ABAP DDIC.

Here is an overview of the available DDIC lookup functions:

Table 2-B: DDIC Functions

	Function Modules	Structures/Tables	ABAP Classes
RFC Lib Handle for holding the information	RFC_FUNCTION_DESC_HANDLE	RFC_TYPE_DESC_HANDLE	RFC_CLASS_DESC_HANDLE
Retrieval functions	RfcGetFunctionDesc() RfcGetCachedFunctionDesc() RfcDescribeFunction()	RfcGetTypeDesc() RfcGetCachedTypeDesc() RfcDescribeType()	RfcGetClassDesc() RfcGetCachedClassDesc() RfcDescribeAbapObject()
Construction function	RfcCreateFunctionDesc()	RfcCreateTypeDesc()	RfcCreateClassDesc()
Deletion function	RfcDestroyFunctionDesc()	RfcDestroyTypeDesc()	RfcDestroyClassDesc()
Members containing detail information	RFC_PARAMETER_DESC RFC_EXCEPTION_DESC	RFC_FIELD_DESC	RFC_CLASS_ATTRIBUTE_DESC
Functions for getting the details	RfcGetFunctionName() RfcGetParameterCount() RfcGetParameterDescByIndex() RfcGetParameterDescByName() RfcGetExceptionCount() RfcGetExceptionDescByIndex() RfcGetExceptionDescByName()	RfcGetTypeName() RfcGetTypeLength() RfcGetFieldCount() RfcGetFieldDescByIndex() RfcGetFieldDescByName()	RfcGetClassName() RfcGetClassAttributesCount() RfcGetClassAttributeDescByIndex() RfcGetClassAttributeDescByName() RfcGetParentClassesCount() RfcGetParentClassByIndex() RfcGetImplementedInterfacesCount() RfcGetImplementedInterfaceByIndex()
Functions for setting the details	RfcAddException() RfcAddParameter()	RfcAddTypeField() RfcSetTypeLength()	RfcAddClassAttribute() RfcAddParentClass() RfcAddImplementedInterface()

Important Note: In the above overview, you have noticed the functions for creation and deletion of metadata descriptions and may wonder, when to use them. The answer is: when using the automatic lookup mechanism described in the current chapter, you do not and must not use these functions. In particular, never call the destroy function on a description object you obtained from one of the retrieval functions. This would leave an invalid pointer as leftover in the DDIC cache and certainly lead to a crash the next time the application fetches this description object from the cache and tries to use it. These functions are meant for those cases, where

it is not possible to use the DDIC lookup and hence the application has to create its own metadata descriptions. See the next chapter *Hardcoding Metadata Descriptions* for details.

⚠ Metadata descriptions obtained via the retrieval functions are immutable, so you cannot add further members to them using the “setter functions”.

Another point that may have caught your attention and got you wondering is, there are functions that deal with *ABAP classes*. What are these for?

Background: In ABAP system release 7.11 a new feature was introduced that allowed remote-enabled function modules to throw so called class-based exceptions, i.e. any ABAP OO object deriving from the root class `CX_ROOT`. In ABAP system release 7.20, this feature was fully functional and could also be used by external RFC programs in C/C++, Java and Microsoft .NET (i.e. the connectors).

An external program acting as RFC client could catch an exception object thrown by an ABAP function module, an external program acting as RFC server could throw such an exception object, and the ABAP side could catch it. This allowed much more error details to be transferred between the two communication partners than the classic (non-OO) ABAP Exception thrown by the ABAP statements “`RAISE …`” and “`MESSAGE … RAISING …`”. However, then people started realizing, that this new mechanism was incompatible to the existing code base, especially in the ABAP-to-ABAP communication, and therefore this feature was deleted again in ABAP system release 7.30.

So, if you still have an old 7.20 system and RFMs that throw class-based exceptions, you can still handle this correctly in your C/C++ programs, but sooner or later the backend will be upgraded to an up-to-date release, and then your ABAP coding as well as the C/C++ side has to be changed back to classic exceptions. Therefore, in new projects it is probably best to just ignore this feature.

Let’s conclude this chapter with a word about performance. Programs that need to use many different function modules and possibly hundreds or thousands of different structures and tables, may take quite a while to load all these structure descriptions into the cache. This is because there is one roundtrip to the backend required for each function module that is used and one additional roundtrip for each single structure/table used by this function module. All this can potentially add up to hundreds or even thousands of roundtrips.

To speed this up significantly, a new way of reading the DDIC was introduced in SAP_BASIS release 7.00. (SAP note 1456826 gives some details about the minimum support package level required for this feature to be available, and it also contains an ABAP correction instruction that you can implement manually if you cannot upgrade to that support package level, but still would like to use this feature.)

Once your backend system fulfils the necessary requirements, you can set the connection parameter
`USE_REPOSITORY_ROUNDTRIP_OPTIMIZATION = 1`
in the login parameters of the connection that is used for the DDIC lookups, and then the NW RFC library will fetch the complete metadata of a function module (the function description as well as all structure descriptions required by this function module) in one single roundtrip to the backend.

If the program is using the API `RfcMetadataBatchQuery()`,the performance gain will be even bigger, as it is possible to fetch the descriptions for all function modules and their structures in one single roundtrip. The DDIC data is even compressed during communication, in contrast to the old lookup mechanism which transferred the data uncompressed, so the network bandwidth is also reduced.

The new DDIC lookup mechanism requires the backend user used for login to have additional authorizations: see SAP note 460089 for details about user authorizations required for RFC communication.

2.2 Hardcoding Metadata Descriptions

The mechanism of obtaining metadata descriptions explained in the previous section should be the way to go for 99% of all projects, and if it can be used, it is strongly recommended to do so. However, sometimes it is not possible to use the lookup mechanism, and for that reason there is an alternative way of getting the metadata descriptions: the application code just cobbles them together itself.

Below there are three valid reasons, why an application should do this:

1. It is an end-user facing program that needs to be highly responsive, and after some serious performance analysis you find that the short delay in the startup time caused by the DDIC lookups is unacceptable.
2. It is an RFC server program that must be able to run without having access to user credentials. (So, the program is not able to logon to the backend system and perform the DDIC lookups.)
3. It is an RFC server program that wants to export “ABAP” function modules, which do not exist in the backend. (In the backend it is possible to write the ABAP code for calling a function module that does not even exist in that backend system. If the receiving server program knows that function module, and if the data definitions on both sides match, it will work fine.) An example of this approach is described in chapter 5 *RFC Server Programs*.

Before you embark on this journey, you should also be aware of the disadvantages and risks of this approach:

1. Setting up structure descriptions is an error-prone and tedious task, and it is easy to get it wrong, because subtle details about alignment bytes for different data types on different platforms must be considered.
2. With hardcoded metadata in the C program you have to remember to adjust and recompile the C program, if at a later point of time the backend system is upgraded to a new release, or some other action is performed that causes certain parameters to be changed (for example, fields are enlarged or new fields are added to a structure). If you forget this, the data definitions used on the two sides would get out of sync, causing data truncation or data corruption.
3. If there are errors or missing updates in the metadata descriptions of the C program, data loss or data corruption may be the consequence. Problems like these are hard to debug and to find.

After these words of warning, let’s now jump right into it and write the necessary code for constructing the metadata of the function module `STFC_STRUCTURE`, which exists in every ABAP system. Looking up this function in the function builder, we find that it has four parameters:

Parameter	Direction	Type
`IMPORTSTRUCT`	`IMPORTING`	`RFCTEST`
`ECHOSTRUCT`	`EXPORTING`	`RFCTEST`
`RESPTEXT`	`EXPORTING`	`SY-LISEL`
`RFCTABLE`	`TABLES`	`RFCTEST`

`SY-LISEL` is a simple field of type CHAR255, but `RFCTEST` is a structure with 12 fields, which looks as follows in the ABAP Data Dictionary (transaction SE12):

Structure: RFCTEST Active

Short Description: RFC: Test structure

Attributes Components Input Help/Check Currency/Quantity Fields

Built-In Type 1 / 12

Component	Typing Method	Component Type	Data Type	Length		Coordinate	Short Description
RFCFLOAT	Type	RFCFLOAT	FLTP	16	16	0	Float field
RFCCHAR1	Type	RFCCHAR1	CHAR	1	0	0	Character field of length 1
RFCINT2	Type	RFCINT2	INT2	5	0	0	INT2 field
RFCINT1	Type	RFCINT1	INT1	3	0	0	INT1 field
RFCCHAR4	Type	RFCCHAR4	CHAR	4	0	0	Character field of length 4
RFCINT4	Type	RFCINT4	INT4	10	0	0	INT4 field
RFCHEX3	Type	RFCHEX3	RAW	3	0	0	HEX field of length 3
RFCCHAR2	Type	RFCCHAR2	CHAR	2	0	0	Character field of length 2
RFCTIME	Type	RFCTIME	TIMS	6	0	0	Validity Period
RFCDATE	Type	RFCDATE	DATS	8	0	0	Date field
RFCDATA1	Type	CHAR50	CHAR	50	0	0	Comment
RFCDATA2	Type	CHAR50	CHAR	50	0	0	Comment

So, let's first create an `RFC_TYPE_DESC_HANDLE` that describes this structure. For this we need the type description handle itself and then 12 field descriptions that have to be added to the type description. A field description is a struct with the following members, which you have to fill with the correct values:

Table 2-C: Field Description Members

`RFC_ABAP_NAME name;`	Field name, null-terminated string
`RFCTYPE type;`	Field data type
`unsigned nucLength;`	Field length in bytes in a 1-byte-per-SAP_CHAR system (a so-called non-Unicode System)
`unsigned nucOffset;`	Field offset in bytes in a 1-byte-per-SAP_CHAR system (a so-called non-Unicode System)
`unsigned ucLength;`	Field length in bytes in a 2-byte-per-SAP_CHAR system (a so-called Unicode System)
`unsigned ucOffset;`	Field offset in bytes in a 2-byte-per-SAP_CHAR system (a so-called Unicode System)
`unsigned decimals;`	If the field is of type "packed number" (BCD), this member gives the number of decimals.
`RFC_TYPE_DESC_HANDLE typeDescHandle;`	Pointer to an RFC_STRUCTURE_DESC structure for the nested sub-type if the "type" field above is set to RFCTYPE_STRUCTURE or RFCTYPE_TABLE
`void* extendedDescription;`	Not used by the NW RFC library. This parameter can be used by applications that want to store additional information in the repository (like F4 help values).

"`name`" and "`type`" can be taken directly from the Data Dictionary definition. (The *enum* `RFCTYPE` is defined in *sapnwrfc.h*. The *enum* values correspond one-to-one to the typedefs

listed in Table 2-A: *ABAP Data Types Supported by the RFC API* on page 8, so it should be obvious, how to map the ABAP data type to the corresponding *enum* value.)

"`nucLength`" and "`ucLength`" for character-like data types can be taken from the "Length" column in the Data Dictionary: just take that value for the non-Unicode length and multiply it by two for the Unicode length. However, for other data types the "Length" column is of no use to us, because it gives the required "display length" on a screen, not the "byte length" of the data type… (For example, you'll notice that the length for the INT2 field (a 2-byte integer) is given as 5 instead of 2. This is because the maximum number that can be stored in a 2-byte signed integer is 32767, which has 5 decimal digits.)

But for other data types you can simply take the length from the table on page 8. And if a field is of type STRING, XSTRING, structure or table, then it is referenced by a pointer-like mechanism, and the length must be given as 8. ("Flat" sub-structures, i.e. structures with a fixed pre-calculable length, can also be inlined into the parent structure. Then things could become a bit complex.)

"`decimals`" can also be taken from Data Dictionary. We are not using "`extendedDescription`" in this example, and `RFCTEST` has only scalar fields, no sub-structures, so "`typeDescHandle`" can be NULL as well. That leaves only "`nucOffset`" and "`ucOffset`" to consider.

Unfortunately, we cannot simply take "*offset + length of the previous field*" for these, because on all computer architectures structure members are aligned on addresses which are divisible by their length. So, an INT4 always has to start at an address divisible by 4, an 8-byte floating point on an address divisible by 8. This also means that in the non-Unicode metadata, character-like fields have no restrictions regarding their offsets (as the byte-length of a CHAR is 1 in that case), while in the Unicode metadata, these fields must have an offset divisible by 2 (byte-length of a CHAR is 2). The rule of thumb for calculating offsets is:

```
offset[0] = 0
offset[n] = offset[n-1] + length[n-1] rounded up to the next
            number divisible by datatypeLength[n]
```

Doing that for the structure `RFCTEST`, we get the following values:

Table 2-D: Metadata of RFCTEST

name	type	nucLength	nucOffset	ucLength	ucOffset
RFCFLOAT	`RFCTYPE_FLOAT`	8	0	8	0
RFCCHAR1	`RFCTYPE_CHAR`	1	8	2	8
RFCINT2	`RFCTYPE_INT2`	2	10	2	10
RFCINT1	`RFCTYPE_INT1`	1	12	1	12
RFCCHAR4	`RFCTYPE_CHAR`	4	13	8	14
RFCINT4	`RFCTYPE_INT`	4	20	4	24
RFCHEX3	`RFCTYPE_BYTE`	3	24	3	28
RFCCHAR2	`RFCTYPE_CHAR`	2	27	4	32
RFCTIME	`RFCTYPE_TIME`	6	29	12	36
RFCDATE	`RFCTYPE_DATE`	8	35	16	48
RFCDATA1	`RFCTYPE_CHAR`	50	43	100	64
RFCDATA2	`RFCTYPE_CHAR`	50	93	100	164

The remaining three fields (`decimals, typeDescHandle` and `extendedDescription`) can all be null in this example. (The `decimals` value for the FLOAT field is given as 16 by the ABAP Data Dictionary, but in C a floating-point value (as the name implies) actually does not have a

fixed number of decimals. Also, for the RFC layer the `decimals` value for `RFCTYPE_FLOAT` does not matter. The RFC layer needs the `decimals` value only for BCD numbers ("binary coded decimal", `RFCTYPE_BCD`, ABAP type "p" or DDIC type DEC).)

Now let's put all this together into a few lines of code:

```
RFC_ERROR_INFO errorInfo;

// Create the type description
RFC_TYPE_DESC_HANDLE rfctestDesc =
                RfcCreateTypeDesc(cU("RFCTEST"), &errorInfo);
// Do some error handling in case we have an out-of-memory…

// Define the field descriptions
RFC_FIELD_DESC fields[] = {
   { iU("RFCFLOAT"), RFCTYPE_FLOAT, 8,   0,   8,   0, 0, 0, 0},
   { iU("RFCCHAR1"), RFCTYPE_CHAR,  1,   8,   2,   8, 0, 0, 0},
   { iU("RFCINT2"),  RFCTYPE_INT2,  2,  10,   2,  10, 0, 0, 0},
   { iU("RFCINT1"),  RFCTYPE_INT1,  1,  12,   1,  12, 0, 0, 0},
   { iU("RFCCHAR4"), RFCTYPE_CHAR,  4,  13,   8,  14, 0, 0, 0},
   { iU("RFCINT4"),  RFCTYPE_INT,   4,  20,   4,  24, 0, 0, 0},
   { iU("RFCHEX3"),  RFCTYPE_BYTE,  3,  24,   3,  28, 0, 0, 0},
   { iU("RFCCHAR2"), RFCTYPE_CHAR,  2,  27,   4,  32, 0, 0, 0},
   { iU("RFCTIME"),  RFCTYPE_TIME,  6,  29,  12,  36, 0, 0, 0},
   { iU("RFCDATE"),  RFCTYPE_DATE,  8,  35,  16,  48, 0, 0, 0},
   { iU("RFCDATA1"), RFCTYPE_CHAR, 50,  43, 100,  64, 0, 0, 0},
   { iU("RFCDATA2"), RFCTYPE_CHAR, 50,  93, 100, 164, 0, 0, 0}
};

// Add the field descriptions to the type
for (int i=0; i<12; ++i)
   RfcAddTypeField(rfctestDesc, &fields[i], &errorInfo);


// Set total length of the type. (This locks the handle, so it can
// no longer be modified.)
RfcSetTypeLength(rfctestDesc, 144, 264, &errorInfo);
```

One detail to note is: the total length of a structure has to be aligned for the first field of the structure. Why is that? In case that a structure description is used in a table, the next line of the table follows immediately at the end of the previous line in the memory layout. But the first field of the second (third, etc.) line also has to be aligned according to the requirements of its data type. In this case, the first field is of type `FLOAT`, and therefore it (and consequently the entire line) must be aligned at an address divisible by 8. This is the reason, why in the last line of the above code sample, the non-Unicode total length of the structure is set to 144 (=18x8) instead of the expected 143 (=50+93).

This defines the structure `RFCTEST`, which is used several times in the function module parameters of `STFC_STRUCTURE`. Now we are ready to define that function module in pretty much the same way we defined the structure above: create a function description handle and then add the parameter descriptions to it. (If the function module in question would throw ABAP Exceptions (see the "Exceptions" tab in Function Builder), we would also have to add exception descriptions to the function description, but `STFC_STRUCTURE` does not have any Exceptions, so we can omit that step in this example.) A parameter description looks pretty much like a field description, with the exception that some attributes (the offsets) are not needed and instead some other attributes (is it importing or exporting? does it have a default value?) are now present:

Table 2-E: Parameter Description Members

`RFC_ABAP_NAME name;`	Parameter name, null-terminated string
`RFCTYPE type;`	Parameter data type
`RFC_DIRECTION direction;`	Specifies whether the parameter is an input, output or bi-directional parameter
`unsigned nucLength;`	Parameter length in bytes in a 1-byte-per-SAP_CHAR system (a so-called non-Unicode System)
`unsigned ucLength;`	Parameter length in bytes in a 2-byte-per-SAP_CHAR system (a so-called Unicode System)
`unsigned decimals;`	If the parameter is of type "packed number" (BCD), this member gives the number of decimals.
`RFC_TYPE_DESC_HANDLE typeDescHandle;`	Handle to the structure definition in case this parameter is a structure or table
`RFC_PARAMETER_DEFVALUE defaultValue;`	Default value as defined in function builder
`RFC_PARAMETER_TEXT parameterText;`	Description text of the parameter as defined in function builder. Null-terminated string.
`RFC_BYTE optional;`	Specifies whether this parameter is defined as optional in function builder. 1 is optional, 0 non-optional.
`void* extendedDescription;`	Not used by the NW RFC library. This parameter can be used by applications that want to store additional information in the repository (like F4 help values).

Determining these values for the four parameters of `STFC_STRUCTURE`, we find that none of the parameters has decimals or a default value and that none of them are optional. Also, we just skip the parameter text, as this is used only for documentation. Omitting these four columns, we arrive at the following table:

name	type	direction	nucLength	ucLength	typeDescHandle
IMPORTSTRUCT	`RFCTYPE_STRUCTURE`	`RFC_IMPORT`	144	264	`rfctestDesc`
ECHOSTRUCT	`RFCTYPE_STRUCTURE`	`RFC_EXPORT`	144	264	`rfctestDesc`
RESPTEXT	`RFCTYPE_CHAR`	`RFC_EXPORT`	255	510	NULL
RFCTABLE	`RFCTYPE_TABLE`	`RFC_TABLES`	144	264	`rfctestDesc`

The necessary coding for putting this together now looks very similar to what was done for creating the type description of the RFCTEST structure:

```
// Create the function description
RFC_FUNCTION_DESC_HANDLE stfcStructureDesc =
          RfcCreateFunctionDesc(cU("STFC_STRUCTURE"), &errorInfo);
// Do some error handling in case we have an out-of-memory…


// Define the parameter descriptions
RFC_PARAMETER_DESC parameters[] = {
   {iU("IMPORTSTRUCT"), RFCTYPE_STRUCTURE, RFC_IMPORT,
      144, 264, 0, rfctestDesc, iU(""),iU(""), 0, 0},
   {iU("ECHOSTRUCT"), RFCTYPE_STRUCTURE, RFC_EXPORT,
      144, 264, 0, rfctestDesc, iU(""),iU(""), 0, 0},
   {iU("RESPTEXT"), RFCTYPE_CHAR, RFC_EXPORT,
      255, 510, 0, 0, iU(""),iU(""), 0, 0},
   {iU("RFCTABLE"), RFCTYPE_TABLE, RFC_TABLES,
      144, 264, 0, rfctestDesc, iU(""),iU(""), 0, 0}
};


// Add the parameter descriptions to the function
for (int i=0; i<4; ++i)
   RfcAddParameter(stfcStructureDesc, &parameters[i], &errorInfo);
```

The function description for STFC_STRUCTURE is now ready to use the same way as a function description obtained by one of the lookup functions described in the previous chapter would be.

> Tip: if the structures and function modules your RFC program is going to use, are defined in the ABAP DDIC of your backend system, you can use the following trick to ease the task of calculating all the necessary field lengths and offsets and to ensure that all your numbers are correct: write a small program that logs into the backend, looks up the structures and functions you need and prints all the values to the console. Then copy these values into the hardcoded metadata of the productive program, which can then run without needing metadata lookups. For example (error handling omitted):

```
// Login to backend system
RFC_CONNECTION_HANDLE conn = RfcOpenConnection(loginParams,
                                numParams, &errorInfo);

// Perform DDIC lookup
RFC_TYPE_DESC_HANDLE rfctestDesc = RfcGetTypeDesc(conn,
                             cU("RFCTEST"), &errorInfo);
RfcCloseConnection(conn, NULL);

// Print results
RFC_FIELD_DESC fd;
unsigned numFields = 0;

RfcGetFieldCount(rfctestDesc, &numFields, &errorInfo);
for (unsigned i=0; i<numFields; ++i) {
   RfcGetFieldDescByIndex(rfctestDesc, i, &fd, &errorInfo);
   printfU(cU("%s, %s, %d, %d, %d, %d\n"), fd.name,
          RfcGetTypeAsString(fd.type), fd.nucLength,
          fd.nucOffset, fd.ucLength, fd.ucOffset);
}
```

2.3 Saving and Restoring Metadata from a File

Since NW RFC SDK 7.50 patch level 3 a third way of providing the necessary metadata to an application is available. This lies somewhere between the two extremes of looking up the metadata at runtime and completely hard-wiring them into the program code: you can generate a file with the necessary function and structure descriptions and then load that file into your program at startup time. This has a similar performance gain as hard-coding the metadata, but it does not come with the penalty that you have to modify and recompile the program, whenever a structure or function module gets modified in the ABAP backend: you can just regenerate the file containing the metadata and replace it in the installation directory of the application.

Like in the example above, you would write a small helper tool that logs into the backend system and loads the metadata for all function modules needed by the productive application code. For example, by making a call to `RfcMetadataBatchQuery()`. Afterwards just save the entire repository into a file via

```
RfcSaveRepository(SAP_UC const * repositoryID,
          FILE* const targetStream, RFC_ERROR_INFO* errorInfo);
```

Here, `repositoryID` would be the system ID of the backend from which you have loaded the metadata, and `targetStream` is a handle to the file, to which you want the metadata to be written.

In the productive code you would just need to add one line at the beginning of the initialization routine of the application, which loads the metadata from that file back into memory:

```
RfcLoadRepository(SAP_UC const * repositoryID,
          FILE* const targetStream, RFC_ERROR_INFO* errorInfo);
```

and afterwards whenever business logic needs a function description or structure description, it can get it with API calls like `RfcGetCachedFunctionDesc()` and `RfcGetCachedTypeDesc()`, passing the same `repositoryID` that was used for loading the repository into memory.

When loading a repository via `RfcLoadRepository()`, any metadata that may have been cached previously under the given ID, are deleted before the metadata from the file is loaded.
Also, precautions ought to be taken that the metadata descriptions contained in the file are always in sync with the metadata on ABAP side. If, for example, a new field is added to a structure definition, or the length or offset of an existing field is changed, and the repository file is not updated, truncation or corruption of data may be the result.

3 Creating and Working with Data Containers

Now that we have talked at great length about how to get *metadata descriptions*, you may wonder what you need them for. The answer is basically: they serve as “blueprints” for any piece of data that your RFC program wants to exchange with an SAP backend system. You take a metadata description, you create a data container from that blueprint, and that data container can hold field values or even entire tables (which are then sub-containers of the original container) in the correct format needed by the ABAP processor in the backend system.

Basically, there are five different types of data containers:

Table 3-A: Data Container Types

Container	C Type	Description
General data container	`DATA_CONTAINER_HANDLE`	Serves as a kind of “base class” for the other four containers. All setter and getter APIs (see below) are defined for the general `DATA_CONTAINER_HANDLE`, so you can use the same API no matter whether you are setting/getting the parameter of a function or a field of a structure.
Container for a function module	`RFC_FUNCTION_HANDLE`	Holds the values for all parameters of a certain function module. A function module has four types of parameters: `IMPORTING`, `EXPORTING`, `CHANGING` and `TABLES`. Parameters can be “scalar” (simple CHAR, INT or FLOAT values) or “structured”, in which case this container would contain sub-containers of type structure or table.
Container for a structure	`RFC_STRUCTURE_HANDLE`	Holds the values for all fields of a certain structure.
Container for a table	`RFC_TABLE_HANDLE`	Holds the values for all fields of all rows of a certain table. You can traverse the rows of a table by moving a so-called “table cursor” up and down. The table handle can then be used the same way you use a structure handle: the setter/getter APIs then work on the table row to which the cursor currently points to. (This will become clearer in the examples.)
Container for an ABAP Object	`RFC_ABAP_OBJECT_HANDLE`	Originally introduced for working with ABAP class-based exceptions. As already mentioned in the chapter about metadata, these exceptions were discontinued with Kernel release 7.30, so this type can usually be ignored.

3.1 Creating and Destroying Data Containers

Before we describe the APIs for creating and destroying data containers, let's point out, that in most cases you need to create (and destroy) only one single container: a container for the function module you want to call. All sub-containers for the structures and tables of that function module are then created automatically when needed, and destroyed automatically, when the parent container is destroyed. And in the case your program acts as an RFC server, it may not even be needed to create a single container, as the NW RFC library does this automatically. (More on this will be explained later.)

Here now the overview over the different creation and destruction functions. It is in fact simple and works as expected:

Table 3-B: Functions for Data Container Management

Data Container	Creation Function (and input)	Destruction Function
`RFC_FUNCTION_HANDLE`	`RfcCreateFunction(` `RFC_FUNCTION_DESC_HANDLE)`	`RfcDestroyFunction()`
`RFC_STRUCTURE_HANDLE`	`RfcCreateStructure(` `RFC_TYPE_DESC_HANDLE)`	`RfcDestroyStructure()`
`RFC_TABLE_HANDLE`	`RfcCreateTable(` `RFC_TYPE_DESC_HANDLE)`	`RfcDestroyTable()`
`RFC_ABAP_OBJECT_HANDLE`	`RfcCreateAbapObject(` `RFC_CLASS_DESC_HANDLE)`	`RfcDestroyAbapObject()`

Not much else needs to be said about this topic. Only a word of warning that you have to take special care to never destroy a container, which is still used as a sub-container of some other container. This would lead to an access violation/segmentation fault, whenever the parent container tries next time to access the sub-container, or to a heap corruption due to "double-free", when the parent container is destroyed and tries to destroy its sub-containers.

3.2 Accessing the Values of a Data Container

For every data type listed in Table 2-A, there are two setter and two getter APIs defined as follows:

```
RfcGet<Datatype>(DATA_CONTAINER_HANDLE dataHandle,
             SAP_UC const* name, ..., RFC_ERROR_INFO* errorInfo);

RfcGet<Datatype>ByIndex(DATA_CONTAINER_HANDLE dataHandle,
             unsigned index, ..., RFC_ERROR_INFO* errorInfo);

RfcSet<Datatype>(DATA_CONTAINER_HANDLE dataHandle,
             SAP_UC const* name, ..., RFC_ERROR_INFO* errorInfo);

RfcSet<Datatype>ByIndex(DATA_CONTAINER_HANDLE dataHandle,
             unsigned index, ..., RFC_ERROR_INFO* errorInfo);
```

These APIs can be applied to all four types of data containers: functions, structures, tables and ABAP objects. They are used to read or fill a certain function module parameter, structure/table field or ABAP Object attribute. You can access a parameter/field/attribute either by specifying its name (using `RfcGet<Datatype>`) or by specifying its index (using `RfcGet<Datatype>ByIndex`). The index starts at zero and complies to the order of fields as defined in the DDIC (or the order in which the members have been added to the metadata description, when working with hardcoded metadata).

For structures/tables with many fields, the "by-index" APIs have a significant performance improvement compared to the "by-name" APIs. So especially when looping over large tables

and accessing every field in every row, it is worth finding out the index for each field from the metadata and then using the “by-index” APIs.

Another performance improvement that can be used for structures and tables whose line type has only char-like fields (CHAR, NUM, DATE, TIME), is to use the functions `RfcGetStructureIntoCharBuffer()`/ `RfcSetStructureFromCharBuffer()`. This allows getting all fields of a structure with one single API call instead of with one API call for each field. As an example, we do this for the structure `BANK_ADDRESS` of `BAPI_BANK_GETDETAIL`:

```
struct BAPI1011_ADDRESS {
   SAP_UC BANK_NAME[60];
   SAP_UC REGION[3];
   SAP_UC STREET[35];
   SAP_UC CITY[35];
   SAP_UC SWIFT_CODE[11];
   SAP_UC BANK_GROUP[2];
   SAP_UC POBK_CURAC[1];
   SAP_UC BANK_NO[15];
   SAP_UC POST_BANK[16];
   SAP_UC BANK_BRANCH[40];
   SAP_UC ADDR_NO[10];
   SAP_UC fullStop;   // add a terminating zero add the end of
}                     // the structure as a safety net

struct BAPI1011_ADDRESS address;
address.fullStop = 0;

RFC_STRUCTURE_HANDLE bank_address = … // From an invocation of
                                      // BAPI_BANK_GETDETAIL

RfcGetStructureIntoCharBuffer(bank_address, &address,
                    sizeofU(address)-1, &errorInfo);

printfU(cU("Bank Name: %.60s\n"), address.BANK_NAME);
printfU(cU("Swift Code: %.11s\n"), address.SWIFT_CODE);
// etc.
```

A convenience function that may be very useful in a program that gets its data mainly as user input from text fields or reads them from text files like XML files, is `RfcSetString()`/ `RfcSetStringByIndex()`. These functions can be used for parameters and fields of almost any data type and converts the string input into the required data type of the target parameter/field.

For example, if the application got the string value “1234” from a UI or text file and needs to use it as value of some INT4 field, it can do so without first having to convert the string value to an `int`:

```
RfcSetString(someStruct, cU("SOME_INT_FIELD"), cU("1234"), 4, &errorInfo);
```

However, you have to be prepared that the call will fail with return code `RFC_CONVERSION_ERROR`, if the string value cannot be converted to the required data type, for example, if the user enters “xxx” into a field that ought to be a numerical value.

Similarly, the API function `RfcGetString()/ RfcGetStringByIndex()` returns a string representation of fields of any data type. The output of these functions is always zero terminated.

Next it is necessary to talk about who is responsible for creating and deleting the data and how the data flow works. This is slightly different, depending on whether your RFC program is acting as an RFC client or an RFC server. Here is the general overview over these two different cases:

3.3 Data Flow in RFC Client Programs

1. The application code creates a data container for the function module that shall be called.
```
RFC_FUNCTION_DESC_HANDLE stfcStructureDesc = …
                 // See chapter 2 Obtaining Metadata Descriptions
RFC_FUNCTION_HANDLE stfcStructure =
             RfcCreateFunction(stfcStructureDesc, &errorInfo);
```
   Usually, this is all the application needs to do. Even if the function call the application wants to make, uses structures and tables, it doesn't need to create containers for them, as they get created automatically, when needed (lazy initialization). See next point.

2. The application code sets all necessary input parameters of that container using the API functions `RfcSet<Datatype>()`. If a parameter is of type `structure` or `table`, it is recommended to first "get" it from the function container (which creates a suitable sub-container on the fly) and then fill its fields with the setter functions:
```
RFC_STRUCTURE_HANDLE importStruct = RfcGetStructure(
       stfcStructure, cU("IMPORTSTRUCT"), &errorInfo);
RfcSetFloat(importStruct, cU("RFCFLOAT"), 3.1415, &errorInfo);
RfcSetInt(importStruct, cU("RFCINT4"), 42, &errorInfo);
RfcSetTime(importStruct, cU("RFCTIME"), cU("115500"), &errorInfo);
// etc.
```
3. The application code executes the call in the backend system. For this, the RFC library takes the input parameters that the application filled into the data container and sends them to the backend. (See chapter 4 *RFC Client Programs*.)
4. The backend processes the function module and sends the output parameters back to the RFC library.
5. The RFC library receives these parameters and fills them into the same function module data container.
6. The application code reads the output parameters from the container using the API functions `RfcGet<Datatype>()`.
```
RFC_FLOAT resultFloat;
RFC_INT resultInt;
RFC_STRUCTURE_HANDLE echoStruct = RfcGetStructure(
       stfcStructure, cU("ECHOSTRUCT"), &errorInfo);
RfcGetFloat(echoStruct, cU("RFCFLOAT"), &resultFloat, &errorInfo);
RfcGetInt(echoStruct, cU("RFCINT4"), &resultInt, &errorInfo);
// etc.
```
7. The application code destroys the data container for the function module:
```
RfcDestroyFunction(stfcStructure, NULL);
```
   All sub-containers are automatically deleted, when deleting the parent container. This means that your code must not delete them itself, or the application will get crashes caused by "double-free". It also means that your code has to create clones of structures or tables, whose data is used after the lifetime of the corresponding function handle. This can be achieved using the API functions `RfcCloneStructure()` and `RfcCloneTable()`. (Make sure to destroy the clones, once you are done with them.)

3.4 Data Flow in RFC Server Programs

1. The NW RFC library receives an RFC call from the backend.
2. The NW RFC library creates a data container for the called function module and fills the input parameters, it received from the backend, into the container.
3. The NW RFC library calls your implementation function and passes the above container (`RFC_FUNCTION_HANDLE`) into it.
4. The server program reads these input parameters from the function handle using the getter APIs `RfcGet<Datatype>()`, and processes them as necessary.
5. The server program produces the necessary output parameters and fills them into the same function handle, using the setter APIs `RfcSet<Datatype>()`. Again, the application code doesn't need to create sub-containers for exporting structures or tables that it wants to fill: it just calls the getter API on the corresponding parameter of the function handle, and it will get a handle to an already existing sub-container (or the NW RFC lib will automatically create one on the fly, if none is existing yet).
6. Once the program is done with processing the current function call, it simply returns from the implementing function, handing control back to the NW RFC library.
7. The NW RFC library reads the values of the output parameters from the function handle and sends them back to the calling SAP System.
8. The NW RFC library destroys the container for the function module created in step 2. (This also destroys all sub-containers for structures and tables that may have been created in the course of processing the current function call.)

3.5 Working with Tables

Working with table parameters may be a bit unaccustomed at first, especially if you are used to the mechanism the classic RFC library used for processing tables. First note that in the NW RFC library table rows are numbered from 0 to n-1, as is custom in C/C++. This differs from ABAP as well as from the classic RFC library, which number table rows from 1 to n.

Also, you may be tempted to first fetch a certain row, before working with it. This is no longer necessary: you can work directly with the table handle and treat it like a structure handle containing the current row.

The table handle keeps an internal "cursor", which always points to the current row, and all actions performed on the table handle are then applied to the current row. A table handle that is first obtained from a parent container (for example, from a function handle) is always initialized so that the cursor points to the first row (index 0). This allows the following elegant table handling:

```
RFC_ERROR_INFO errorInfo;
unsigned rowCount = 0;
RFC_FUNCTION_HANDLE bapi = … // Handle to some function module
                             // you have executed in the backend.

RFC_TABLE_HANDLE table = RfcGetTable(bapi, cU("TAB"), &errorInfo);

// Check whether the backend sent us some data:
RfcGetRowCount(table, &rowCount, &errorInfo);

if (rowCount) {
   SAP_UC buffer[256] = iU("");
   do {
      RfcGetString(table, cU("FIELD_1"), buffer,
                   sizeofU(buffer), NULL, &errorInfo);
      printfU(cU("Field_1 = %s\n"), buffer);
   } while (RfcMoveToNextRow(table, NULL) == RFC_OK);

}
```

An alternative way of looping over a table is the following index-based one:

```
RFC_ERROR_INFO errorInfo;
unsigned rowCount = 0;
RFC_FUNCTION_HANDLE bapi = … // Handle to some function module
                             // you have executed in the backend.

RFC_TABLE_HANDLE table = RfcGetTable(bapi, cU("TAB"), &errorInfo);

RfcGetRowCount(table, &rowCount, &errorInfo);

for (unsigned i=0; i<rowCount; ++i) {
   RfcMoveTo(table, i, NULL);
   RfcGetString(table, cU("FIELD_1"), buffer,
                    sizeofU(buffer), NULL, &errorInfo);
   printfU(cU("Field_1 in row %d = %s\n"), i, buffer);
}
```

For filling data into a table that is sent into the backend, a wide variety of table manipulation functions is available:

Table 3-C: Table Manipulation Functions

Function	Description
`RfcAppendNewRow()`	Appends a new empty row at the end of the table and moves the table cursor to that row.
`RfcAppendNewRows()`	Appends a set of new empty rows at the end of the table and moves the table cursor to the first new row. If there is a good estimate of how many rows the table will contain in the end, it is advisable to use this API, as it provides a definite performance improvement compared to adding one row at a time for thousands of times.
`RfcInsertNewRow()`	Inserts a new empty row at the current position of the table cursor. The row, on which the table cursor is currently positioned, and all following rows are moved one index "down". For example, if the table currently has rows 0 – n-1 and the cursor points to row i, then the rows i – n-1 are moved to positions i+1 – n, and the new row is inserted at position i.
`RfcDeleteCurrentRow()`	Deletes the row, on which the table cursor is currently positioned. For example, if the row cursor is currently at index i between 0 – n-2, then row i will be deleted and the rows i+1 – n-1 will be moved one index "up" and will now be rows i – n-2. The table cursor will remain fixed at index i. If the cursor is currently on the last row (n-1), then that row will be deleted, all other position will remain unchanged, and the table cursor will move up to index n-2 (the new last row of the table).

`RfcDeleteAllRows()`	Deletes all rows from the table.

In some cases, you may already have the data that shall be inserted into a table in form of a structure handle with the same line type as the table. In this case, the following APIs can be used, which copy all fields of the structure into the current line of the table:

Table 3-D: Row-based Table Manipulation Functions

Function	Description
`RfcAppendRow()`	Creates a new table row at the end of the table and copies the fields of the given RFC_STRUCTURE_HANDLE into that row, but only if the line type of the given structure is identical to the line type of the table. The table cursor is moved to that row (starting with NW RFC library 7.50, patch level 3…)
`RfcInsertRow()`	Inserts a new table row at the current position of the cursor and copies the fields of the given RFC_STRUCTURE_HANDLE into it. The behaviour of the cursor is as described for `RfcInsertNewRow()` above.
`RfcGetCurrentRow()`	Returns the current row as an RFC_STRUCTURE_HANDLE, or NULL if the table is empty or the cursor is at the end of the table. **Note:** this is not a copy. So, changes to the structure are in fact changes to the underlying table row. In particular, the program code should not destroy this structure handle, as it would lead to crashes, if the parent table later tries to access or free the underlying memory!

4 RFC Client Programs

An RFC client program is a program that opens a network connection to an SAP backend system, logs in under a certain user ID and executes a remote-enabled function module on behalf of that user.

4.1 Managing Login Information

In order to log in to the backend, several connection parameters are required, and you ought to think about where to keep these parameters in a secure way, especially if user credentials are concerned.

The necessary parameters can be divided into three parts:

- Parameters identifying the backend system. (Hostname, system number, logon group etc.)
- Parameters identifying the user and configuring its session. (User credentials, language, etc.)
- Technical parameters. (Trace level, compression type, whether to use the delta manager or not, etc.)

The complete list of parameters that an application can use with the NW RFC library is given in the sample file *sapnwrfc.ini* from the demo folder shipped with the NW RFC SDK. Their detailed documentation can be found there, too.

For storing these parameters and passing them to the NW RFC library, there are basically three options available, which will be described in the following three chapters. Almost all parameters described in the sample *sapnwrfc.ini* can be used in all three cases, except for the ones in the GLOBAL section of the *sapnwrfc.ini* file, which can be used only in that file.

Note: when analysing the parameters provided to `RfcOpenConnection()`, the NW RFC library is ignoring the case of the parameter names.

4.1.1 Passing Login Parameters Directly

The simplest and most programmer-friendly way to provide the necessary login parameters is to specify them directly in the code and pass them to the RFC API function that needs them. For example:

```
#define NUM_PARAMS 6

RFC_ERROR_INFO errorInfo;
RFC_CONNECTION_HANDLE conn;
RFC_CONNECTION_PARAMETER loginParams[NUM_PARAMS];

loginParams[0].name = cU("ashost");
loginParams[0].value = cU("hostname");

loginParams[1].name = cU("sysnr");
loginParams[1].value = cU("05");

loginParams[2].name = cU("client");
loginParams[2].value = cU("000");

loginParams[3].name = cU("user");
loginParams[3].value = cU("MickeyMouse");

loginParams[4].name = cU("passwd");
loginParams[4].value = cU("******");

loginParams[5].name = cU("lang");
loginParams[5].value = cU("EN");
```

```
conn = RfcOpenConnection(loginParams, NUM_PARAMS, &errorInfo);

if (errorInfo.code != RFC_OK) {
   printfU(cU("Login failed: %s - %s\n"), errorInfo.key,
             errorInfo.message);
}
```

However, this approach is not advisable for productive programs, because it would have to be recompiled and reinstalled, whenever one of the parameters changes. Furthermore, it violates the principle of separation of concerns: Those parameters are known and configured by an administrator and therefore should not be set by the programmer of an application. If there is an end user using this program, you could prompt him/her with a popup window, where he/she can enter the necessary parameters, and then you fill them into an `RFC_CONNECTION_PARAMETER` array as above and pass it into the corresponding API call. For parameters like username, password or desired logon language this makes indeed sense, but it would be tedious for the end user to have to repeat technical parameters all the time, which he may not know or understand. Therefore, in many situations the best solution is to use a kind of “hybrid” approach:

- The backend-specific and the technical parameters are stored in the *sapnwrfc.ini* file or taken from SAPLogon. There they can easily be changed in case the backend changes (for example, the productive server is temporarily shut down and the backup system takes over) without having to recompile the program.
- The user-specific parameters are prompted from the user at runtime. Then many different users can use the same program, and no sensitive parameters like passwords need to be stored anywhere.

The NW RFC library then allows merging these different sets of parameters very easily. This will be described in the next chapter.

4.1.2 Using sapnwrfc.ini

Instead of passing all parameters directly in the `RFC_CONNECTION_PARAMETER` array as seen above, they can be defined in an *sapnwrfc.ini* file accompanying the application. In the code, these parameters are just referenced via a “destination string”. For example, let’s assume you have an *sapnwrfc.ini* file with the following contents:

```
DEST=PRD
ASHOST=hostname
SYSNR=05
CLIENT=000
USER= testuser
PASSWD=**************
LANG=EN
```

Then an RFC connection can be opened simply like this:

```
RFC_ERROR_INFO errorInfo;
RFC_CONNECTION_HANDLE conn;
RFC_CONNECTION_PARAMETER loginParams;

loginParams.name = cU("dest");
loginParams.value = cU("PRD");

conn = RfcOpenConnection(&loginParams, 1, &errorInfo);

if (errorInfo.code != RFC_OK) {
   //…
}
```

Of course, it is not advisable to keep passwords in an .ini file. Therefore, we now describe the afore-mentioned “hybrid” approach in detail. For this, set up an .ini file that contains only the backend related parameters, possibly for several SAP systems:

```
DEST=PRD
MSHOST=a74main
SYSID=A74
GROUP=PUBLIC
CLIENT=000

DEST=DEV
ASHOST=devmain
SYSNR=18
CLIENT=800
```

The RFC program would then prompt the end user for the missing parameters:

```
#define NUM_PARAMS 4

RFC_ERROR_INFO errorInfo;
RFC_CONNECTION_HANDLE conn;
RFC_CONNECTION_PARAMETER loginParams[NUM_PARAMS];

SAP_UC userName[12+1] = iU("");
SAP_UC password[40+1] = iU("");
SAP_UC language[2+1] = iU("");

loginParams[0].name = cU("dest");
loginParams[0].value = cU("PRD");

loginParams[1].name = cU("user");
loginParams[1].value = userName;

loginParams[2].name = cU("passwd");
loginParams[2].value = password;

loginParams[3].name = cU("lang");
loginParams[3].value = language;

printfU(cU("Please enter user name: "));
gets_sU(userName, sizeofU(userName));

printfU(cU("Please enter password: "));
gets_sU(password, sizeofU(password));

printfU(cU("Please enter logon language: "));
gets_sU(language, sizeofU(language));


conn = RfcOpenConnection(loginParams, NUM_PARAMS, &errorInfo);

if (errorInfo.code != RFC_OK) {
    //…
}
```

Where does the NW RFC library look for the *sapnwrfc.ini* file? The default location is the current working directory (CWD) of the current process. But the application can change this – for example, during start-up – via the API `RfcSetIniPath()`. This automatically loads the .ini file into memory. Furthermore, the program can also trigger a reload of the .ini file into memory by calling the API `RfcReloadIniFile()`, if it got modified during the runtime of the application.

4.1.3 Using Login Parameters Maintained by SAPLogon

Let's refine the approach described in the previous chapter. If the application is running on a Microsoft Windows end user device, then most probably SAPLogon and SAPGui are already installed on that machine, and SAPLogon keeps and maintains connection data for all SAP systems to which this user may need to log on to daily. Wouldn't it be nice, if we could just reuse the data maintained by SAPLogon and don't need to duplicate it in a second file like *sapnwrfc.ini*?

This is indeed possible. SAPLogon traditionally maintains its logon data and system information in a file named *saplogon.ini*. Newer SAPLogon releases (starting with SAPLogon 7.40) also support an alternative XML format in a file named *SAPUILandscape.xml*. The NW RFC library 7.50 supports both formats.

This mechanism works as follows:

1. When SAPGui is installed, it creates several Windows registry keys, defining whether the *saplogon.ini* format or the *SAPUILandscape.xml* format is used, and where the file is stored.
2. When login data from SAPLogon is requested, the NW RFC library first checks the registry key `HKEY_LOCAL_MACHINE\SOFTWARE\SAP\SAPLogon\LandscapeFormatEnabled` or `HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\SAP\SAPLogon\LandscapeFormatEnabled`.
3. If this "enabled" key is set to *false* (=0) or does not exist, the NW RFC library looks for the *saplogon.ini* file in the following locations exactly in that order and takes the first one it finds for the further processing:
   a. Filename given in environment variable `SAPLOGON_INI_FILE`
   b. Directory `%APPDATA%`\SAP\Common
   c. Directory `%WINDIR%` or C:\Windows
   d. Directory or Windows network share given by one of the registry keys
      `HKEY_LOCAL_MACHINE\SOFTWARE\SAP\SAPLogon\Options\ConfigFileOnServer`
      `HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\SAP\SAPLogon\Options\ConfigFileOnServer`
      `HKEY_CURRENT_USER\SOFTWARE\SAP\SAPLogon\Options\ConfigFileOnServer`
4. If the "enabled" key is set to *true* (>0), the NW RFC library looks for the *SAPUILandscape.xml* file in the following locations exactly in that order and takes the first one it finds for the further processing:
   a. Filename given in environment variable `SAPLOGON_LSXML_FILE`
   b. Directory `%APPDATA%`\SAP\Common
   c. The NW RFC Library checks the registry keys
      `HKCU\SOFTWARE[X]\SAP\SAPLogon\Options\PathConfigFilesLocal`
      `HKLM\SOFTWARE[X]\SAP\SAPLogon\Options\PathConfigFilesLocal`
      `HKCU\SOFTWARE[X]\SAP\SAPLogon\Options\LandscapeFileOnServer`
      `HKLM\SOFTWARE[X]\SAP\SAPLogon\Options\LandscapeFileOnServer`
      `HKLM\SOFTWARE[X]\SAP\SAPLogon\Options\CoreLandscapeFileOnServer`
      `HKCU\SOFTWARE[X]\SAP\SAPLogon\Options\CoreLandscapeFileOnServer`
      in that order. Here `[X]` is `\Wow6432Node`, if in step 2 the NW RFC Library found the "enabled" key under `Wow6432Node`, and empty otherwise.
      Here `PathConfigFilesLocal` is either a directory (which may contain two files, *SAPUILandscape.xml* and *SAPUILandscapeGlobal.xml*) or a file name. The parameters `LandscapeFileOnServer` and `CoreLandscapeFileOnServer` can either specify a file on a central Windows network share or a URL pointing to an XML hosted on a Web server, in which case it is fetched with an HTTP GET request.
      All files found that way (and possibly additional files referenced from these files via the `<Includes>` tag) are then merged together.
5. In the end, the application can use the SAPLogon data in a similar way to how it can use the data from the *sapnwrfc.ini* file: when referencing an entry in the *sapnwrfc.ini* file, you would specify an `RFC_CONNECTION_PARAMETER` with name="dest" and

value="value given in the DEST line". Similarly, for referencing an entry defined in *saplogon.ini* or *SAPUILandscape.xml*, you have to specify an `RFC_CONNECTION_PARAMETER` with name="saplogon_id" and value="value given in the `Name` column of SAPLogon"

Let's look at some sample coding to illustrate this. There is one subtle difference to the *sapnwrfc.ini* case, which must be considered, and which complicates the coding a bit: the *sapnwrfc.ini* file usually accompanies the RFC program, and the application has it more or less under its control. This is not the case with the data from SAPLogon, which may differ from frontend installation to frontend installation. There are two problems that need to be taken care of:

1. The program may not necessarily know the names that the end user has given his/her SAPLogon entries.
   This can be solved by obtaining a list of all SAPLogon entries, displaying it to the end user and letting him/her choose one. For this the API function `RfcGetSaplogonEntries()` may prove useful.
2. The program does not know, whether the current entry is set up for SNC login or for user/password login. To solve this problem, the application code can take a look at the parameters of the current entry via `RfcGetSaplogonEntry()` and then depending on the value of the SNC_MODE parameter either pop up the usual user/password prompt, or just let the user choose logon language and client and add the path to the SNC library to the set of parameters, if necessary.

A simple program that does all that, may look as follows:

```
RFC_CONNECTION_PARAMETER loginParams[6];
RFC_ERROR_INFO errorInfo;
RFC_CONNECTION_HANDLE connection;
RFC_CONNECTION_PARAMETER* currentParams;

SAP_UC** logonIDList;
unsigned numEntries, currentEntry = 0, numParams, i;
RfcGetSaplogonEntries(&logonIDList, &numEntries, &errorInfo);

if (errorInfo.code != RFC_OK) {
   // Exit with error
}

printfU(cU("Available SAP Systems:\n"));
for (i = 0; i<numEntries; ++i)
   printfU(cU("[%d] %s\n"), i, logonIDList[i]);

printfU(cU("Please chose one of the above systems [0 - %d]: "),
             numEntries-1);
if (scanfU(cU("%u"), &currentEntry) != 1 || currentEntry >= numEntries) {
       RfcFreeSaplogonEntries(&logonIDList, &numEntries, &errorInfo);
       // Exit with error
}

RfcGetSaplogonEntry(logonIDList[currentEntry], &currentParams,
                    &numParams, &errorInfo);

int sncMode = 0;
for (i = 0; i < numParams; ++i) {
   printfU(cU("%s = %s\n"), currentParams[i].name,
             currentParams[i].value);

   if (strcmpU(currentParams[i].name, cU("SNC_MODE")) == 0 &&
       strcmpU(currentParams[i].value, cU("1")) == 0){
       sncMode = 1;
          break;
   }
}
```

```
RfcFreeSaplogonEntry(&currentParams, &numParams, NULL);

SAP_UC client[4] = iU("");
SAP_UC language[3] = iU("");

printfU(cU("Please enter client: "));
scanfU(cU("%3s"), client);
printfU(cU("Please enter logon language: "));
scanfU(cU("%2s"), language);

loginParams[0].name = cU("SAPLOGON_ID"); loginParams[0].value =
                                          logonIDList[currentEntry];
loginParams[1].name = cU("client"); loginParams[1].value = client;
loginParams[2].name = cU("trace");  loginParams[2].value = cU("3");
loginParams[3].name = cU("lang");   loginParams[3].value = language;

if (sncMode) {
   loginParams[4].name = cU("snc_lib"); loginParams[4].value =
   cU("C:\\Program Files\\SAP\\FrontEnd\\SecureLogin\\lib\\sapcrypto.dll");
   numParams = 5;
}
else {
   SAP_UC user[13] = iU("");
   SAP_UC password[41] = iU("");

   printfU(cU("Please enter user name: "));
   scanfU(cU("%12s"), user);
   printfU(cU("Please enter password: "));
   scanfU(cU("%40s"), password);
   loginParams[4].name = cU("user");   loginParams[4].value = user;
   loginParams[5].name = cU("passwd"); loginParams[5].value = password;
   numParams = 6;
}

connection = RfcOpenConnection(loginParams, numParams, &errorInfo);
RfcFreeSaplogonEntries(&logonIDList, &numEntries, NULL);

if (connection == NULL) {
   printfU(cU("Error during logon: %s\n%s: %s\n"),
             RfcGetRcAsString(errorInfo.code),
             errorInfo.key, errorInfo.message);
   // Exit
}
```

4.2 Synchronous RFC Client

Now there is everything in place to describe the general design of an RFC client program. Such a program needs to perform the following steps:

1. Get the necessary login information for the backend system.
2. Open a connection to that backend system.
3. Lookup the metadata description for the function module(s) it wants to call, or get that metadata in one of the other ways described in chapter 2 *Obtaining Metadata Descriptions.*
4. Create a function module container (`RFC_FUNCTION_HANDLE`) from the metadata and fill the inputs.
5. Execute the function module in the backend and then either
    a. process the error details in case something went wrong, or
    b. process the output values of the function module.
6. Delete the function module container and close the connection. (Or execute further calls as necessary.)

Most of these points we have already discussed in the previous chapters. Let's now fill in a few useful details.

4.2.1 Opening a Connection

As we have already seen in the previous samples, opening an RFC connection can easily be done with the API `RfcOpenConnection()`. But there are quite a few things that can go wrong in this step:

1. The target system cannot be reached. For example, a network problem, wrong hostname, target system is currently down. All these cases will result in a COMMUNICATION_FAILURE, which corresponds to `errorInfo.code == RFC_COMMUNICATION_FAILURE`. The member `errorInfo.message` will give further details about what exactly went wrong, usually a standard TCP/IP error message from the operating system's TCP/IP stack.
2. The target system is currently in deep trouble and cannot accept any user logins. For example, no connection to its database, currently running an upgrade. This will result in a SYSTEM_FAILURE, which corresponds to `errorInfo.code == RFC_ABAP_RUNTIME_FAILURE`, or in an ABAP Message (`RFC_ABAP_MESSAGE`), depending on where in the stack between low level kernel and ABAP engine the abort happened. In both cases, the `message` member as well as the `abapMsg` fields (SY-MSG fields) may give further details.
3. There is a problem with the current user. For example, wrong password, user locked, insufficient authorizations, insufficient authentication information. This will be indicated by the return code `errorInfo.code == RFC_LOGON_FAILURE` and the `message` member providing further details.

From this you see that it is quite important to have a thorough error check after the `RfcOpenConnection()` call and to prepare for all these cases.

This seems to be an appropriate point to say a few words about thread-safety. Of course, the NW RFC Library is thread-safe in the sense that you can call any API simultaneously in different threads. However, that does not mean the application can also use the same "object", like a connection or a table, in several threads at the same time and expect something reasonable to happen. Here are two typical examples, that were implemented in practice:

- A connection handle is shared between two threads and both of them are trying to execute a function call at the same time. Common sense already indicates that this can lead to no good. This would be similar to opening one single HTTP connection to an HTTP server and then sending two different GET or POST requests over that

connection at the same time from two parallel threads. Even if the HTTP server would somehow be able to correctly recognize and serve both requests, there would be no guarantee as to which of the two threads receives which HTTP response… Most probably both threads would end up reading parts of each other's responses.
As is the case with any kind of connection-based network programming (like HTTP connections, ODBC connections, etc.) an RFC_CONNECTION_HANDLE must not be shared between different threads.

- A table handle is shared across two threads, and both threads start filling rows into that table. Even if the insert operations were synchronized, it would not help: the end result would still be a completely intermingled sequence of thread A's and thread B's data, depending mainly on how the CPU distributes the time-slots between the two threads.
  Therefore, for best performance, the NW RFC API does not synchronize any table operations and relies on table handles not being shared between two threads.

Consequently, as a general rule, all NW RFC library objects that are represented by a HANDLE, must not be shared across threads, the exception being immutable objects like metadata descriptions.

4.2.2 Controlling Input and Output Parameters

As we have already seen, there is a plethora of `RfcSet…` functions, which can be used to set the parameter and field values of function modules, structures and tables. However, there are two more useful features worth knowing:

Function module parameters, no matter whether they are optional or non-optional, may have so-called "default values" defined in their definition in the function builder (transaction SE37). If the client does not send a value for the parameter, the server uses this default value when executing the code of the function module. If you have not set a value for a certain parameter, the NW RFC library does not send any value for this parameter, so you can use this feature quite easily. But if you have set a value and then later decide you would much rather have the backend use the default value, there is no way back: even setting an "initial value" for the data type (like empty string for `CHAR` or zero for `INT/FLOAT` parameters) does not undo the first setter: the NW RFC library will then send the initial value, and the backend will use that one instead of the default value. However, in order to instruct the NW RFC library to not send this parameter, the program can use the API function
`RfcSetParameterActive(funcHandle, cU("NAME"), 0, &errorInfo)`.
The parameter is then "inactive" and will not be sent.

Similarly you can use the opposite mechanism to suppress the usage of the backend-side default value: set the parameter to "active", and the NW RFC library will send the proper initial value for the current data type, which will then be used instead of the default value:
`RfcSetParameterActive(funcHandle, cU("NAME"), 1, &errorInfo)`.

Moreover, this mechanism of active/inactive parameters cannot only be used to control the usage of default values for input parameters, it can also be used to control the behaviour of output parameters. Let's illustrate this with an example: suppose the application needs to call a certain BAPI that returns lots of data in a dozen table parameters, but the scenario requires only the content of one of these tables. Then the program code can set all the other table parameters that are not needed to "inactive". This has at least one, maybe even two performance benefits:

1. The RFC layer in the backend, when preparing to serialize the output data of the BAPI after the ABAP code of that BAPI is finished executing, will notice that these tables are inactive (not requested by the client) and will therefore not send any data for them. This may reduce the network traffic significantly. This is guaranteed to happen.
2. Additionally, the ABAP code in the BAPI also has a chance to find out, whether a certain export, changing, or table parameter is active or not, using the ABAP keyword `IS SUPPLIED` (or `IS REQUESTED` which is obsolete and should no longer be used). So if the BAPI is programmed in an intelligent way, it can skip a few potentially very

expensive SQL statements needed for collecting the data for these unrequested tables and thus reduce the stress on the underlying backend database significantly. (**Note**: in older versions of the ABAP documentation, in the `IS SUPPLIED` keyword section it was stated that `IS SUPPLIED` does not work, if the function module has been called from an external RFC interface. However, this still refers to the old classic RFC library. All modern RFC Connectors – like JCo 3.x, NCo 3.x and the NW RFC library – now support this feature.)

4.2.3 Executing the Function Module

After all the preparations of the previous chapters, executing a function module in the backend now is not particularly difficult, just one line of code:

```
RfcInvoke(connection, funcHandle, &errorInfo);
```

However, just as was the case with `RfcOpenConnection()`, there are a lot of things that can go wrong with `RfcInvoke()`. Therefore, a proper error handling is important in order to make the program robust and to provide the end user with all available information that may help troubleshooting an issue, once it has occurred. Here is now a list of possible error situations the client program must be prepared for.

1. First, there is still the possibility that the connection breaks down due to network problems or because the communication partner goes down unexpectedly. This will result in `RFC_COMMUNICATION_FAILURE` with error details provided in the `message` parameter. The connection is of course closed afterwards, and the application has to open a new one.

2. Next the function call could run into a serious technical problem on ABAP kernel level, for example, an attempted division by zero or an out of memory situation. This results in a so called SYSTEM_FAILURE and will be indicated by the NW RFC library by return code `RFC_ABAP_RUNTIME_FAILURE` with error details being provided in the `key` and `message` parameters.
   The backend logs error details about what happened as a short dump, which can then be analysed in the ABAP runtime error monitor (transaction ST22). Again, the connection is closed as a result of a SYSTEM_FAILURE, so the application has to open a new one.

3. Also the ABAP code of the called function module could run into a serious problem and decide to abort processing using the ABAP statement
   `MESSAGE ID '..' TYPE '.' NUMBER '...' WITH ....`
   Here, `ID` and `NUMBER` identify a so-called T100-message, `TYPE` specifies the error type (see below) and following `WITH` there can be up to four runtime parameters, which are inserted into the place holders of the T-100 message (represented by ampersands: &). The long text corresponding to a certain T100-message can be looked up in the message maintenance screen (transaction SE91): you enter the `ID` into the field “Message class” and the `NUMBER` into the field “Number”, and can then display the error text in the current logon language.

   The effect of a `MESSAGE` on RFC communication depends on the error type. There are six possible types, sorted by increasing severity: I (Info), S (Status), W (Warning), E (Error), A (Abend), and X (Exit). The first three are ignored by RFC, so you will not get notified, if the ABAP code issues one of them. The execution of the function module will continue normally after such a message. However, the other three will abort the execution, and the connection will be closed. The NW RFC library reports this case to the calling code via return code `RFC_ABAP_MESSAGE`. What kind of error details it will get, depends on the error type:
   a. Type ‘E’: This case is very similar to an ordinary ABAP exception, the difference being, that in this case the user session is ended, the connection is closed and an entry in the current work process trace is written, if the trace

level of the system is high enough.
The NW RFC library fills the members of the error info structure as follows:
`message`: the error text of the T100-message in the current logon language, as would be displayed by the message maintenance screen.
`abapMsgClass`: the `ID` of the message. (Corresponds to SY-MSGID.)
`abapMsgType`: 'E'. (Corresponds to SY-MSGTY.)
`abapMsgNumber`: the `NUMBER` of the message. (Corresponds to SY-MSGNO.)
`abapMsgV1 – V4`: the optional runtime parameters specified via `WITH`. (Corresponds to SY-MSGV1 – SY-MSGV4.)

b. Type 'A': This is a more serious error, which is logged by the SAP system in the system log under SysLog Area D01. You can analyse the system log using transaction SM21.
The NW RFC library fills the error info members the same way as in the previous case, with the exception that **`abapMsgType`** of course equals 'A'.

c. Type 'X': This is such a severe error that the work process on SAP side is aborted and a short dump with "runtime error = MESSAGE_TYPE_X" is generated. Again, you can analyse the short dump in the ABAP runtime error monitor (transaction ST22).

Unfortunately, the error details that the RFC layer transports to the client in this case, depend on the release of the ABAP kernel. In releases up to 7.49 only the message details of this MESSAGE_TYPE_X message are sent to the RFC client. Therefore the error info structure is always filled with the following hardcoded values:
`key`: MESSAGE_TYPE_X
`message`: "The current application has triggered a termination with a short dump."
`abapMsgClass`: "00"
`abapMsgType`: 'X'
`abapMsgNumber`: "341"
`abapMsgV1`: MESSAGE_TYPE_X
Therefore, if you want to find out what happened, you will have to look at the short dump in the ABAP runtime error monitor (ST22), note the values of the SY-MSGID, MSGNO and MSGV1 – MSGV4 fields, and copy them to the message maintenance screen (SM91).

Starting with Kernel release 7.50, the message class, number and V-parameters of the "real" ABAP message that triggered the short dump, are sent to the RFC client. So now the error info fields are filled the same way as described for type 'E' and 'A'.

4. And finally, there may be a problem on application level. From a technical point of view, everything may be fine, but for example, the end user has entered a bank account number that does not exist, and consequently the business logic of the ABAP function module cannot continue and reports this mistake by raising an ABAP exception. This can be done via two different ABAP statements:
```
RAISE name_of_exception.
```
or
```
MESSAGE ID '..' TYPE '.' NUMBER '...' WITH ... RAISING
name_of_exception.
```

Note: even though the second form looks very similar to the ABAP message discussed in the previous point, it does not trigger an ABAP message, but only a "harmless" ABAP exception!
So in both cases no user sessions are ended, no work processes aborted and no RFC connections closed... Only a simple application level error message is transported back to the client program, which can then continue using the connection as before.

The NW RFC library indicates an ABAP exception via return code `RFC_ABAP_EXCEPTION`. If the ABAP code used the keyword `RAISE`, then the **`key`** member of the error info structure is filled with the name of the exception. If the ABAP code used `MESSAGE … RAISING`, then in addition the "**`abapMsg`**" parameters are filled with the details of the T100-message as described under 3.a above.

All this is probably confusing when reading about it the first time, so let's summarize it in a small table:

Table 4-A: Error Types in RFC

Error	Description	Return Code (Connection State)	RFC_ERROR_INFO members	Error Info in Backend
Communication Failure	Problem in network layer or hardware, crash of communication partner	`RFC_COMMUNICATION_FAILURE` (Connection is closed)	`message`	— Possibly some traces in the gateway monitor (**SMGW**)
System Failure	Technical problem on ABAP system kernel level	`RFC_ABAP_RUNTIME_FAILURE` (Connection is closed)	`key` `message`	ABAP runtime error monitor (**ST22)**

ABAP Message	Technical problem on ABAP level	`RFC_ABAP_MESSAGE` (Connection is closed)	`key`
			`message`
			`abapMsgClass`
			`abapMsgType`
			`abapMsgNumber`
			`abapMsgV1`
			`abapMsgV2`
			`abapMsgV3`
			`abapMsgV4`
				Type = 'E'
				work process traces (can be viewed within SAP directories **AL11**)
				
				Type = 'A'
				system log **(SM21)**
				
				Type = 'X'
				ABAP runtime error monitor (**ST22)**

ABAP Exception	Problem on application level or in the business logic	`RFC_ABAP_EXCEPTION` (Connection remains open)	If triggered via keyword 'RAISE':
			`key`
			
			If triggered via keyword 'MESSAGE … RAISING':
			`key`
			`message`
			`abapMsgClass`
			`abapMsgType`
			`abapMsgNumber`
			`abapMsgV1`
			`abapMsgV2`
			`abapMsgV3`
			`abapMsgV4`
				—
				
				message maintenance screen (**SE91)**

Of course, if `RfcInvoke()` ends with `RFC_OK`, the program now proceeds with processing the function module's return parameters as indicated in chapter 3.2 *Accessing the Values of a Data Container.*

We end this chapter by noting one fundamental difference the NW RFC library has compared with the other two connectors, SAP Java Connector (JCo 3.1) and SAP .NET Connector (NCo 3.0): every RFC client connection you open with the NW RFC library is *stateful* by default. In the other two connectors they are *stateless* by default and you have to put some effort into it, if you need a stateful connection. This of course makes calling an Update-BAPI or some other RFM that stores intermediate results in the current user's ABAP session memory very easy. For example, the program just calls the BAPI or function module, and if it succeeds, just calls

`BAPI_TRANSACTION_COMMIT` on the same connection, and it will run inside the same user session in the backend.

However, there are also situations, where the business logic needs a stateless connection, for example, if you have just called a function module that stored a lot of data in the ABAP session memory, you now want to call further RFMs in the same user session, and the leftover data from the previous call is now superfluous garbage that is no longer needed, which degrades the performance, or – even worse – causes unwanted side effects for the following call(s).

Of course, the application could simply close the connection and open a fresh one, which also creates a fresh ABAP user session, but this could be a bit time and resource consuming, especially when an SNC handshake must be performed during login. In this case it is easier to just use the API function `RfcResetServerContext()`. This function cleans up any user session state in the backend, but keeps the connection and the user session alive.

Note: In contrast to NCo and JCo, stateless communication induces a larger performance penalty compared to stateful communication, as it requires an additional roundtrip to the backend that is performed when calling `RfcResetServerContext()`.

4.3 Transactional RFC Client

There are three variants of the RFC protocol that provide some form of transactional security:

1. Transactional RFC (tRFC), which guarantees the execution of a logical unit of work (LUW) ***exactly once***.
2. Queued RFC (qRFC), which guarantees the execution of multiple LUWs ***exactly once in order***.
3. Background RFC (bgRFC), which is the successor of the previous two variants and can be used in both ways, as a transactional communication (type T), or as a queued communication (type Q). The main improvement over the older variants is, that the scheduler used for processing the queues, has a better performance due to improved load distribution approaches and algorithms for deadlock prevention.

Here, a *logical unit of work* could be a single function module or a sequence of several function modules, which are then executed as an atomic unit: either all of them are executed or none is.

The basic concepts for all three variants are the same. Therefore, we explain them once using the example of tRFC. For qRFC, you just have to add a queue name at the right location (we will indicate this, when we get to that point), and for bgRFC you just have to replace some API functions that have the term “Transaction” in their name, with the corresponding ones that have “Unit” in their name. This will be explained at the end of the current discussion.

Transactional security does not come for free: it requires a bit of effort on the part of the external program. Just having a transaction ID and using the tRFC protocol does not guarantee transactional security. You could still experience lost transactions or duplicate transactions, if the external program does not implement the state handling of the transactions correctly. In order to avoid the common pitfalls, you need to understand the “philosophy” behind the tRFC protocol. Here is the general idea how it works:

The tRFC protocol is implemented as a sequence of steps/actions between the client (sender) and the server (receiver). Over the entire span of this sequence, both sides have to keep track of the current state of the transaction. Five different states are possible, but not all of them are necessarily taken on by every transaction: Created, Executed, Committed, Rolled Back and Confirmed. A transaction now passes through these states as follows:

1. The client generates a 24-digit transaction ID (TID) and creates the payload for the current transaction. It should now persist the payload together with the TID (in status Created), so that it can be resent multiple times, if this should become necessary. Then the client opens a connection to the server and transmits the TID. The server

creates an entry for this TID in its status-keeping component (which usually is a database), sets the status of this transaction to Created and returns ok to the client. If the server side is currently not able to persist status information because of technical problems (for example, database down or full), it should return an error to the client, so that the transaction can be aborted and retried at a later time.
If the server already knows the TID, it should also return ok. The prevention of duplicates is taken care of in the next step.

2. Now the client transmits the actual data of the transaction. The server's action then depends on the transaction's state in the server-side state-keeping component:
    a. If it is Rolled Back or Created, the server now executes the function modules of the LUW. In an ideal scenario, the server at this point would only insert the transaction's data into a database *without triggering the `COMMIT WORK`*. This is important especially if the LUW consists of more than one function module, because if you would already persist the data of FM1 and then FM2 ends with an exception, it is no longer possible to revert the changes of FM1, violating the atomic property ("all or nothing") of the LUW. However, if the expected LUWs consist of only one single FM, it is ok to work without a database. At the end, the server returns OK or an error, depending on whether the current function module ended successfully or not. This step is repeated once for every function module contained in the LUW.
    b. If the server-side state of this transaction is already Executed or Committed, the server simply ignores the data and returns OK to the client. This detail is important, as it allows the client to resend a transaction as many times as it wants, until it finally got an OK code from the server and therefore knows for sure, that the data has arrived at the server side successfully. The client can do so without having to fear duplicates.
3. If the server runtime got an OK for every FM of the LUW, it triggers a `COMMIT WORK` event, if not, it triggers a `ROLLBACK WORK` event. The server has to perform the corresponding action on its database (if it is using a database) and then set the state of the transaction to Committed or Rolled Back respectively and return the fitting response to the calling client.
4. If during the entire procedure up to now the client receives an error or no response at all (because of timeout or network errors), it should set the state of the transaction in its local state-keeping component to Rolled Back and try again later. When repeating the transaction execution later, the client has to use the very same TID again that was used originally, in order to guarantee the exactly-once promise of tRFC.
If it got an OK from step 3, it can now proceed to the last stage, which is clean-up of resources. The client should now delete the TID as well as the payload data from its local storage in order to guarantee that this transaction is never again retransmitted to the server. If this is successful, it can trigger the Confirm event, which tells the server that the client is now aware of the fact that the transaction has finished successfully. The Confirm event is basically a promise from the client to the server, that it will never resend this transaction with this TID again. The server therefore no longer needs to protect itself against duplicates and can now delete this TID from its state-keeping component (cleaning up resources).

So, we can observe here, that the following two basic ingredients of the tRFC protocol are the ones that guarantee transactional security: the client keeps sending the transaction with its associated TID until it is convinced that the data has been processed by the server successfully. This ensures the "at least once" part of the equation. And the server keeps track of the status and just ignores data for TIDs it already processed before. This ensures the "at most once" part of the equation. Both parts combined then guarantee the "exactly once" quality of tRFC.

The worst-case scenario that can be imagined in the communication between two systems, is when the client sends the data successfully to the server, the server processes it successfully, but while trying to transmit the OK response back to the client, the network connection breaks down. In this case the client only sees the network error but has no information about whether the data arrived at its destination, and if it did, whether it was processed there successfully or ran into an error. But even that case is no problem for tRFC: the client just sends the transaction using the same TID again. If the server doesn't know it

yet, it processes it now, if the server recognizes the transaction as already processed previously, it just ignores it and sends another OK response, which this time will hopefully make it back to the client.

Note: there is one common misunderstanding about tRFC, which has to be cleared up here: in the case of an external program talking to an ABAP system, tRFC is **not** an asynchronous execution, technically it's fully synchronous. If you want an asynchronous execution - i.e. the client sends the data, the server only stores it and then processes it at a later point in time - then you need to use qRFC or bgRFC of type Q.

Let's now proceed to some pseudo code that illustrates how a client program may use tRFC in a secure way. We will split the functionality into two functions, one for creating and sending a new LUW, and another one for resending old failed ones. Both use a third function for the actual sending. We also assume that the program has access to two further components: one for keeping track of the status of all transactions (possibly in a database), and another one for persisting the payload data safely in case it is needed for a later resend attempt.

```
// An API for keeping track of transaction status.

typedef enum _TIDStatus {
   Status_Created,
   Status_Executed,
   Status_Committed,
   Status_RolledBack,
   Status_Confirmed
} TIDStatus;

typedef enum _StatusRC {
   RC_OK,
   Error,
   NotFound,
} StatusRC;

StatusRC getTIDStatus(RFC_TID tid, TIDStatus* status,
                      SAP_UC* errorMessage);
StatusRC setTIDStatus(RFC_TID tid, TIDStatus status,
                      SAP_UC* errorMessage);
StatusRC deleteTID(RFC_TID tid);
StatusRC listTIDs(RFC_TID** tids, unsigned* numEntries);


// An API for persisting the payload data.

class PayloadData;
StatusRC persistPayload(RFC_TID tid, PayloadData* payload);
StatusRC readPayload(RFC_TID tid, PayloadData* payload);
StatusRC deletePayload(RFC_TID tid);
void fillPayloadIntoFunction(RFC_FUNCTION_HANDLE function,
                             PayloadData* payload);

// The basic functions of our program

void sendTransaction(RFC_CONNECTION_HANDLE connection, RFC_TID tid,
                     RFC_FUNCTION_HANDLE function) {
   RFC_RC rc = RFC_OK;
   RFC_ERROR_INFO errorInfo;

   RFC_TRANSACTION_HANDLE tHandle =
             RfcCreateTransaction(connection, tid, NULL, errorInfo);
   if (tHandle == NULL) {
```

```
        setTIDStatus(tid, Status_RolledBack, errorInfo->message);
        return;
    }

    rc = RfcInvokeInTransaction(tHandle, function, errorInfo);
    if (rc != RFC_OK) {
        setTIDStatus(tid, Status_RolledBack, errorInfo->message);
        RfcDestroyTransaction(tHandle, NULL);
        return;
    }

    rc = RfcSubmitTransaction(tHandle, errorInfo);
    if (rc == RFC_OK) {

        StatusRC src = deletePayload(tid);
        if (src == RC_OK) {
            // If the payload has been deleted successfully, we can now
            // confirm the TID in the backend without risk.
            rc = RfcConfirmTransaction(tHandle, errorInfo);
            if (rc == RFC_OK)
                deleteTID(tid);
            else // We can try a confirm later.
                 setTIDStatus(tid, Status_Committed, NULL);
        }

    }
    else {
        setTIDStatus(tid, Status_RolledBack, errorInfo->message);
    }

    RfcDestroyTransaction(tHandle, NULL);
}


// initialTransaction will be called in your program, whenever a new
// payload has been created that needs to be sent to the backend system.

void initialTransaction(RFC_CONNECTION_HANDLE connection,
                        PayloadData* payload) {

    RFC_RC rc = RFC_OK;
    StatusRC status_rc = RC_OK;
    RFC_TID tid;
    RFC_FUNCTION_DESC_HANDLE functionDesc = NULL;
    RFC_FUNCTION_HANDLE function = NULL;
    RFC_ERROR_INFO errorInfo;

    // First make sure, we get a transaction ID for our data and
    // persist it safely for potential later retries.

    rc = RfcGetTransactionID(NULL, tid, &errorInfo);
    if (rc != RFC_OK)
        goto severe_error;

    status_rc = persistPayload(tid, payload);
    if (status_rc != RC_OK)
        goto severe_error;

    status_rc = setTIDStatus(tid, Status_Created, NULL);
    if (status_rc != RC_OK)
        goto severe_error;


    // Once the data is stored, we make our first send attempt.
```

```
   functionDesc = RfcGetFunctionDesc(connection, cU("FUNCTION_NAME"),
                                     &errorInfo);
   if (functionDesc == NULL)
      goto rollback;

   function = RfcCreateFunction(functionDesc, &errorInfo);
   if (function == NULL)
      goto rollback;

   fillPayloadIntoFunction(function, payload);

   sendTransaction(connection, tid, function);

   RfcDestroyFunction(function, NULL);
   return;

   // Error handling:

severe_error:
   // Log the error and notify an administrator, so the problem can
   // be fixed. The payload data is irreversibly lost.
   return;

rollback:
   setTIDStatus(tid, Status_RolledBack, errorInfo->message);

}

// Can be run periodically in a scheduler thread, for example, once per
// hour, or started manually by an administrator, after a certain error
// situation has been resolved.

void retryTransactions(RFC_CONNECTION_HANDLE connection) {
   unsigned numEntries = 0, i;
   RFC_TID* tids = NULL;
   StatusRC status_rc = RC_OK;
   RFC_ERROR_INFO errorInfo;

   status_rc = listTIDs(&tids, &numEntries);

   if (status_rc == RC_OK) {
      TIDStatus status;
      PayloadData payload;

      for (i=0; i<numEntries; ++i) {
         status_rc = getTIDStatus(tids[i], &status, NULL);

         if (status_rc == RC_OK) {
            switch(status) {
               case Status_Created:
               case Status_RolledBack:
                  status_rc = readPayload(tids[i], &payload);
                  if (status_rc == RC_OK)
                     goto severe_error;

                  functionDesc = RfcGetFunctionDesc(connection,
                                    cU("FUNCTION_NAME"), &errorInfo);
                  if (functionDesc == NULL)
                     goto rollback;

                  function = RfcCreateFunction(functionDesc, &errorInfo);
                  if (function == NULL)
                     goto rollback;

                  fillPayloadIntoFunction(function, &payload);

                  sendTransaction(connection, tids[i], function);
```

```
                    RfcDestroyFunction(function, NULL);
                    break;

                case Status_Committed:
                    // Processing the LUW already worked successfully last
                    // time, only the Confirm call ran into a problem.
                    // Just repeat the Confirm in order to clean up the
                    // backend. If this attempt again runs into an error,
                    // we just keep the status entry as is and try again
                    // next time.
                    RFC_TRANSACTION_HANDLE tHandle =
                        RfcCreateTransaction(connection, tids[i], NULL,
                                             errorInfo);
                    if (tHandle != NULL) {
                        RfcConfirmTransaction(tHandle, errorInfo);
                    if (errorInfo.code == RFC_OK)
                         deleteTID(tids[i]);
                     RfcDestroyTransaction(tHandle, NULL);
                }
            }

            continue;

            // Error handling:
            severe_error:
            // Payload could not be read. Log error and inform an
            // administrator.
            continue;


            rollback:
            setTIDStatus(tid, Status_RolledBack, errorInfo->message);
        }
        else {
            // Status for this TID could not be read from our status
            // management component. Log the error, notify an
            // administrator and continue with next TID.
            continue;
        }
    }
}
else {
    // Something wrong with our status management component.
    // Log the error and notify an administrator.
}

}
```

4.4 Queued and Background RFC Client

Now we discuss the differences that a qRFC or bgRFC client would need to implement compared to a tRFC client. qRFC is a bit easier, so let's start with that.

In order to switch the above sample program from tRFC to qRFC, two things need to be changed:

1. On backend side, you have to create a qRFC inbound queue. This can be done in qRFC Monitor (transaction SMQS) → Goto → Monitors → QIN-Scheduler.
2. In the helper function `sendTransaction()` above in the invocation of `RfcCreateTransaction()`, you have to replace the third argument "NULL" with the name of the qRFC inbound queue.

In order to switch the above sample program from tRFC to bgRFC, several things need to be changed:

1. On backend side, you have to create a bgRFC inbound queue. This can be done in the bgRFC configuration (transaction SBGRFCCONF).
2. In the helper function `initTransaction()`, replace the call to `RfcGetTransactionID()` and `RFC_TID` with `RfcGetUnitID()` and `RFC_UNITID`. (And in general, all APIs that use an `RFC_TID`, need to be switched to `RFC_UNITID`.)
3. In the helper function `sendTransaction()`, replace the `RFC_TRANSACTION_HANDLE` and the calls to `RfcCreateTransaction()`, `RfcInvokeInTransaction()`, `RfcSubmitTransaction()`, `RfcConfirmTransaction()` and `RfcDestroyTransaction()` with an `RFC_UNIT_HANDLE` and the corresponding functions `RfcCreateUnit()`, `RfcInvokeInUnit()`, `RfcSubmitUnit()`, `RfcConfirmUnit()` and `RfcDestroyUnit()`.
4. The call to `RfcCreateUnit()` needs a bit more explanations. This function has quite a few additional parameters compared to `RfcCreateTransaction()`:
   - **queueNames & queueNameCount:** Contrary to the old qRFC, where only one queue name could be used, a bgRFC unit of type Q can be sent to several inbound queues in parallel. The program can just pass their names in this list. If it is sending a bgRFC unit of type T, the application should just set these two parameters to 0.
   - **unitAttr:** This parameter allows passing several additional flags and processing instructions to the bgRFC call. In most cases, the application can just pass an initialized structure to `RfcCreateUnit()`, as the default values used by the NW RFC library are exactly what makes sense. This parameter becomes interesting only, if you want to trace or debug the current bgRFC unit inside the backend system.
   - **identifier:** This structure is filled by `RfcCreateUnit()`. Keep it for later, (or make sure to store in the application, which unit was of type T and which of type Q), then the application can use the identifier in `RfcGetUnitState()` and `RfcConfirmUnit()`.
     This, by the way, is a feature that's new to bgRFC: in bgRFC you can at a later point of time query the backend system for the current processing state of a particular unit by executing `RfcGetUnitState()` with the corresponding unit identifier. This will tell the program, whether a unit is still waiting in the queue, whether it was already executed and failed, or whether it was executed successfully.
5. The status management component (database) must be changed a bit, because unit IDs for bgRFC are 32 digits plus one char for the unit type (T/Q), whereas transaction IDs for tRFC/qRFC are only 24 digits.

But other than the above differences, the concepts of these three transactional RFC types are quite similar.

4.5 WebSocket RFC Client

Starting with kernel release 777 (used in S/4HANA 1909), an ABAP system can receive RFC calls not only via the CPIC protocol and the gateway port, but also via a so-called WebSocket protocol and an HTTP(S) ICM port. This can be handy, if the RFC client and receiving system are separated by a firewall, as now the RFC call can be made via a standard HTTP Proxy in the firewall.

So, what needs to be done to create an RFC client program that uses WebSocket communication? Well, basically: nothing. This is one of the charming points about WebSocket RFC: all you need to do is to provide different logon parameters to `RfcOpenConnection()`. For example, instead of providing ASHOST and SYSNR, you provide the parameters WSHOST and WSPORT. Here you would specify either the hostname of the SAP application server and the HTTP(S) port on which its ICM is listening, or the hostname and port number where an external RFC Server program is listening. Most of the remaining parameters stay the same, with three noteworthy exceptions:

1. Instead of with SAProuter and SAProuter-strings, you will work with HTTP Proxy settings in order to communicate across subnet boundaries.
2. Instead of SNC settings and PSEs, you will use TLS and certificate stores for encryption and certificate-based logon.
3. Parameters maintained by SAPLogon (*saplogon.ini*/*SAPUILandscape.xml*) cannot be used.

The complete set of parameters to be used with WebSocket RFC communication is documented as usual in the *sapnwrfc.ini* file that comes with the NW RFC SDK. See the section 4.1 in the *sapnwrfc.ini* file.

Hence, in theory it is possible to switch an existing RFC client program from CPIC-based RFC to WebSocket RFC without having to modify or even recompile it. Just replace the sapnwrfc library with the latest version that supports WebSocket and change the logon parameters, for example, in *sapnwrfc.ini*, and you are ready to go.

5 RFC Server Programs

There are three different types of RFC server programs:

1. **Registered RFC Server:** This is the most common type. The RFC program registers itself at the SAP system's CPIC/RFC gateway and then listens for incoming RFC requests. This is the best option for servers that continuously have to process many requests and need a high performance.
2. **Started RFC Server:** The RFC program does not run continuously, but resides in a certain location as an executable, which is then started on demand by the CPIC/RFC gateway. This is an old mechanism used, for example, by SAP standard programs like "tp". There are several different ways on how and where the server executable can be started: via fork or via remote shell, on the current application server, on a specific application server or on the frontend machine (if the current user session is logged in via SAPGui).
   As there is a big overhead for starting and cleaning up a new OS level process, this mechanism is not recommended for scenarios, where a large number of single calls are to be executed.
3. **WebSocket RFC Server:** This type is available for client systems as of SAP kernel release 7.77. The server program is running continuously, but in contrast to the registered server, no network connection needs to remain open all the time. A network connection from the SAP system to the server program is opened only, when RFC requests are being executed.
   This type is well suited for scenarios, in which the SAP system and the server program reside in different LAN segments. When using the traditional registered server in such a setup, firewalls or network idle timeouts would cause problems by unexpectedly closing the network connection that needs to remain open constantly between the CPIC/RFC gateway and the registered server.
   Furthermore, this low-level connection protocol can easily traverse existing standard network infrastructure in corporate networks such as proxies as establishing the connection is based on the HTTP/S protocol.

Before we discuss these different types in more detail, let us first have a look at a few basic tasks, which must be performed by every RFC server program, independently of which type it is.

5.1 Preparing a Server Program for Receiving RFC Requests

The steps described in the current chapter remain the same for every type of RFC server program.

5.1.1 Implementing a Function Module in C/C++

For each function module that the server program is expected to process, you have to implement a C function of type `RFC_SERVER_FUNCTION`. This function has the following prototype:

```
RFC_RC SAP_API myFunctionModule(  RFC_CONNECTION_HANDLE rfcHandle,
                                  RFC_FUNCTION_HANDLE funcHandle,
                                  RFC_ERROR_INFO* errorInfoP)
```

The implementation of this function has to perform the following steps:

1. Read the IMPORTING, CHANGING and TABLES parameters, which the SAP system sent to the RFC server program, from the given `RFC_FUNCTION_HANDLE`. For details on how this is done, please see chapter 3.2 *Accessing the Values of a Data Container*.

2. Execute the “business logic”, that is: given the input values received from the SAP system, perform the necessary steps to compute the output values that are to be sent back to the SAP system (if any).
3. Fill the output values into the corresponding EXPORTING, CHANGING and TABLES parameters of the given `RFC_FUNCTION_HANDLE`. For details on how this is done, please see again chapter 3.2 *Accessing the Values of a Data Container*.
   Note that in most cases the server program will never need to actually create or delete a data container in the course of processing a function module request: the RFC library has already created most containers when it received the data from the SAP system, or it will create other (output) containers on the fly, when the program first accesses them. And the RFC library will automatically destroy all these containers, after the server function implementation has returned and the response data it produced has been sent back to the SAP system.
4. If during any of the previous steps an error occurs, the function module implementation can abort processing and raise any of the error conditions that a function module implemented in ABAP could raise: a **SYSTEM_FAILURE**, if a severe technical problem happens (for example, a database or other technical component that the function module needs to perform its work, is currently down), an **ABAP Message** of type E, X or A, if some kind of severe error on application level occurs, or an **ABAP Exception**, if an ordinary application error occurs (for example, one of the input parameters provided by the SAP system is not in the expected range). Of course, this must be one of the ABAP Exceptions defined by the function module in its EXCEPTIONS clause.
   For each of these possible error conditions, the server program code has to fill the required fields of the provided `errorInfoP` structure and then return the corresponding return code from your server function implementation as indicated by the following table:

Table 5-A - Raising Error Conditions from C Function Module Implementations

Error Condition	ERROR_INFO Fields to be Filled	Return Code	Effect on Connection	Effect in the Backend
SYSTEM_FAILURE	message	`RFC_EXTERNAL_FAILURE`	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.
SYSTEM_FAILURE with SY-Parameter	message abapMsgType abapMsgClass abapMsgNumber abapMsgV1-V4	`RFC_EXTERNAL_FAILURE`	Is closed	As above plus the fields SY-MSGTY, SY-MSGID, SY-MSGNO, and SY-MSGV1-V4 are filled.
ABAP Message	abapMsgType abapMsgClass abapMsgNumber abapMsgV1-V4	`RFC_ABAP_MESSAGE`	Is closed	SY-SUBRC is set corresponding to the SYSTEM_FAILURE key in the EXCEPTIONS clause and the fields SY-MSGTY, SY-MSGID, SY-MSGNO, and SY-MSGV1-V4 are filled.
ABAP Exception	key	`RFC_ABAP_EXCEPTION`	Remains Open	SY-SUBRC is set corresponding to the exception key in the EXCEPTIONS clause.

ABAP Exception with SY-Parameter	key abapMsgType abapMsgClass abapMsgNumber abapMsgV1-V4	`RFC_ABAP_EXCEPTION`	Remains Open	As above plus the fields SY-MSGTY, SY-MSGID, SY-MSGNO, and SY-MSGV1-V4 are filled.

The server application should make extensive use of these features, because the more error details it sends back to the ABAP side, the easier it is going to be to trouble-shoot problems when they occur. In general, a good error handling is important for developing a robust interface between two remote components.

Once we get to the point in the next sections where we have a working RFC server sample program, we will also provide some sample ABAP code to illustrate the last column of the above table, i.e. how ABAP can react to error messages returned by the C/C++ server program. See chapter 5.7 *Error Handling on ABAP Side When Calling an External RFC Server*.

For some more background on these steps and how they interact with the NW RFC library, please see chapter 3.4 *Data Flow in RFC Server Programs*.

5.1.2 Installing the Function Module Implementation for the RFC Library to Use

Next the application needs to obtain an `RFC_FUNCTION_DESC_HANDLE` for the function module the server is supposed to process. This can be done in any of the various ways described in chapter 2 *Obtaining Metadata Descriptions*.

This function description then has to be installed together with the function pointer pointing to the `RFC_SERVER_FUNCTION` implemented in the previous chapter. Sample code for this could look as follows:

```
RFC_RC SAP_API myFunctionModule( RFC_CONNECTION_HANDLE rfcHandle,
                                 RFC_FUNCTION_HANDLE funcHandle,
                                 RFC_ERROR_INFO* errorInfoP) {

   // Implement functionality of function module MY_FUNCTION_MODULE.

   return RFC_OK;

}

RFC_FUNCTION_DESC_HANDLE createFuncDesc() {

   // Create or obtain function description for MY_FUNCTION_MODULE
   // from somewhere, as described in chapter 2.

}

int mainU(int argc, SAP_UC** argv) {

   RFC_ERROR_INFO errorInfo;
   RFC_FUNCTION_DESC_HANDLE myFuncDesc = createFuncDesc();

   RfcInstallServerFunction(cU("ABC"), myFuncDesc,
                            myFunctionModule, &errorInfo);

   if (errorInfo.code == RFC_OK) {

      // Start listening for incoming calls as described in the
      // following chapters.

   }

   return 0;

}
```

The first parameter in `RfcInstallServerFunction()` specifies the SAP system ID of the backend system from which the application wants to accept function calls. (Note that an RFC server can not only receive requests from the SAP system at whose gateway it is registered, but also from other SAP systems, that may use the gateway of the first SAP system as a remote gateway. For more information on how to allow or prevent this, see the gateway documentation on SAP Help Portal, in particular the description of the *sec_info* and *reg_info* files, depending on whether the server is a started or registered server: SAP Gateway Security Files)

If the server program sets this parameter to NULL, the RFC library uses the given server function for processing incoming RFC requests from any SAP system.

Instead of (or in addition to) providing one `RFC_SERVER_FUNCTION` implementation for every function module, the server program needs to be able to handle, it can also provide one generic function, that can handle all function modules (or to be more precise: all function modules, for which no explicit `RFC_SERVER_FUNCTION` has been installed). This allows more flexibility, but also requires a bit more work: as the program cannot know in advance, which different function modules the backend system may call, it cannot provide the necessary `RFC_FUNCTION_DESC_HANDLE`s for all of them. Consequently, the function descriptions must be created/obtained on the fly, when a request for a certain function module first comes in. This is achieved by providing an additional function pointer to a callback function whose responsibility it is to provide a function description for any incoming call. This callback function must be of type `RFC_FUNC_DESC_CALLBACK`. Let's put this into code again:

```
RFC_RC SAP_API genericHandler(RFC_CONNECTION_HANDLE rfcHandle,
                              RFC_FUNCTION_HANDLE funcHandle,
                              RFC_ERROR_INFO* errorInfoP) {

   RFC_FUNCTION_DESC_HANDLE funcDesc;
   RFC_ABAP_NAME functionName;

   funcDesc = RfcDescribeFunction(funcHandle, errorInfoP);

   RfcGetFunctionName(funcDesc, functionName, errorInfoP);

   // Check whether functionName is a function module we know how
   // to handle and handle the other ones generically.

   return RFC_OK;

}

RFC_RC SAP_API metadataLookup(SAP_UC const* functionName,
                              RFC_ATTRIBUTES rfcAttributes,
                              RFC_FUNCTION_DESC_HANDLE* funcDescHandle) {

   RFC_RC rc = RFC_NOT_FOUND;

   if (weWantToHandle(functionName)) {

      // Create or obtain function description for functionName
      // from somewhere, as described in chapter 2.
      // Use rfcAttributes.sysId or rfcAttributes.partnerRel in
      // order to make sure we provide the correct metadata, as
      // function modules or structures may be defined differently
      // in different backend releases.
      rc = RFC_OK;

   }

   return rc;

}
```

```
int mainU(int argc, SAP_UC** argv) {
   RFC_ERROR_INFO errorInfo;

   RfcInstallGenericServerFunction(genericHandler,
                                   metadataLookup, &errorInfo);

   if (errorInfo.code == RFC_OK) {

   // Start listening for incoming calls as described in the
      // following chapters.

   }

   return 0;

}
```

Now, when the RFC library receives an RFC request from an SAP system, it performs the following steps:

1. First it determines the system ID of the calling SAP system and the name of the requested function module.
2. Then it checks, whether for this combination of system ID plus function name a pair of `RFC_FUNCTION_DESC_HANDLE` and `RFC_SERVER_FUNCTION` has been installed via `RfcInstallServerFunction()`. If yes, it uses the function description to decode the received data, creates an `RFC_FUNCTION_HANDLE` from the data and then calls the given server function, passing the function handle as input.
3. If no, it checks whether for this function name a “global” (meaning sysID = NULL) pair of `RFC_FUNCTION_DESC_HANDLE` and `RFC_SERVER_FUNCTION` has been installed via `RfcInstallServerFunction()`. If yes, it uses the function description to decode the received data, creates an `RFC_FUNCTION_HANDLE` from the data and then calls the given server function, passing the function handle as input.
4. If no, it checks whether a pair of `RFC_SERVER_FUNCTION` and `RFC_FUNC_DESC_CALLBACK` has been installed via `RfcInstallGenericServerFunction()`. If no, it sends a SYSTEM_FAILURE back to the SAP system, indicating that the requested function module could not be found.
5. If yes, it calls the `RFC_FUNC_DESC_CALLBACK`, passing the function name and an attributes structure that identifies the backend system.
6. If that function description callback is able to come up with a function description of the requested function module suitable for the current backend system, it uses the function description to decode the received data, creates an `RFC_FUNCTION_HANDLE` from the data and then calls the generic server function, passing the function handle as input.
7. If not, it sends a SYSTEM_FAILURE back to the SAP system, indicating that the requested function module could not be found.

Note: Passing NULL as the system ID in `RfcInstallServerFunction()` is convenient, if the server program shall serve several different backend systems and it shall provide a single implementing function for them all.
A word of warning, however: Different backend releases may have different metadata 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 its parameters (for example, from CHAR30 to CHAR50).
If the server application fetches the metadata description from the older backend, registers it globally via sysID=NULL, and then a call comes in from the newer backend, the data received from the backend would be truncated, because it wouldn't fit into the data container created from the old metadata description.
If SAP function modules are changed between releases, then it is guaranteed that this is done in a backwards compatible way: parameters are only enlarged, never

shortened, and new fields of structures/tables are added only at the end, never in the middle.
This means, it is safer to obtain your metadata descriptions from the backend with the latest release. Then, when communicating with an older release, a few extra fields or enlarged fields won't hurt: the extra memory in the data container will just remain empty. (You only need to make sure that when the server is sending data to the older system, that the returned data is still meaningful without the extra fields, as the backend will truncate them. Also, if fields have been added in the middle of a structure, this will definitely lead to data corruption.)

5.1.3 Restricting Access to the Server

If you want to restrict access to the server program or to just a few critical function modules it exposes, you can do so in several different ways. First, you can build it right into the server function implementations. This works best, if there are only a few function modules you want to protect against unauthorized access. From the connection object that is passed into your server function, the server program can obtain the RFC attributes and from these it can get information like the system ID of the calling SAP system, the client and the user name of the SAP user that triggered the call. For example:

```
RFC_RC SAP_API myFunctionModule( RFC_CONNECTION_HANDLE rfcHandle,
                                 RFC_FUNCTION_HANDLE funcHandle,
                                 RFC_ERROR_INFO* errorInfoP) {

   RFC_ATTRIBUTES rfcAttributes;

   RfcGetConnectionAttributes(rfcHandle, &rfcAttributes, errorInfoP);
   if (errorInfoP->code != RFC_OK)
       return errorInfoP->code;

   if (strcmpU(rfcAttributes.sysId, cU("ABC"))  ||
       strcmpU(rfcAttributes.client, cU("800")) ||
       strcmpU(rfcAttributes.user, cU("ADMIN")) ) {

      // Only user ADMIN from system ABC, client 800 may execute
      // this function module.

      errorInfoP->code = RFC_EXTERNAL_FAILURE;
      strcpyU(errorInfoP->message, cU("Access denied"));
      return errorInfoP->code;

   }

   // Implement functionality of function module MY_FUNCTION_MODULE.

   return RFC_OK;

}
```

If there are many function modules to which you want to apply access control checks, it would be tedious to implement this over again in each server function. In this case you can implement another C callback function of type `RFC_ON_AUTHORIZATION_CHECK` and install it with the RFC library. Then, whenever a function call arrives at the RFC library, it will first invoke this callback function with a wide variety of information about the caller, and the callback can then decide whether it wants to accept this particular call or not. In terms of code this may look like follows:

```
RFC_RC SAP_API authorizationHandler(RFC_CONNECTION_HANDLE rfcHandle,
                          RFC_SECURITY_ATTRIBUTES* secAttributes,
                          RFC_ERROR_INFO* errorInfo){

   // Here you can check any combination of function name, system ID,
   // client, calling ABAP report and username, as well as the
   // backend's SNC credentials.
```

```
    if (!strcmpU(secAttributes->user, cU("ADMIN"))) {

        // User ADMIN is allowed to call function modules FUNCTION_1,
        // ... FUNCTION_3.

        if (!strcmpU(secAttributes->functionName, cU("FUNCTION_1")) ||
             !strcmpU(secAttributes->functionName, cU("FUNCTION_2")) ||
             !strcmpU(secAttributes->functionName, cU("FUNCTION_3")))

            return RFC_OK;

    }

    else if (!strcmpU(secAttributes->user, cU("BUSINESS_USER"))) {

        // User BUSINESS_USER is allowed to call function modules
        // BAPI_A, and BAPI_B.

        if (!strcmpU(secAttributes->functionName, cU("BAPI_A")) ||
            !strcmpU(secAttributes->functionName, cU("BAPI_A")))

            return RFC_OK;

    }

    // All other cases are denied.
    return RFC_AUTHORIZATION_FAILURE;

}


int mainU(int argc, SAP_UC** argv) {
    RFC_ERROR_INFO errorInfo;

    RfcInstallAuthorizationCheckHandler(authorizationHandler, &errorInfo);

    if (errorInfo.code != RFC_OK){

        // Probably an out-of-memory situation.
        return 1.

    }

    // Remaining server setup

    return 0;

}
```

A word of warning: the RFC library passes the current `RFC_CONNECTION_HANDLE` into the `RFC_ON_AUTHORIZATION_CHECK` function. At this point of time, the server application shall not use the connection for anything that would cause a read or write operation on it, because there is currently an unfinished RFC request pending on this connection, and any read or write would corrupt the call or even crash the program. It is ok, however, to get the `RFC_ATTRIBUTES` from the connection, in case the application needs more information about the caller beyond what is provided in the `RFC_SECURITY_ATTRIBUTES`.

If the RFC connection is secured with SNC, you can additionally implement to verify the backend system's SCN name using the `sncAclKey/sncAclKeyLength` members of the `RFC_SECURITY_ATTRIBUTES`. The necessary steps for this are:

- Convert the expected/allowed backend's SCN name to binary format using `RfcSNCNameToKey()` and store it (and its length) in static memory.
- In the authorization handler make a binary comparison of that key with the one given in `sncAclKey/sncAclKeyLength`.
- If they don't match, return `RFC_AUTHORIZATION_FAILURE`.

Note: Don't use the human readable SNC name for comparisons, as this string is not unique. For example the string representations
```
p:CN=ABC, OU=IT Department, O=MyCompany, C=DE
p:CN = ABC, OU = IT Department, O = MyCompany, C = DE
p/secude:CN=ABC, OU= IT Department, O=MyCompany, C=DE
```
are all valid representations of the same SNC name. If the backend uses a slightly different format than the SNC library on your local frontend, the comparisons may fail, even if the SNC names match.

The above two possibilities must be built into the server program right from the start, but what if you want to secure an existing program whose source code cannot be changed at this time? Even in this case something can be done.

The first option is to use the gateway security features. Modify the *reg_info* file of the gateway to which the RFC server is going to register. Here you can add rules that specify exactly, which application servers and which SAP users are allowed to access this server program. Unfortunately, this is an "all-or-nothing" decision: you cannot restrict access on a function module level here. See again the gateway documentation at SAP Gateway Security Files

The second option is to provide additional connection parameters when starting the server. (Make sure that the server program does ***not*** hardcode these in the source as this violates the concept of separation of concerns. It is very important that any RFC program's connection parameters can be changed on the fly by an administrator and are not defined by the developer, if only to be able to activate RFC trace for trouble-shooting a problem that pops up.) See chapter 4.1 *Managing Login Information* for more information. It talks mainly about RFC client programs, but most of it applies to RFC server programs as well. In this case you can add the following two parameters to the set of connection parameters:

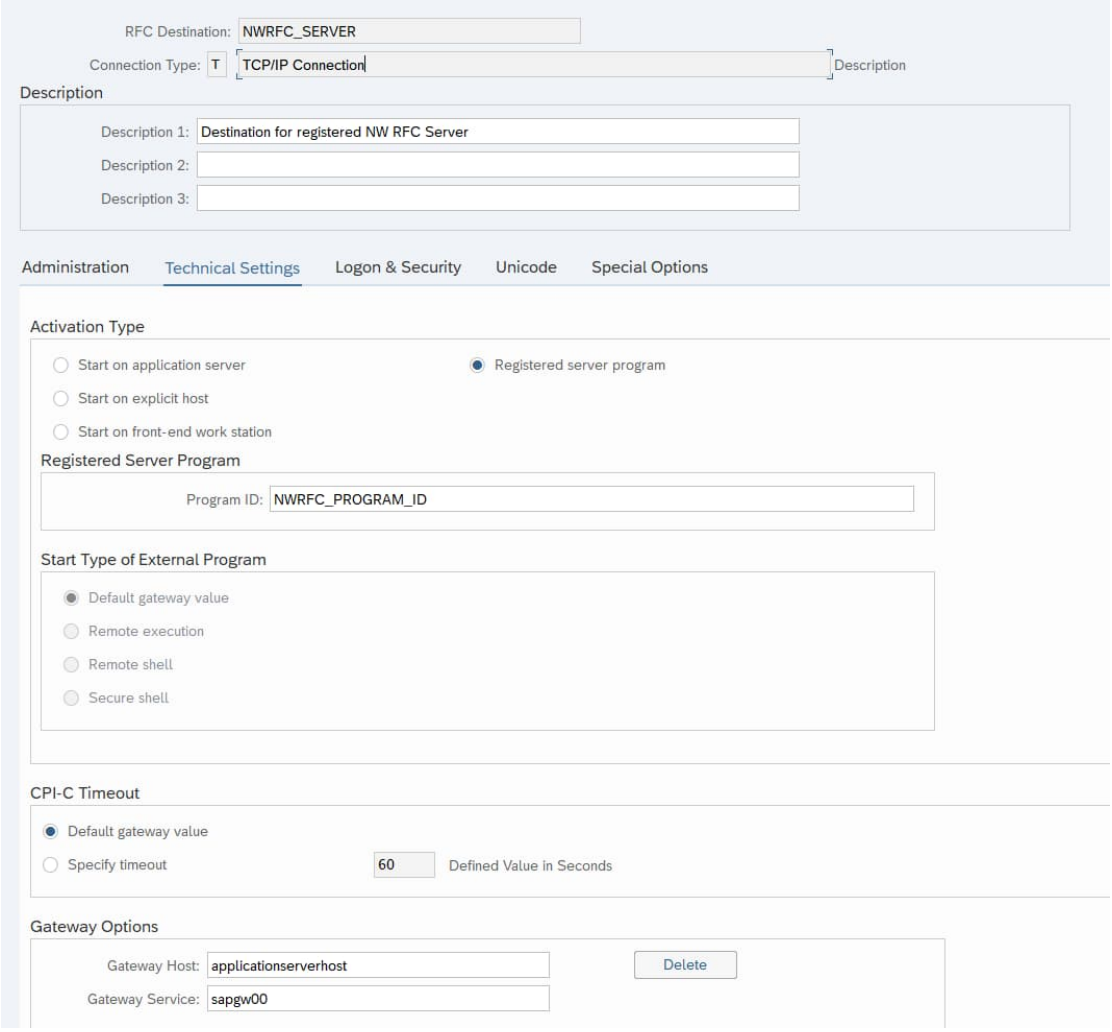
- SYS_IDS: here you can provide a list of SAP system IDs of the systems, from which the server program shall accept RFC calls. Calls from any other SAP system will be rejected, if this parameter is specified. The list must be separated by ',' characters.
  Example:
  ```
  SYS_IDS = ABC,H9U,XYZ
  ```
- SNC_PARTNER_NAMES: here you can provide a list of SNC names identifying the systems, from which the server program shall accept RFC calls. The RFC library will convert the SNC names to the unique SNC ACL keys and compare with the keys of the backends trying to send an RFC call to this server. This provides additional security as SNC names are validated using cryptographic algorithms. The list must be separated by '|' characters.
  Example:
  ```
  SNC_PARTNER_NAMES = p:CN=ABC, O=MyCompany, C=DE|p:CN=H9U,
  O=MyCompany, C=DE|p:CN=XYZ, O=MyCompany, C=DE
  ```

After these tasks that are common to all RFC servers, let's now discuss the three possible types of RFC servers.

5.2 Registered RFC Server

5.2.1 Setting up the RFC Destination

Before you can use a registered RFC server, you need to create an RFC destination on ABAP system side. This can be done in the configuration of RFC connections (transaction SM59). Make sure to choose connection type ‘T’ and activation type “Registered Server Program”. The complete setup may look as follows:



Here the gateway options are particularly interesting, as they can be used to setup load balancing among different RFC server programs or different application servers:

- If you leave gateway host and gateway service empty, every application server of the SAP system will check its own gateway, when sending an RFC request to this destination. Thus, by registering one server program (or one thread) at each application server, you can setup a kind of one-to-one relationship between application server and its corresponding server program. (1-1 relationship, or 1-n relationship, if n programs/threads are registered at each of the application servers.) This approach has the advantage, that, if one application server/gateway is shutdown, the remaining application servers can still use the external program.
- If you fill these options and register one program/thread at the specified gateway host, all application servers will use this one server program. This will of course cause problems, if two or more application servers try and send RFC requests to the server at the same time. If the requests take longer than the CPI-C timeout, some of the app servers may run into a timeout error, before it’s their turn to send an RFC request. (m-1 relationship.)

- If you fill these options and register multiple programs/threads at the specified gateway host, multiple application servers can access the same set of RFC server programs at the same time, and the gateway will perform load balancing, distributing the load evenly over the registered programs/threads. (m-n relationship.)

The value for the parameter “Program ID” can be chosen freely. However, no matter how you set up this destination, you have to make sure that the server program uses the exact same values in its `PROGRAM_ID`, `GWHOST` and `GWSERV` parameters. (Or if you leave the gateway options empty in the configuration of RFC connections (transaction SM59), the server program has to specify one of the existing application server hosts in its parameters. The gateway service is usually “sapgwXX”, where XX is the instance number of the application server. The server program then is accessible only from this specific application server.)

- **Note:** 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 at the same RFC destination. In that case, half the RFC calls (or IDocs) go to the production instance and get processed 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.

A registered RFC server can be implemented in two different ways. The original way that had to be used up to release 7.20, requires setting up a dispatch loop and managing the registered server connections manually. Starting with release 7.50, however, a simplified approach is possible: a JCo-like server object that only needs to be started and then manages the dispatch loop and the re-registration of broken server connections automatically. We start with describing this new approach.

5.2.2 Managed RFC Server

5.2.2.1 Main Server Program

After setting up the above preparations, starting to listen for incoming RFC calls is now very easy. You just need to create a server handle with the right connection parameters and start it. When you are done processing RFC calls or want to pause for a while, you can simply stop the server. Here is some sample coding for this:

```
int mainU(int argc, SAP_UC** argv) {

   // Preliminary steps omitted.

   RFC_CONNECTION_PARAMETER serverParams;
   RFC_ERROR_INFO errorInfo;
   RFC_SERVER_HANDLE myServer = NULL;


   serverParams.name = cU("DEST");
   serverParams.value = cU("MyServer");

   myServer = RfcCreateServer(&serverParams, 1, &errorInfo);
   if (errorInfo.code != RFC_OK)
      goto errorHandling;

   RfcLaunchServer(myServer, &errorInfo);
   if (errorInfo.code != RFC_OK)
      goto errorHandling;

   printfU(cU("Server is up and running. Press \"s\" to stop.\n"));


   while (1) {
      SAP_UC c = (SAP_UC)fgetcU(stdin);
```

```
        if (c == cU('s')) {
            RfcShutdownServer(myServer, 60, NULL);
            RfcDestroyServer(myServer, NULL);

            return 0;
        }
    }

    errorHandling:
    // Log the error

    if (myServer)
        RfcDestroyServer(myServer, NULL);

    return 1;

}
```

Two points are worth noting here:

1. This example expects the necessary parameters for the server to be maintained in the *sapnwrfc.ini* file as follows:

   ```
   DEST=MyServer
   SERVER_NAME=A descriptive name
   PROGRAM_ID=As defined in destination configuration (SM59)
   GWHOST=Gateway hostname as defined in SM59
   GWSERV=Gateway service as defined in SM59
   REG_COUNT=5
   ```

   The parameters `PROGRAM_ID`, `GWHOST`, and `GWSERV` are the standard RFC server parameters as described in the previous section 5.2.1. The `REG_COUNT` parameter can be used to set up load balancing as also described in that section. The RFC library starts `REG_COUNT` threads and registers one connection for each thread. The backend system can then send multiple RFC requests from different application servers (or work processes) in parallel.

   When using the managed server, you can also use a feature called group registration. Replace the two parameters `GWHOST` and `GWSERV` with the parameters `MSHOST, SYSID` and `GROUP`. The RFC library will then connect to the message server, obtain the list of application servers contained in the given group and register `REG_COUNT` connections on each application server belonging to that system. This feature is very useful for the mechanism described in the first point of 5.2.1, i.e. the 1-1 or 1-n relationship of application server to registered program. With the managed server, you can set up this mechanism without any additional programming effort, just by adding two connection parameters.
   Note that this feature is not available for the dispatch-loop based server.

2. The second parameter in `RfcShutdownServer()` is a timeout value indicating how many seconds “grace period” you want to give currently ongoing RFC calls. It is possible, that at the same time the application calls `RfcShutdownServer(),` some of the RFC library’s worker threads may be in the middle of processing an RFC call. If the server program sets the timeout to 0, the server stops immediately and aborts all ongoing calls. The waiting work processes on backend side will also abort with an error. But by setting a timeout of a couple of seconds or minutes (depending on how long the processing of the function modules usually takes) the program can delay the shutdown of the server so that all currently ongoing requests have a chance to complete. In any case, upon calling `RfcShutdownServer()`, the server immediately stops accepting new requests.

As you can see, compared to the dispatch loop approach (described later in chapter 5.2.3), the managed server already does a lot of work for you. Basically, you just concentrate on implementing the business logic of your function modules, and the RFC library takes care of the technical details like registering the required connections at the gateway(s), checking for network errors, gateway shutdowns, etc. and trying periodic reconnect attempts.

However, even though you don't need to implement anything yourself for handling these technical problems (as you would in the dispatch loop server), you may still be interested in getting information about the current status of your server program, for example, whether connectivity to the gateway is currently ok, how many connections are busy, what function modules are currently being processed, etc. For this purpose, there are several additional tools you can use:

5.2.2.2 Status and Error Listeners

You can register a status listener and an error listener. These are callback functions that the RFC library invokes, whenever the server changes its state (for example, a gateway comes back online, and the server can start listening again) or whenever a network error interrupts a connection. This information may be useful for sending alarms to an administrator and fixing problems as soon as possible. The application needs to implement a function of type `RFC_SERVER_STATE_CHANGE_LISTENER` and/or `RFC_SERVER_ERROR_LISTENER`. Let's look at an example:

```
void SAP_API stateChangeListener(RFC_SERVER_HANDLE serverHandle,
                                 RFC_STATE_CHANGE* stateChange) {

   RFC_SERVER_ATTRIBUTES attributes;
   RFC_ERROR_INFO errorInfo;

   RfcGetServerAttributes(serverHandle, &attributes, &errorInfo);

   printfU(cU("State for server %s changed from %s to %s\n"),
               attributes.serverName,
               RfcGetServerStateAsString(stateChange->oldState),
               RfcGetServerStateAsString(stateChange->newState));

   if (stateChange->newState == RFC_SERVER_BROKEN) {

      // Tell someone

   }
}


void SAP_API errorListener(RFC_SERVER_HANDLE serverHandle,
             RFC_ATTRIBUTES* clientInfo, RFC_ERROR_INFO* errorInfo) {

   RFC_SERVER_ATTRIBUTES attributes;
   RfcGetServerAttributes(serverHandle, &attributes, NULL);


   printfU(cU("Error in server %s: %s: %s\n"), attributes.serverName,
               errorInfo->key, errorInfo->message);

   // An error may happen, while the server is currently idle,
   // or while a client is connected to it. In the first case,
   // clientInfo will be NULL, but in the second case, we get
   // a bit of information about the client, whose connection
   // broke down:

   if (clientInfo) {

      printfU(cU("Client from host %s, system %s, client %s,
                                       report %s, user %s\n"),
                  clientInfo->partnerHost,
```

```
                    clientInfo->sysId,
                    clientInfo->client,
                    clientInfo->progName,
                    clientInfo->user);

    }
    else {
        printfU(cU("No client currently connected.\n"));
    }
}


int mainU(int argc, SAP_UC** argv) {

    RFC_ERROR_INFO errorInfo;
    RFC_SERVER_HANDLE myServer = NULL;

    // Preliminary steps omitted.

    RfcAddServerStateChangedListener(myServer, stateChangeListener,
                                     &errorInfo);
    if (errorInfo.code != RFC_OK)
        goto errorHandling;

    RfcAddServerErrorListener(myServer, errorListener, &errorInfo);
    if (errorInfo.code != RFC_OK)
        goto errorHandling;

    RfcLaunchServer(myServer, &errorInfo);
    if (errorInfo.code != RFC_OK)
        goto errorHandling;

    printfU(cU("Server is up and running. Press \"s\" to stop.\n"));

    while (1) {
        SAP_UC c = (SAP_UC)fgetcU(stdin);

        if (c == cU('s')) {
            RfcShutdownServer(myServer, 60, NULL);
            RfcDestroyServer(myServer, NULL);

            return 0;
        }
    }

    errorHandling:
    // Log the error

    if (myServer)
        RfcDestroyServer(myServer, NULL);

    return 1;

}
```

5.2.2.3 Server Connection Monitor

Another interesting feature that may be useful for performance monitoring and investigating sizing questions, is the server monitor. The best way to use it, is probably to set up a separate thread and in it execute a routine like the following:

```
RFC_SERVER_HANDLE myServer;
int serverIsRunning;

RFC_ERROR_INFO errorInfo;
RFC_SERVER_MONITOR_DATA* monitorData;
unsigned numConnections;
SAP_UC timeBuf[26] = iU("");

while (serverIsRunning) {
#ifdef WIN32
   Sleep(60000);
#else
   sleep(60);
#endif
   RfcGetServerConnectionMonitorData(myServer, &numConnections,
                                     &monitorData, &errorInfo);

   if (errorInfo->code != RFC_OK)
      continue;

   for (unsigned k = 0; k < numConnections; ++k) {

      if (monitorData[k].isActive) {

         printfU(cU("Connection %s from %s/%s is currently processing
                        FM %s. IsStateful = %d\n"),
                     monitorData[k].clientInfo->cpicConvId,
                     monitorData[k].clientInfo->partnerHost,
                     monitorData[k].clientInfo->user,
                     monitorData[k].functionModuleName,
                     monitorData[k].isStateful);

      }
      else {

#ifdef WIN32

         _wctime_s(timeBuf, sizeofU(timeBuf),
                   &(monitorData[k].lastActivity));

#else

         char utf8TimeBuf[26] = "";
         ctime_s(utf8TimeBuf, sizeof(utf8TimeBuf),
                 &(monitorData[k].lastActivity));

         RfcUTF8ToSAPUC(utf8TimeBuf, sizeof(utf8TimeBuf), timeBuf,
                        sizeofU(timeBuf), NULL, &errorInfo);

#endif

         timeBuf[24] = cU('.');

         printfU(cU("Connection %s from %s/%s currently idle. Last
                activity: %s IsStateful = %d\n"),
                monitorData[k].clientInfo->cpicConvId,
                monitorData[k].clientInfo->partnerHost,
                monitorData[k].clientInfo->user,
                timeBuf,
                monitorData[k].isStateful);
```

```
        }
    }
    RfcDestroyServerConnectionMonitorData(numConnections,
                                          monitorData, &errorInfo);
    printfU(cU("\n"));
}
```

5.2.2.4 Stateful Server

We will complete the chapter about the managed RFC server with a feature that is not (or only rudimentarily) available in the old dispatch-loop server: stateful connections. Contrary to RFC client applications, where every connection is stateful by default, in an RFC server application a connection is stateless by default. In some situations, however, it may be useful to have stateful server connections. A stateful connection is distinguished by two qualities:

1. There is a one-to-one relationship between the ABAP system user session on backend side and the RFC connection to the external program. Or in other words: whereas usually a connection to an external server program is acquired by a user session for just one RFC call and afterwards released again, so that any user session/app server/work process in the system may use it for the next RFC call, a *stateful* server connection is bound to a particular ABAP system user session for *exclusive use*.
2. The external program can keep state information in between different RFC requests. For example, one function module may produce intermediate results and store it in the session memory, and a later request to another (or the same) function module may continue using these intermediate results. If there are multiple different user sessions executing these function modules in parallel, each user session will have its own separate copy of those intermediate results.

How can a registered server connection be turned stateful? Basically, there are two different ways to achieve this:

1. The calling ABAP program may request a connection for exclusive use by executing the function module `RFC_SET_REG_SERVER_PROPERTY` with parameter `EXCLUSIV = 'Y'`. The RFC library will then reserve one connection exclusively for this ABAP system user session. (This works already with the old dispatch-loop approach.)
2. The external C program can decide to turn a connection stateful by calling `RfcSetServerStateful()` with `isStateful = 1` on the connection handle. This must be done inside an `RFC_SERVER_FUNCTION` implementing one of your function modules. For example, you could have one “entrance” function module that is always called first by the ABAP application, followed by a sequence of other function calls and finished with a function module that ends the stateful session again. This works only for the managed RFC server approach.

How long does the stateful session live inside the external server program? A stateful session will be terminated by one of the following events:

1. The calling ABAP program may end the stateful session by executing the function module `RFC_SET_REG_SERVER_PROPERTY` with parameter `EXCLUSIV = 'N'`.
2. The corresponding ABAP system user session ends, for example, the user logs off or ends the current internal mode by executing /n in the T-code field or by pressing the Exit button (Shift-F3) or Cancel button (F12).
3. The external C program ends the stateful connection by calling `RfcSetServerStateful()` with `isStateful = 0` from inside a function module implementation.
4. The network connection breaks down due to network problems or due to a SYSTEM_FAILURE.

In order to turn an ordinary server program into a stateful server, two tasks have to be added:

1. You need to start and end the stateful sessions, either on ABAP side using `RFC_SET_REG_SERVER_PROPERTY`, or on C/C++ side using `RfcSetServerStateful()`.
2. You need to manage your session data, i.e. create a copy of the session data for each session that is started, make the data available for function module implementations running inside the session, and delete the data again, when the session ends. For this task a callback function of type `RFC_SERVER_STATE_CHANGE_LISTENER` can be used.

Let's look at a small example to illustrate how this can work in practice. In this example, the "session data", of which the server program has to keep one separate copy for each active user session, is just a simple `int`, in which it counts the number of function calls received during the lifetime of the current user session. But of course, this could be anything your application may need to keep track off on a per user basis.

```
#include "sapnwrfc.h"
#include <map>
using namespace std;

class SessionData {
   public:
      SessionData() : callCount(0){};
      int callCount;
};

// Note: on non-windows, it is necessary to convert the SAP_UC-based
// session ID to UTF-8 (char) and use the standard string class as key.
static map<wstring, SessionData*> sessions;


extern "C" void SAP_API sessionChangeListener(
                       RFC_SERVER_HANDLE serverHandle,
                       RFC_SESSION_CHANGE* sessionChange) {
   RFC_SERVER_ATTRIBUTES attributes;
   RFC_ERROR_INFO errorInfo;

   RfcGetServerAttributes(serverHandle, &attributes, &errorInfo);
   printfU(cU("Session %s for server %s changed to %s\n"),
                sessionChange->sessionID,
                attributes.serverName,
                RfcGetSessionEventAsString(sessionChange->event));

   wstring sessionID = sessionChange->sessionID;
   SessionData* sessionData;

   switch (sessionChange->event) {
      case RFC_SESSION_CREATED:
         sessionData = new SessionData();
         sessions[sessionID] = sessionData;
         break;
      case RFC_SESSION_ACTIVATED:
         break; //Ignore
      case RFC_SESSION_PASSIVATED:
         break; //Ignore
      case RFC_SESSION_DESTROYED:
         sessionData = sessions[sessionID];
         sessions.erase(sessionID);
```

```
            delete sessionData;
            break;
        default:;
            // Cannot happen.
    }
}


extern "C" RFC_RC stfcConnection(RFC_CONNECTION_HANDLE conn,
                    RFC_FUNCTION_HANDLE function, RFC_ERROR_INFO* errorInfo) {
    SAP_UC req_text[256], resp_text[256];
    wstring sessionID;
    SessionData* sessionData = NULL;
    RFC_SERVER_CONTEXT context;

    RfcGetString(function, cU("REQUTEXT"), req_text, sizeofU(req_text),
                    NULL, errorInfo);
    printfU(cU("Received call for STFC_CONNECTION. Value of REQUTEXT: %s\n"),
                    req_text);

    RfcGetServerContext(conn, &context, errorInfo);
    if (errorInfo->code != RFC_OK)
        return errorInfo->code;

    // Set connection stateful on first incoming call.
    if (!context.isStateful)
        RfcSetServerStateful(conn, 1, errorInfo);

    sessionID.assign(context.sessionID);
    sessionData = sessions[sessionID];
    sessionData->callCount++;
    sprintfU(resp_text, cU("This is call number %d for session %s"),
                            sessionData->callCount, context.sessionID);

    printfU(cU("%s\n\n"), resp_text);

    RfcSetString(function, cU("ECHOTEXT"), req_text,
                    strlenU(req_text), errorInfo);
    RfcSetString(function, cU("RESPTEXT"), resp_text,
                    strlenU(resp_text), errorInfo);

    // After 10 calls, end stateful session.
    if (sessionData->callCount == 10)
        RfcSetServerStateful(conn, 0, errorInfo);

    return RFC_OK;
}


int mainU(int argc, SAP_UC** argv) {
    RFC_RC rc = RFC_OK;
    RFC_CONNECTION_PARAMETER loginParams[6];
    RFC_ERROR_INFO errorInfo;
    RFC_FUNCTION_DESC_HANDLE functionDesc;
    RFC_PARAMETER_DESC paramDesc;

    functionDesc = RfcCreateFunctionDesc(cU("STFC_CONNECTION"), &errorInfo);
    paramDesc = { cU("REQUTEXT"), RFCTYPE_CHAR, RFC_DIRECTION::RFC_IMPORT,
                    255, 510, 0, 0, cU(""), cU(""), 0, 0};
```

```
RfcAddParameter(functionDesc, &paramDesc, &errorInfo);
paramDesc = { cU("ECHOTEXT"), RFCTYPE_CHAR, RFC_DIRECTION::RFC_EXPORT,
              255, 510, 0, 0, cU(""), cU(""), 0, 0 };
RfcAddParameter(functionDesc, &paramDesc, &errorInfo);
paramDesc = { cU("RESPTEXT"), RFCTYPE_CHAR, RFC_DIRECTION::RFC_EXPORT,
              255, 510, 0, 0, cU(""), cU(""), 0, 0 };
RfcAddParameter(functionDesc, &paramDesc, &errorInfo);

RfcInstallServerFunction(cU("A74"), functionDesc,
                 (RFC_SERVER_FUNCTION)stfcConnection, &errorInfo);
if (errorInfo.code != RFC_OK) {
   printfU(cU("Error installing server function: %s\n%s: %s\n"),
                 RfcGetRcAsString(errorInfo.code), errorInfo.key,
                 errorInfo.message);
   return 1;
}

loginParams[0].name  = cU("gwhost");
loginParams[0].value = cU("host");
loginParams[1].name  = cU("gwserv");
loginParams[1].value = cU("sapgw00");
loginParams[2].name  = cU("tpname");
loginParams[2].value = cU("progID");
loginParams[3].name  = cU("reg_count");
loginParams[3].value = cU("4");
loginParams[4].name  = cU("max_reg_count");
loginParams[4].value = cU("8");
loginParams[5].name  = cU("server_name");
loginParams[5].value = cU("TestServer");


RFC_SERVER_HANDLE myServer =
                   RfcCreateServer(loginParams, 6, &errorInfo);
if (myServer == NULL) {
   printfU(cU("Error creating server: %s\n%s: %s\n"),
             RfcGetRcAsString(errorInfo.code), errorInfo.key,
             errorInfo.message);
   return 1;
}

RfcAddServerSessionChangedListener(myServer,
          (RFC_SERVER_SESSION_CHANGE_LISTENER)sessionChangeListener,
          &errorInfo);

RfcLaunchServer(myServer, &errorInfo);
if (errorInfo.code != RFC_OK) {
   printfU(cU("Error launching server: %s\n%s: %s\n"),
             RfcGetRcAsString(errorInfo.code), errorInfo.key,
             errorInfo.message);
   return 1;
}

printfU(cU("Enter \"s\" to stop the server.\n"));
while (1) {
   SAP_UC c = fgetcU(stdin);
   if (c == cU('s'))
      break;
}
```

```
    RfcShutdownServer(myServer, 0, &errorInfo);
    RfcDestroyServer(myServer, &errorInfo);

    return 0;
}
```

Start this program and then open two or more SAPGui sessions on backend side. Go to the function builder (transaction SE37) in both (or all) of them and call `STFC_CONNECTION` against the corresponding RFC destination at which the program is listening. You will see, that in both SAPGui sessions the server program returns the numbers 1 – 10, independent of the order in which you execute a couple of calls in one SAPGui session or the other. If you close one session via F12 and then go back to the function builder, numbering starts from 1 again.

One slightly annoying fact is, that whenever the backend side ends a session that was started by the server program, the connection is silently closed, and the server program receives a network error as follows:
`RFC_COMMUNICATION_FAILURE: connection closed without message (CM_NO_DATA_RECEIVED)`
Therefore it is best to stick to the following rule for normal operations: the side that starts the session should also be the side ending the session.

5.2.3 Dispatch-Loop RFC Server

Before the managed server was introduced with NW RFC Library release 7.50, applications had to manage their own dispatch loop. In case you need to maintain an older program that still uses this approach, we describe it here as well. Note that this approach requires the application to do more work:

- The application needs to react correctly to a number of return codes and reopen broken registrations.
- If the application wants to register multiple connections in parallel, it needs to manage its own threads.

and provides less functionality:

- Error- and Status-Listener cannot be installed.
- The Connection Monitor is not available.
- Starting a stateful session on the external side via `RfcSetServerStateful()` is not possible.

In order to implement a simple dispatch-loop server program, the same preparations as described in chapters 5.1 *Preparing a Server Program for Receiving RFC Requests* and 5.2.1 *Setting up the RFC Destination* are necessary. After that, the main program then looks as follows.

5.2.3.1 Main Server Program

A simple dispatch-loop server needs to implement the following mechanism:

1. Register a connection at the gateway via `RfcRegisterServer()`. The set of parameters is the usual triple of gateway host, gateway service and program ID, and optionally a few more for activating trace, choosing serialization format and table compression, etc. See the sample *sapnwrfc.ini* file from the SDK installation directory for a complete list of available parameters.
2. In a loop listen for incoming requests via `RfcListenAndDispatch()`. It is recommended to use a finite timeout value here, so that you can check periodically, whether the server is still supposed to run, and can break out of the loop, if the server ought to be stopped.

Inside `RfcListenAndDispatch()` a lot of work is performed, when an RFC request arrives on that connection:

- The access restrictions are checked to see, whether the execution of the function module should be allowed (see 5.1.3)
- A metadata description and a function implementation are searched for (see 5.1.2)
- A data container is created that corresponds to the metadata, and the function module's importing parameters are read from the wire, decoded and stored in the data container
- The function module implementation is called, passing in the data container
- After the function module implementation has finished, the exporting parameters are read from the data container (or some error information from the errorInfo structure), encoded and returned over the wire to the backend system
- The data container is destroyed

3. Whenever `RfcListenAndDispatch()` returns, check it's return code and react appropriately:
   - `RFC_RETRY`: the given timeout period is over, but no request arrived during that time. Just loop over.
   - `RFC_NOT_FOUND`: a request for a function module has arrived that the server application did not know. As this has been reported to the backend via a `SYSTEM_FAILURE`, the connection is now aborted and a fresh one has to be opened via `RfcRegisterServer()`.
   - `RFC_COMMUNICATION_FAILURE`: A network problem has killed the connection and a fresh one has to be opened via `RfcRegisterServer()`. (If there are permanent network problems, the server program should be prepared that this reconnect-attempt might fail as well)
   - `RFC_CLOSED`: The registration was cancelled on backend side, for example, from gateway monitor (transaction SMGW).
   - `RFC_OK` or `RFC_ABAP_EXCEPTION`: a request has been processed successfully or with an "application level exception". Just loop over.
   - `RFC_ABAP_MESSAGE` or `RFC_EXTERNAL_FAILURE`: a request has been processed and ended with a "low level exception". As this aborts the connection, the server program has to open a fresh one via `RfcRegisterServer()`.

   Usually, `RfcListenAndDispatch()` returns the return code that was returned from the function module implementation. The exceptions are of course `RFC_RETRY`, `RFC_NOT_FOUND`, `RFC_CLOSED` and `RFC_COMMUNICATION_FAILURE`.
4. Once the server program shall stop listening for requests, it can simply break out of the loop and close the registration with `RfcCloseConnection()`.

For implementing a multi-threaded server that can handle more than one single request at a time, this mechanism can be set up in multiple threads. However, make sure that each thread gets and uses its own exclusive connection handle, as using the same connection in different threads can corrupt the data or even crash the program.

Let's now put this into some sample code:

```
int isRunning = 1;
int traceLevel = 0;

int mainU(int argc, SAP_UC** argv) {

   /* Set up the usual stuff as before:
      - metadata
      - function module implementations
      - access control */


   RFC_CONNECTION_PARAMETER serverParam;
```

```
   RFC_CONNECTION_HANDLE serverHandle = NULL;
   RFC_ERROR_INFO errorInfo;
   RFC_RC rc;

   serverParam.name = cU("dest"); /* Connection parameters are specified
                                     in sapnwrfc.ini. */
   serverParam.value = cU("MY_SERVER");

   /* Set up another thread that can stop the server by setting
      isRunning to zero, when required. Or you could have a certain
      function module, whose implementation sets isRunning to zero,
      so that the server can be stopped from ABAP system side. */

   printfU(cU("Starting to listen...\n\n"));
   while(isRunning) {
      if (serverHandle == NULL) {
         serverHandle = RfcRegisterServer(&serverCon, 1, &errorInfo);
         if (serverHandle == NULL) {
            printfU(cU("Error Starting RFC Server: %s - %s\n"),
                  RfcGetRcAsString (errorInfo.code),
                  errorInfo.message);
            printfU(cU("Trying to reconnect in 120 seconds.\n"));
#ifdef WIN32
            Sleep(120000);
#else
            sleep(120);
#endif
            continue;
         }
      }
      rc = RfcListenAndDispatch(serverHandle, 120, &errorInfo);
      printfU(cU("RfcListenAndDispatch() returned %s\n"),
               RfcGetRcAsString(rc));
      switch (rc) {
         case RFC_OK: // A function module was processed ok.
            continue;
         case RFC_RETRY:  /* This only notifies, that no request came in
                             within the timeout period.
                             Just continue the loop. */
            printfU(cU("No request within 120s.\n"));
            continue;
         case RFC_ABAP_EXCEPTION: /* The function module implementation
                             has returned RFC_ABAP_EXCEPTION.
                             This is equivalent to an ABAP function module
                             throwing an ABAP Exception. The exception has
                             been returned to the ABAP system and our
                             connection is still open. Just loop over. */
            printfU(cU("ABAP_EXCEPTION in implementing function: %s\n"),
                        errorInfo.key);
            continue;
         case RFC_NOT_FOUND: /* The ABAP system tried to invoke a function
                                module, for which the server did not
                                supply an implementation.
                                The ABAP system has been notified of this
                                through a SYSTEM_FAILURE, so we have to
                                refresh our connection. */
            printfU(cU("Unknown function module: %s\n"),
                        errorInfo.message);
            break;
```

```
            case RFC_EXTERNAL_FAILURE:/* The function module implementation
                                         raised a SYSTEM_FAILURE. In this case the
                                         connection has to be refreshed as well. */
               printfU(cU("SYSTEM_FAILURE has been sent to backend: %s\n"),
                           errorInfo.message);
               break;
            case RFC_ABAP_MESSAGE: /* The function module implementation
                                         raised a SYSTEM_FAILURE. In this case the
                                         connection has to be refreshed as well. */
               printfU(cU("ABAP Message has been sent to backend: %s\n"),
                           errorInfo.message);
               break;
            case RFC_COMMUNICATION_FAILURE: // Network error
               printfU(cU("Connection broke down: %s\n"), errorInfo.message);
               break;
            case RFC_CLOSED: /* Connection has been cancelled manually on
                                         backend side (SMGW). */
               printfU(cU("Connection cancelled: %s\n"), errorInfo.message);
               break;
            default: /* Something pretty low level went wrong. Perhaps a
                   serialization/compression/conversion error. */
               printfU(cU("Low level error: %s (%s)\n"),
                   RfcGetRcAsString(rc), errorInfo.message);
         }
         // If we get here, the connection was broken.
         // The following will trigger a reconnect attempt at the beginning
         // of the while loop:
         serverHandle = NULL;
      }

      if (serverHandle)
         RfcCloseConnection(serverHandle, NULL);

      return 0;
}
```

5.3 Started RFC Server

In contrast to a registered server, which is up and running all the time, a started server may be used in situations, where the server program is needed only occasionally, and it may not be worth it to have it running constantly.

In that case, whenever a function call is executed against this destination, the backend (the gateway process to be more precise) starts the RFC program, the RFC program then opens a connection to the gateway, and the gateway routes the waiting RFC call over this connection. Usually a started RFC server program processes just this one RFC call and then closes the connection again and ends itself. However, it is also possible to write started servers that keep the connection open and accept multiple function calls before ending themselves.

5.3.1 Setting up the RFC Destination

In order to use a started server, the RFC destination needs to be setup a bit differently compared to the registered server. Again, you create a destination of type 'T' in the configuration of RFC connections (transaction SM59), but this time you must choose one of the three activation types that begin with "Start on":

- **Start on application server**
  The RFC server program is started directly on the application server that wants to execute the function call. On Windows, this happens directly via `CreateProcess()`, on Unix/Linux, it happens via a fork. As there may be a large performance impact when forking a large process, you may select, whether the child process shall be forked directly by the work process (disp+work) or by the gateway process (gwrd). See the documentation of the profile parameter `rfc/use_gwstart`.

  As the RFC program often needs access to configuration files in its current working directory (CWD) or writes trace files or output files to the CWD, it is important to know, what the CWD of a started server program is. If a work process starts the program, its CWD will be DIR_HOME, if the gateway starts it, the CWD will be the HOME directory of the <sid>adm user running the SAP system.

  DIR_HOME can be checked in SAP directories (transaction AL11) and is usually defined to something like
  /usr/sap/<sid>/D00/work
  /usr/sap/<sid>/DVEBMGS00/work
  or even
  /usr/sap/<sid>/SYS/exe/run
  /usr/sap/<sid>/DVEBMGS00/exe

  The sys admin's HOME directory is usually something like
  /home/<sid>adm

  (Here <sid> denotes the 3-letter system ID of the ABAP system.)

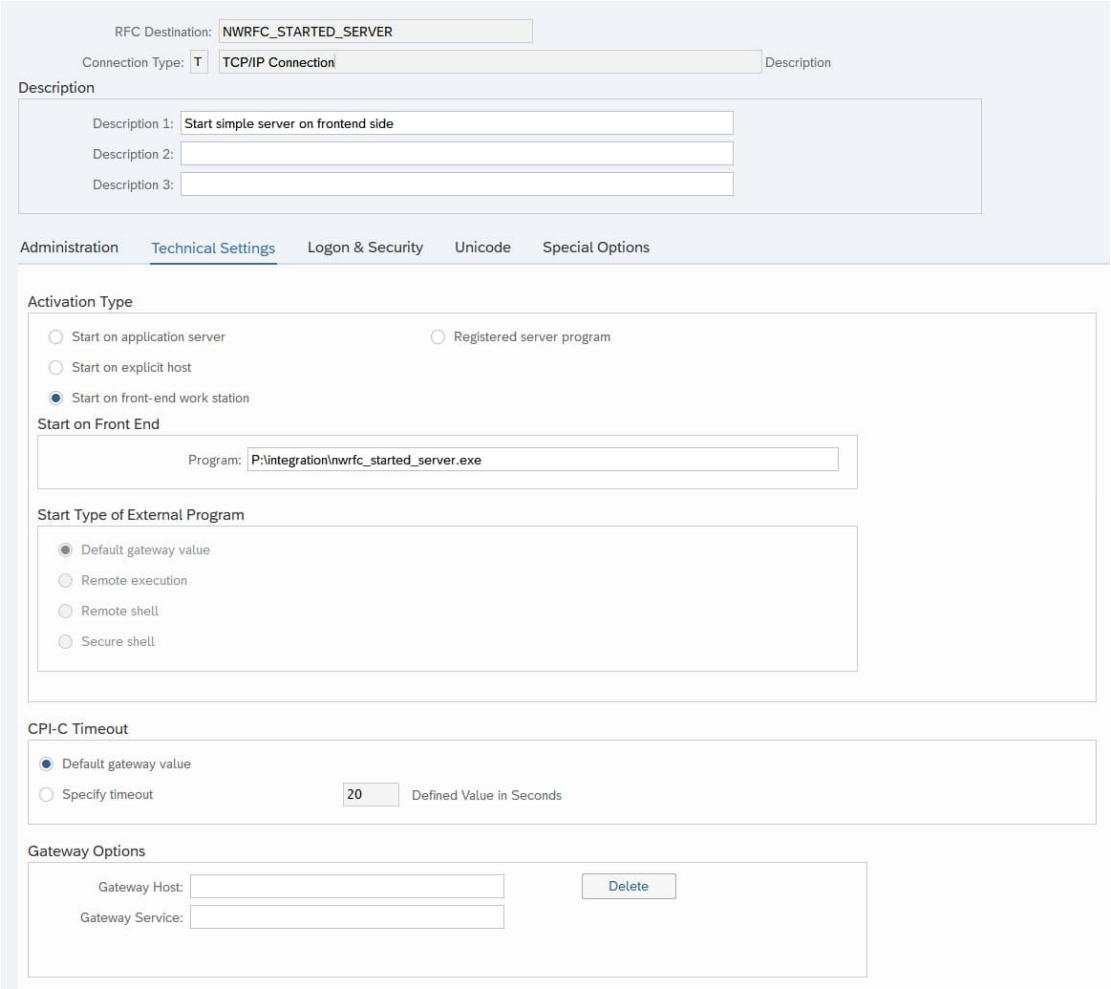
- **Start on explicit host**
  This is a leftover from the past, which nowadays probably does not have much relevance anymore. Under the following conditions it is possible to start the RFC server program on an arbitrary target machine:
  - Both the application server executing the RFC request and the target machine need to support some mechanism for remote login.
  - The <sid>adm user needs to be able to logon remotely to the target machine.
  See also the documentation of profile parameter `gw/rem_start`. Usually this feature is disabled for security reasons.

  The mechanism to be used for remote login can be selected in the section "Start Type of External Program".

- Remote execution: login to the target server is performed via `rexec`.
- Remote shell: login to the target server is performed via `remsh` or `rsh`, depending on the platform.
- Secure shell: login to the target server is performed via `ssh`.

- **Start on front-end workstation**
  In this case, the current user session must have been initiated via SAPGui. The necessary parameters for starting the RFC server program are then transferred back to SAPGui via the current DIAG connection, and the SAPGui process starts the server program as a child process. During that time, the gateway waits for the incoming connection from the server program and then sends the RFC request over that connection.
  Note that this is the easiest way to use, if you need to debug a started server program. Just set it up as “Activation Type = Start on front-end workstation”, modify it to pop up a dialog box at the point of interest, make sure that SAPGui can find the executable and all required DLLs and then start the RFC call. Once the popup appears on the screen, you can attach the debugger to the process, click OK on the popup and continue debugging. However, be aware that you must get this done within the gateway timeout, which is usually quite short, for example 60 seconds. Therefore, it might be a good idea to increase the value of “CPI-C Timeout” in the RFC destination definition.

In all three cases you have to specify the name of the executable to be started. If it can be found in the PATH environment variable of the target machine, just the name will be enough, for security reasons it is better to provide the absolute path.



5.3.2 Implementation

The implementation of a started server program is very similar to the implementation of the dispatch-loop server described in chapter 5.2.3 *Dispatch-Loop RFC Server*. The only difference is, that the connection to the gateway is not established with the function `RfcRegisterServer()`, but via `RfcStartServer()`. This function receives the command line arguments that have been set up by the gateway and passed to the process when it was started. These command line parameters contain the necessary values the RFC layer needs for opening the connection to the gateway, and an additional parameter that allows the gateway to “recognize” the incoming connection and assign it to the waiting RFC request.

If the application wants to pass additional parameters to the RFC layer, for example, a trace level or instructions for the codepage behaviour, it can do so by passing an optional `RFC_CONNECTION_PARAMETER` array.

This may look as follows:

```
#if defined(WIN32) && defined(MY_DEBUG)
   #include <windows.h>
#endif

int mainU(int argc, SAP_UC** argv){

#if defined(WIN32) && defined(MY_DEBUG)
   MessageBox(NULL, L"Attach Debugger and continue", L"Debug Popup",
                 MB_ICONINFORMATION);

#endif

   /* Set up the usual stuff as before:
      - metadata
      - function module implementations
      - access control */


   RFC_CONNECTION_PARAMETER serverParam;
   RFC_CONNECTION_HANDLE serverHandle = NULL;
   RFC_ERROR_INFO errorInfo;
   RFC_RC rc;

   /* By using sapnwrfc.ini, it is possible to add things like trace or
      Codepage without having to recompile. Of course, it needs to be made
      sure, that values like hostname or gateway service are not added in
      the sapnwrfc.ini file, as these would conflict with the parameters
      contained in argv.
    */
   serverParam.name = cU("dest");
   serverParam.value = cU("MY_SERVER");

   /* The following code sets up the server for just one single call.
      If a server should be implemented that continues to run for a while
      and can handle multiple calls, a while loop has to be put around
      this, similar to the dispatch-loop program of section 5.2.3.1.
      Note, however, that in this case it is not possible to reestablish
      the connection, once it gets broken. Another RfcStartServer() with
      the same parameters will result in an error, as the gateway will no
      longer be expecting a connection with the given conversation ID.
    */
   serverHandle = RfcStartServer(argc, argv, &serverCon, 1, &errorInfo);
   if (serverHandle == NULL) {
      printfU(cU("Error Starting RFC Server: %s - %s\n"),
            RfcGetRcAsString (errorInfo.code), errorInfo.message);
      return -1;
```

```
    }

    rc = RfcListenAndDispatch(serverHandle, -1, &errorInfo);
    printfU(cU("RfcListenAndDispatch() returned %s\n"),
              RfcGetRcAsString(rc));
    if (rc != RFC_OK)
       printfU(cU("%s -- %s\n"), errorInfo.key, errorInfo.message);

    if (rc == RFC_OK || rc == RFC_ABAP_EXCEPTION)
       RfcCloseConnection(serverHandle, NULL);

    return 0;
}
```

5.4 WebSocket RFC Server

In chapter 0 we introduced a new communication technology in RFC, the so-called WebSocket RFC. In the case of a client program it was quite easy to use the new technology even in existing programs: only a different set of logon parameters needed to be used, but the code of the client program could remain unchanged or nearly unchanged: potentially, the part that provides the logon parameters needed to be modified.

Changing an existing RFC server program from CPIC to WebSocket communication, however, may not be that simple. There are the following limitations:

1. An RFC server that has been implemented as a “started server” (chapter 5.3) can of course not be converted to WebSocket, as a started server by design is a program that is started only on demand by the backend, while a WebSocket server program runs continuously and waits for incoming requests.
2. A registered RFC server that has been implemented using the older Dispatch-Loop mechanism described in chapter 5.2.3, cannot be switched to WebSocket either without changing the implementation.

However, if the RFC server program has already been implemented as a “Managed Server” (chapter 5.2.2), then it can as easily be switched to WebSocket communication as a client program: you just need to modify the connection parameters that you pass into `RfcCreateServer()`, or that are defined in *sapnwrfc.ini*. This is described in the following.

5.4.1 Connection Parameters for a WebSocket RFC Server

So, let’s assume you have implemented a managed RFC server program as outlined in chapter 5.2.2 and want to start it as a WebSocket server instead of a registered server. The main difference is that you have to replace the triple of GWHOST, GWSERV and TPNAME/PROGRAM_ID, which specifies the gateway to which the server program is supposed to connect to, with the single parameter WSPORT, which specifies the port on which the server is to start listening. As in the client case, there are a few differences:

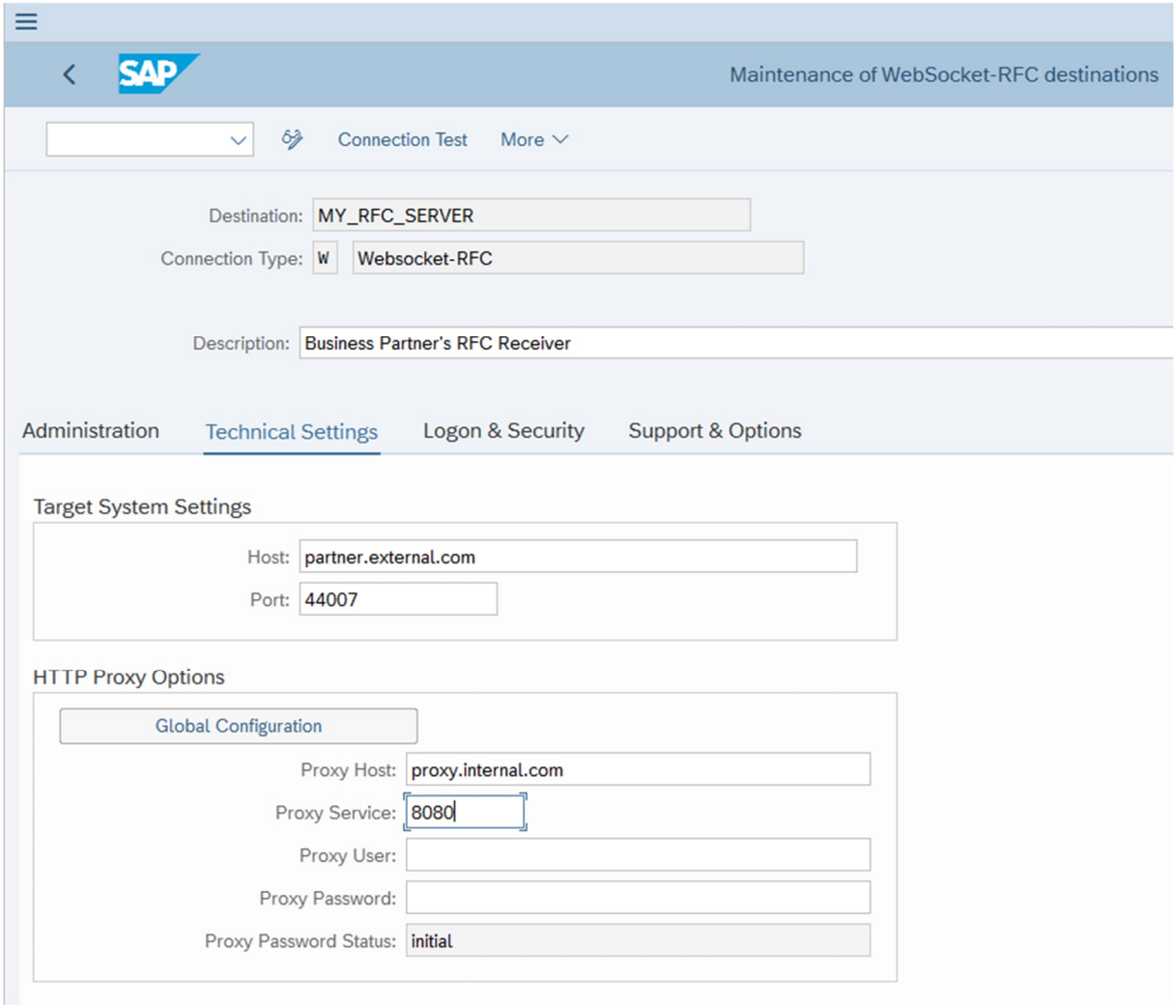
1. Using a SAProuter is not possible, and in fact does not make sense: a WebSocket RFC server does not need to open a connection to some gateway, it just opens a server port and accepts incoming client connections that other programs open to it.
2. Instead of using SNC, you again use TLS.

All connection parameters for WebSocket RFC server programs are documented in the *sapnwrfc.ini* file that comes with the NW RFC SDK. See section 4.2. in *sapnwrfc.ini*.

5.4.2 Setting up the RFC Destination

If you want to call a WebSocket RFC Server from an ABAP system, you need to define a corresponding RFC destination in in the configuration of RFC connections (transaction SM59) like for any other RFC server. For communication via the WebSocket protocol, you have to set up a destination of type 'W' and fill in the host and port of the target server as well as any proxy information, if the server needs to be accessed across an explicit proxy.

Here is an example that allows you to call function modules in a business partner's server program from an ABAP report via the internet.



In the "Logon & Security" tab you can then either enter a username and password or select a client certificate to be used for login at the server, if the partner is an ABAP system. The current implementation of the WebSocket RFC server is like for other server types not interpreting any credentials. Hence, currently only the trusted CA certificates to be used for validating the server's certificate, have to be maintained in transaction STRUST. Note that for security reasons, the kernel WebSocket RFC implementation requires TLS. Plain HTTP is not allowed. The remaining settings on the "Logon & Security" tab are relevant only for ABAP-to-ABAP connections and can be ignored in the case of an external server program.

5.5 Transactional RFC Server

Before going into the details of how to write a tRFC server program, it is helpful to read the introductory section of chapter 4.3 *Transactional RFC Client* again. It explains the general mechanism of the tRFC protocol; and a thorough understanding of how this protocol works and how it is intended to be used, is essential for implementing tRFC client and server programs alike. Note that in this chapter the term "*transaction*" will be used in two different unrelated contexts with completely different meaning:

1. An SAP GUI *transaction code* that allows starting certain screens in the SAP backend system, for example, the function builder with SE37, the configuration of destinations with SM59 and so on.
2. A *unit of work* used for ensuring *guaranteed transmission* (data is processed by the recipient exactly once and is not lost in case errors happen on the way) and *atomic behavior* (data is processed according to the "all-or-nothing" principle, avoiding data inconsistencies).

Normally, it should be clear from the context, which of the two meanings of "transaction" is meant in the given situation.

Just like in the client case, a tRFC server will also need some kind of status keeping component, which ideally should be a database. But it doesn't stop here: writing a fully functional and "transactionally secure" tRFC server program is much more effort than writing an analogous tRFC client program, because of two reasons:

1. If the tRFC LUWs received by the server program consist of several function modules, the external program must guarantee the atomic behavior, meaning it has to make sure that either **all** function modules in that LUW are executed successfully, or **none**. This cannot really be done reliably without a database.
   In the tRFC client case, this work was done by the backend system. Here, we have to implement it ourselves in the tRFC server program.
2. The tRFC server program needs to react to a set of events that are triggered by the backend system or by the NW RFC library runtime. For this, the RFC server application has to install a set of C callback functions, which are invoked by the NW RFC library, whenever such an event is to be triggered.

For keeping track of the status of incoming transactions, we will reuse our (hypothetical) API from chapter 4.3, which looked like this:

```
// An API for keeping track of transaction status.

typedef enum _TIDStatus {
   Status_Created,
   Status_Executed,
   Status_Committed,
   Status_RolledBack,
   Status_Confirmed
} TIDStatus;

typedef enum _StatusRC {
   RC_OK,
   Error,
   NotFound,
} StatusRC;

StatusRC getTIDStatus(RFC_TID tid, TIDStatus* status,
                      SAP_UC* errorMessage);
StatusRC setTIDStatus(RFC_TID tid, TIDStatus status,
                      SAP_UC* errorMessage);
```

```
StatusRC deleteTID(RFC_TID tid);
StatusRC listTIDs(RFC_TID** tids, unsigned* numEntries);
```

When a backend system initiates a tRFC transaction, the NW RFC library first checks whether a set of four transaction-handler callback functions is installed via `RfcInstallTransactionHandlers()` for that system ID. If the RFC library can't find these functions, it checks if they are installed for systemID=NULL. If it still can't find them, the RFC library refuses to accept the tRFC call.

Let's now take a closer look at these four events and their handlers.

5.5.1 Create Event

Upon receiving an incoming tRFC LUW, the NW RFC library first calls the handler of type `RFC_ON_CHECK_TRANSACTION`, passing the current TID as input. This handler is the most important piece in our tRFC implementation for guaranteeing transactional security, because if it behaves incorrectly, we run the risk of both, losing a transaction without a trace and causing duplicates of the same transaction. This handler's task is to check the status of the given TID and then report to the backend how to continue. This check can have three possible outcomes:

1) **The status management component is currently unavailable due to technical problems (for example, the database used for storing the status information is currently down).**
   In this case, the server cannot guarantee transactional security, so it should refuse to process any tRFCs. Your handler implementation must return `RFC_EXTERNAL_FAILURE`, and the NW RFC library will then abort the current call. The backend stores the failed LUW and a corresponding error message in the transactional RFC monitor (transaction SM58). From here it can later be retried, either manually by an administrator or periodically by an automatic job, until it finally succeeds.
   Note: some basic retry settings can be defined in the configuration of RFC connections (transaction SM59) in the setup of the RFC destination pointing to the server program: "Edit → tRFC". Here you can specify, whether the ABAP system's tRFC runtime should or should not perform its resend attempts, if an LUW was aborted because of a network error, how often it should try to resend the LUW automatically until an administrator needs to take care of this case, and what the time interval between two retries is. For more fine-grained control of the tRFC resend behavior, you can create a periodic job with report `RSARFCEX`.
2) **The TID does not yet exist in our status database, or it exists in status "Created" or "Rolled Back".**
   This means the server received a new transaction or the resend attempt of a previously failed/aborted transaction. In this case your handler implementation ought to
   a) create an entry for this TID in the status keeping component with status `CREATED`,
   b) open a database transaction for this TID
      (This step can be omitted, if you are 100% sure that your tRFC server will always receive LUWs consisting of only one single function module.)
   c) and return `RFC_OK`

   After that, the tRFC machinery will continue as usual and the processing of the actual data will commence.
3) **The TID already exists in our status database and is in status "Executed" or "Committed".**
   This means we have the very rare case of an LUW that was already processed successfully on our side at an earlier point of time, but where the technical "OK

acknowledgment" did not make it back to the calling ABAP system (for example, because of a network error after sending the data to the NW RFC server, but before receiving the response). Therefore, in order to play it safe, the backend system now resends the transaction.
Your handler implementation should simply return RFC_EXECUTED. The NW RFC library will then immediately return another "OK acknowledgement" to the calling ABAP system, and the processing of the data is skipped. This time the acknowledgement will hopefully make it back to the caller successfully, and the ABAP system can then clean up the data and transaction status on its side as well.

A sample implementation of the RFC_ON_CHECK_TRANSACTION handler could look like this:

```
RFC_RC SAP_API onCheckTID(RFC_CONNECTION_HANDLE rfcHandle,
                          SAP_UC const *tid) {
   StatusRC rc;
   TIDStatus status;

   printfU(cU("\n\nChecking TID %s\n"), tid);
   rc = getTIDStatus((RFC_TID)tid, &status, NULL);

   switch (rc) {
      case RC_OK:
         if (status == Status_Created || status == Status_RolledBack)
            goto start_processing;
         return RFC_EXECUTED;

      case NotFound:
         rc = setTIDStatus((RFC_TID)tid, Status_Created, NULL);
         if (rc == RC_OK)
            goto start_processing;
         // else fall through and abort
      default:
         // Log an error and inform admin
         return RFC_EXTERNAL_FAILURE;
   }

   // If we get here, we received a transaction that must be
   // processed or re-processed.
   start_processing:
   // Open a connection to the local database, "attach" it to the
   // current tid and start a database transaction for that connection.
   // If you are sure that the tRFC LUWs you will be receiving, only
   // contain one function module, you can of course omit this step.
   return RFC_OK;
}
```

5.5.2 Execute Event

If the RFC_ON_CHECK_TRANSACTION handler ended with RFC_OK, the processing of the actual data will now proceed. For this, the NW RFC library will call one “normal” server function implementation (type RFC_SERVER_FUNCTION) for every function module contained in the current tRFC LUW. These server functions are executed in the correct order, corresponding to their position in the LUW.

This means that your server program needs to implement one or more RFC_SERVER_FUNCTION callback functions for each function module the server shall be able to receive, and to install them via RfcInstallServerFunction() or RfcInstallGenericServerFunction(). This is very similar to the case of a synchronous server program as described in chapter 5.1.1 *Implementing a Function Module in C/C++*. However, there are a few subtle differences, you should keep in mind when implementing a server function that handles tRFCs:

- tRFC does not support any return parameters. Your server function implementation may set the data container’s EXPORTING parameters or change the CHANGING and TABLES parameters, but the changes will be ignored by the RFC runtime of the NW RFC library.
  In particular, this means that for indicating an error situation, your function module must throw an exception (by returning RFC_EXTERNAL_FAILURE). Using a mechanism like the BAPIs, which return error messages in a RETURN structure or table, will not work, as these parameters will be lost, and the call will be interpreted as successful.
- A function module called in tRFC modus cannot throw any ABAP Exceptions or ABAP Messages. It can only use a SYSTEM_FAILURE to report an error condition to the SAP system. If you return RFC_ABAP_EXCEPTION or RFC_ABAP_MESSAGE from a server function while executing a tRFC call, the NW RFC library will translate this into a SYSTEM_FAILURE and preserve as much error detail as it can.
- A server function processing a tRFC call must be able to roll back its changes, even if it finished “successfully”. Another function module, which is from the same LUW, but is executed after this one, may run into an error, and then the entire LUW must be rolled back. Only if you are 100% sure that your server will receive only LUWs consisting of a single function module, you may get away without this capability.

Hence, a server function implementation that wants to be ready for the tRFC protocol, should perform these additional steps:

```
RFC_RC SAP_API myFunctionModule(RFC_CONNECTION_HANDLE rfcHandle,
                                RFC_FUNCTION_HANDLE funcHandle,
                                RFC_ERROR_INFO* errorInfoP) {

   RFC_SERVER_CONTEXT context;

   RfcGetServerContext(rfcHandle, &context, errorInfoP);
   if (errorInfoP->code != RFC_OK)
      return RFC_EXTERNAL_FAILURE;

   if (context.type == RFC_TRANSACTIONAL || context.type == RFC_QUEUED) {
      // Obtain the database connection that was opened for the
      // current TID context.tid in the onCheckTID handler.
   }

   // Implement functionality of function module MY_FUNCTION_MODULE,
   // if any.

   if (context.type == RFC_TRANSACTIONAL || context.type == RFC_QUEUED) {
      // Store the data received in funcHandle as well as any results
      // computed in the business functionality of the FM (if any)
      // via the DB connection obtained above.
```

```
        // Optionally, if you want to make status tracking and
        // forensic analysis in case of an error easier, you can
        // update the TID's status to EXECUTED in this step.
        setTIDStatus(context.tid, Status_Executed,
              cU("Executed function MY_FUNCTION_MODULE"));
    }
    return RFC_OK;
}
```

5.5.3 Rollback or Commit Event

If one of the server function implementations in the "execute phase" returns an error, the NW RFC library skips the remaining function modules contained in the current tRFC LUW and the runtime immediately fires the Rollback event. However, if the loop over the entire LUW has completed and all server functions returned `RFC_OK`, the NW RFC library runtime fires the Commit event.

Your handler for the Rollback event (`RFC_ON_ROLLBACK_TRANSACTION`) then has to set the TID's status to Rolled Back, obtain the database connection corresponding to this TID and then execute a DB Rollback on that connection and close it.

Similarly, the handler for the Commit event (`RFC_ON_COMMIT_TRANSACTION`) will set the TID's status to Committed and perform a DB Commit Work on the database connection corresponding to the current TID. In terms of code, this could look as follows.

```
RFC_RC SAP_API onCommit(RFC_CONNECTION_HANDLE rfcHandle, SAP_UC const *tid) {
    StoreRC rc;

    printfU(cU("Committing TID %s\n"), tid);
    rc = setTIDStatus((RFC_TID)tid, Status_Committed, NULL);

    // Obtain the database connection that was opened for the
    // current TID tid in the onCheckTID handler.
    if (rc == RC_OK) {
        // Perform a COMMIT WORK on the DB connection and close it.
        // If the COMMIT WORK failed, set the TID status back to
        // Rolled Back, and then return RFC_EXTERNAL_FAILURE, so that
        // the backend will retry the LUW.
    }
    else {
        // If we cannot access the TID store at this time, we should
        // not commit the transaction to the database, because the
        // status in our status component may still be Created,
        // and the backend may retry the transaction at a later time,
        // leading to a duplicate.
        // Perform a ROLLBACK WORK on the DB connection and close it.
        return RFC_EXTERNAL_FAILURE;
    }
    return RFC_OK;
}
```

```
RFC_RC SAP_API onRollback(RFC_CONNECTION_HANDLE rfcHandle,
                          SAP_UC const *tid) {
   printfU(cU("Rolling back TID %s\n"), tid);
   // Obtain the database connection that was opened for the
   // current TID tid in the onCheckTID handler.
   // Perform a ROLLBACK WORK on that connection and close it.
   setTIDStatus(tid, Status_RolledBack, NULL);
   return RFC_OK;
}
```

After the corresponding handler has returned, the NW RFC library then sends either a `SYSTEM_FAILURE` with the error details provided by the server function that triggered the error, or an “OK acknowledgement” back to the calling ABAP system. In case of an error, you can find the TID together with that error message in the transactional RFC monitor (transaction SM58).

5.5.4 Confirm Event

If the calling ABAP system receives the “OK acknowledgement” response, it triggers a final event: the Confirm event. This comes in a separate RFC request, so if your server implementation is using the older Dispatch-loop mechanism (chapter 5.2.3), the server logic needs to make sure to loop at least one more time into `RfcListenAndDispatch()` in order to process that event. The Managed RFC server (chapter 5.2.3) handles the event automatically.

The Confirm event is basically a house-keeping event. The calling ABAP system tells the RFC server, that it knows that it has processed this transaction successfully, and that it won’t resend this transaction ever again. Consequently, the tRFC server no longer needs to protect itself against accidental duplicate processing of this transaction and can safely delete the TID from its status keeping component in order to free up allocated resources (disc space or database space). The sample handler below of type `RFC_ON_CONFIRM_TRANSACTION` does exactly that:

```
RFC_RC SAP_API onConfirm(RFC_CONNECTION_HANDLE rfcHandle,
                         SAP_UC const *tid) {
   printfU(cU("Confirming TID %s\n"), tid);
   deleteTID((RFC_TID)tid);

   return RFC_OK;
}
```

5.5.5 Putting Everything Together

Using these four event handlers and the specialized server function implementation(s) for the function module(s) the server shall to be able to process, we can now put everything together for a tRFC-capable server program. Note that we also need to use many of the general RFC server features described in chapters 5.1 and 5.2. Here is now an outline of a tRFC server program:

```
int mainU(int argc, SAP_UC** argv) {

   RFC_CONNECTION_PARAMETER serverParams;
   RFC_ERROR_INFO errorInfo;
   RFC_SERVER_HANDLE myServer = NULL;

   // Obtain or create metadata for MY_FUNCTION_MODULE (and other
   // function modules the program wants to process).
   RFC_FUNCTION_DESC_HANDLE myFuncDesc = createFuncDesc();

   // Install the server function from step 5.5.2
   RfcInstallServerFunction(cU("ABC"), myFuncDesc,
                            myFunctionModule, &errorInfo);
   if (errorInfo.code != RFC_OK)
      goto errorHandling;

   // Install the tRFC event handlers from steps 5.5.1, 5.5.3 and 5.5.4
   RfcInstallTransactionHandlers(NULL, onCheckTID, onCommit, onRollback,
                                 onConfirm, &errorInfo);

   if (errorInfo.code != RFC_OK)
      goto errorHandling;

   // If you want to restrict access to the function module(s),
   // install an authorization handler as described in chapter 5.1.3

   // In this case, parameters for my server are stored in sapnwrfc.ini
   // See chapter 5.2.2.1 for details
   serverParams.name = cU("DEST");
   serverParams.value = cU("MyServer");

   myServer = RfcCreateServer(&serverParams, 1, &errorInfo);
   if (errorInfo.code != RFC_OK)
      goto errorHandling;

   // If you are interested in these events, add a StateChangedListener
   // and/or an ErrorListener here, as described in chapter 5.2.2.2
   // If you want to monitor incoming calls, set up a separate thread
   // that uses the connection monitor feature as described in
   // chapter 5.2.2.3

   RfcLaunchServer(myServer, &errorInfo);
   if (errorInfo.code != RFC_OK)
      goto errorHandling;

   printfU(cU("Server is up and running. Press \"s\" to stop it.\n"));

   while (1) {
      SAP_UC c = (SAP_UC)fgetcU(stdin);

      if (c == cU('s')) {
         RfcShutdownServer(myServer, 60, NULL);
         RfcDestroyServer(myServer, NULL);

         return 0;
      }
   }

   errorHandling:
   // Log the error

   if (myServer)
      RfcDestroyServer(myServer, NULL);

   return 1;

}
```

5.6 Queued and Background RFC Server

The difference between tRFC and qRFC is only that an additional queuing mechanism is added to the picture. But the data transmission itself remains the same. In the case of communication between an ABAP System and an external program, the queuing and scheduling of LUWs is performed by the backend-side qRFC scheduler, i.e. an outbound queue is used for processing the LUWs. Therefore, the external program does not need to make any changes: the same tRFC server program as outlined in the previous chapter 5.5 can also process qRFC LUWs.

When moving from tRFC to bgRFC, however, several changes are necessary. bgRFC uses a 32-digit GUID instead of a 24-digit TID, the data format used for transmission is different, and bgRFC provides a few additional features that tRFC/qRFC do not support. Let's go through these necessary changes in detail.

5.6.1 Status Keeping Component

First, we need a different API for our status keeping component. Instead of working with `RFC_TIDs`, it needs to work with `RFC_UNIT_IDENTIFIERs`, and if it uses a database for storing status information, the key of the database table needs to be changed from a 24 character key field to either a 33 character key field, or a pair of 32 character key field plus 1 character key field. This is because only the combination of the 32-digit unit ID plus the 1-char unit type can be assumed to be unique. So, let's assume that we have an API as follows:

```
// An API for keeping track of transaction status.

typedef enum _UNITStatus {
   Status_Created,
   Status_Executed,
   Status_Committed,
   Status_RolledBack,
   Status_Confirmed
} UNITStatus;

typedef enum _StatusRC {
   RC_OK,
   Error,
   NotFound,
} StatusRC;

StatusRC getUnitStatus(RFC_UNIT_IDENTIFIER* unitId, UNITStatus* status,
                       SAP_UC* errorMessage);
StatusRC setUnitStatus(RFC_UNIT_IDENTIFIER* unitId, UNITStatus status,
                       SAP_UC* errorMessage);
StatusRC deleteUnit(RFC_UNIT_IDENTIFIER* unitId);
StatusRC listUnits(RFC_UNIT_IDENTIFIER** unitIds, unsigned* numEntries);
```

5.6.2 Function Module Implementation

Like the server function implementation for tRFC/qRFC from chapter 5.5.2, a server function meant for bgRFC processing needs to check the server context at the beginning and to perform some extra steps in case it is running in a bgRFC LUW:

```
RFC_RC SAP_API myFunctionModule(RFC_CONNECTION_HANDLE rfcHandle,
                                RFC_FUNCTION_HANDLE funcHandle,
                                RFC_ERROR_INFO* errorInfoP) {

   RFC_SERVER_CONTEXT context;

   RfcGetServerContext(rfcHandle, &context, errorInfoP);
   if (errorInfoP->code != RFC_OK)
       return RFC_EXTERNAL_FAILURE;

   if (context.type == RFC_BACKGROUND_UNIT) {
      // Obtain the database connection that was opened for the
      // current unit identifier context.unitIdentifier in the
      // onCheckUnit handler.
   }

   // Implement functionality of function module MY_FUNCTION_MODULE,
   // if any.

   if (context.type == RFC_BACKGROUND_UNIT) {
      // Store the data received in funcHandle as well as any results
      // computed in the business functionality of the FM (if any)
      // via the DB connection obtained above.


      // Optionally, if you want to make status tracking and
      // forensic analysis in case of an error easier, you can
      // update the context.unitIdentifier's status to EXECUTED
      // in this step.
      setUnitStatus(context.unitIdentifier, Status_Executed,
                    cU("Executed function MY_FUNCTION_MODULE"));

   }

   return RFC_OK;
}
```

5.6.3 Implementation of Event Handlers

For processing tRFC/qRFC, we had to implement four event handlers: `RFC_ON_CHECK_TRANSACTION`, `RFC_ON_COMMIT_TRANSACTION`, `RFC_ON_ROLLBACK_TRANSACTION` and `RFC_ON_CONFIRM_TRANSACTION`. For bgRFC there are now five event handlers. The first four (`RFC_ON_CHECK_UNIT`, `RFC_ON_COMMIT_UNIT`, `RFC_ON_ROLLBACK_UNIT` and `RFC_ON_CONFIRM_UNIT`) are completely analogous to the corresponding four handlers for tRFC/qRFC and need to perform exactly the same actions, only with an `RFC_UNIT_IDENTIFIER` instead of a TID. The small change, which is necessary to convert the four transaction handlers from chapter 5.5 to four unit handlers, should be obvious. Hence, let's assume we have implemented the following four functions:

```
RFC_RC SAP_API onCheckUnit(RFC_CONNECTION_HANDLE rfcHandle,
                           RFC_UNIT_IDENTIFIER const *identifier);
RFC_RC SAP_API onCommitUnit(RFC_CONNECTION_HANDLE rfcHandle,
                            RFC_UNIT_IDENTIFIER const *identifier);
RFC_RC SAP_API onRollbackUnit(RFC_CONNECTION_HANDLE rfcHandle,
                              RFC_UNIT_IDENTIFIER const *identifier);
RFC_RC SAP_API onConfirmUnit(RFC_CONNECTION_HANDLE rfcHandle,
                             RFC_UNIT_IDENTIFIER const *identifier);
```

New is now the fifth handler of type `RFC_ON_GET_UNIT_STATE`: when using bgRFC, the backend side status monitor may call the server program and inquire about the current status

of a certain unit. If this happens, the NW RFC library will trigger this handler, and here the implementation has to check the given unit identifier in its status keeping component and return the unit's current status. A sample implementation could look like this:

```
RFC_RC SAP_API ongetUnitState(RFC_CONNECTION_HANDLE rfcHandle,
                              RFC_UNIT_IDENTIFIER const *identifier,
                              RFC_UNIT_STATE* unitState) {

   UNITStatus ourStatus;
   StatusRC rc = getUnitStatus(identifier, &ourStatus, NULL, errorMessage);

   switch (rc) {
      case Error: // Our status database is currently down
         return RFC_EXTERNAL_FAILURE;
      case NotFound:
         *unitState = RFC_UNIT_NOT_FOUND;
         break;
      case RC_OK:
         switch (ourStatus) {
            case Status_Created:
            case Status_Executed:
               *unitState = RFC_UNIT_IN_PROCESS;
               break;
            case Status_Committed:
               *unitState = RFC_UNIT_COMMITTED;
               break;
            case Status_RolledBack:
               *unitState = RFC_UNIT_ROLLED_BACK;
               break;
            case Status_Confirmed:

               // This case can happen only, if we implemented our
               // onConfirmUnit handler to set the status to confirmed
               // instead of deleting the status entry altogether.
               *unitState = RFC_UNIT_CONFIRMED;
               break;

         }

   }

   return RFC_OK;
}
```

5.6.4 Registration of Event Handlers

Now it only remains to replace the call to `RfcInstallTransactionHandlers()` in the main function of the tRFC program above with

```
RfcInstallBgRfcHandlers(NULL, onCheckUnit, onCommitUnit, onRollbackUnit,
                        onConfirmUnit, ongetUnitState, &errorInfo);
```

and we are ready to go.

5.7 Error Handling on ABAP Side When Calling an External RFC Server

In order to ensure a robust connectivity between ABAP and an external server program, a good error handling is necessary also on ABAP side. The minimum is to catch COMMUNICATION_FAILURE (network problems can always occur) and SYSTEM_FAILURE.

Then, depending on what ABAP Exceptions and ABAP Messages the C/C++ function module implementation may trigger, the ABAP code calling this function module also needs to be prepared for that. The following ABAP sample code illustrates, how the different types of failures, exceptions and messages can be caught by the calling program. Assume that the function module FUNCTION_NAME has an ordinary ABAP Exception named INVALID_INPUT defined in its interface, and that it may also trigger ABAP Messages in case of severe errors. Unfortunately, the ABAP language does not allow distinguishing between a SYSTEM_FAILURE with SY-MSG parameters and an ABAP Message, so the following ABAP code will treat these two cases identically.

```
REPORT Z_CALL_EXTERNAL_SERVER LINE-SIZE 170.

DATA: rfc_message(128), error_message(256).

DATA: my_output ... ,
      my_table ... .  "As needed by the function module

CALL FUNCTION 'FUNCTION_NAME' DESTINATION 'MY_SERVER'
    EXPORTING
      INPUT           = 'value'
    IMPORTING
      OUTPUT          = my_output
    TABLES
      A_TABLE         = my_table
    EXCEPTIONS
      INVALID_INPUT         = 1
      SYSTEM_FAILURE        = 2  MESSAGE rfc_message
      COMMUNICATION_FAILURE = 3  MESSAGE rfc_message.

IF sy-subrc NE 0.
  CASE sy-subrc.

    WHEN 1.

      WRITE AT / 'An invalid input was supplied'.
      "Check whether the server provided more details
      "(ABAP Exception with Sy-parameters)
      IF sy-msgid IS NOT INITIAL.

        MESSAGE ID sy-msgid TYPE sy-msgty NUMBER sy-msgno
          WITH sy-msgv1 sy-msgv2 sy-msgv3 sy-msgv4 INTO
          error_message.
        WRITE: AT /(*) 'More details:', error_message.

      ENDIF.
    WHEN 2.

      CONCATENATE 'The server ran into serious trouble:'
        rfc_message INTO error_message SEPARATED BY space.
      WRITE AT /(*) error_message.
```

```
      "Check whether the server provided more details
      "(SYSTEM_FAILURE or E/A/X Message with Sy-parameters
      IF sy-msgid IS NOT INITIAL.
        MESSAGE ID sy-msgid TYPE sy-msgty NUMBER sy-msgno
          WITH sy-msgv1 sy-msgv2 sy-msgv3 sy-msgv4 INTO
          error_message.
        WRITE: AT /(*) 'More details:', error_message.
      ENDIF.
    WHEN 3.
      CONCATENATE 'A network error occured:' rfc_message
        INTO error_message SEPARATED BY space.
      WRITE AT /(*) error_message.
  ENDCASE.
ENDIF.
```

One point worth noting is, that an ABAP program does not need to manage its RFC connections and to open a fresh one, if an error broke the current one: the ABAP runtime automatically opens a fresh connection upon encountering the next `CALL FUNCTION` statement, if the current connection got broken during the previous call. However, the ABAP program needs to implement a special logic, if it calls a stateful server and relies on session state from the previous calls being available on server side: now, whenever it receives a SYSTEM_FAILURE or COMMUNICATION_FAILURE (cases 2 or 3 in the sample code above), it knows that the current user state on server side is destroyed and consequently that it has to abort the current stateful session and start a new one from scratch or display a permanent error to the end user and stop.

6 Performance Analysis of RFC Calls

Sometimes RFC calls take longer than expected and it is not quite clear, where exactly the time is spent. It could be in the application coding, in the NW RFC library coding, in the ABAP coding or database layer on backend side or in the network transfer. In order to help with trouble-shooting problems like these, the NW RFC library provides a feature similar to JCo's *JCoThroughput* object. Of course, this cannot answer all questions, but if you combine it with other data, for example, a timestamp added at the beginning and end of the invoked function module's ABAP code or the statistics data in the backend, you are able to compute times for other components, for example, the network time. This feature can be used for client applications as well as server applications, but the usage is slightly different in both cases, so we will treat them separately.

6.1 Measuring RFC Client Connections

The steps for measuring times and data sizes consumed during RFC client calls are quite straight-forward:

1. Open a client connection.
2. Create an object of type `RFC_THROUGHPUT_HANDLE`.
3. Attach it to the connection.
4. Execute one or more function calls over the connection.
   The NW RFC library aggregates the performance numbers of multiple function calls made over this connection for as long as the throughput object is attached to the connection.
5. Detach the throughput object from the connection.
6. Evaluate the results you are interested in.

In the case of a client application, the collected numbers have the following meaning:

- **Number of Calls**: total number of function modules executed over this connection
- **Total Time**: basically, the time between entering `RfcInvoke()` and leaving it
- **Serialization Time**: the time needed by the NW RFC library for converting the C data types of the function module's input parameters into a byte stream that can be sent over the network. This includes operations like codepage conversions, endianness conversions and compression.
- **Deserialization Time**: the time needed by the NW RFC library for converting the received byte stream of the function module's output parameters back into C data types. This includes operations like codepage conversions, endianness conversions and decompression.
- **Backend Time**: the time between sending off the request data and receiving the response data. (So, this is basically the sum of the following times: network time, time spent in the backend kernel/RFC layer, execution time of the ABAP code and database time.)
- **Sent Bytes**: number of bytes written to the network
- **Received Bytes**: number of bytes received from the network

All numbers are summed up over all calls executed over this connection during the time the throughput object was attached to it. All time values are measured in milliseconds.

If you can add timestamps at the end and beginning of the ABAP function module and/or use the statistics feature of the SAP Kernel, you can break down the Backend Time into finer details and pinpoint a potential bottleneck more precisely. But even if you only have the "ABAP Time" (difference of the two timestamps at the beginning and end of your ABAP coding), you can already make some pretty detailed observations:

- Time spent in NW RFC library coding:
  "Total Time – Backend Time",
  which should approximately be the same as "Serialization Time + Deserialization Time"
- Time spent in ABAP coding and in the backend's database:
  "ABAP Time"
- Time spent on the network and in the SAP Kernel:
  "Backend Time – ABAP Time"

In terms of coding, this will look as follows:

```
RFC_ERROR_INFO errorInfo;
RFC_CONNECTION_HANDLE connection;
RFC_FUNCTION_HANDLE funcHandle;
RFC_THROUGHPUT_HANDLE throughput;

SAP_ULLONG numberOfCalls, totalTime, serializationTime,
           deserializationTime, backendTime, sentBytes, receivedBytes;

/* Perform the usual steps of a client program, i.e. open a connection, obtain
metadata, create the function handle and fill its input parameters. */

throughput = RfcCreateThroughput(&errorInfo);

RfcSetThroughputOnConnection(connection, throughput, &errorInfo);

// Execute one or more function modules over the connection.
RfcInvoke(connection, funcHandle, &errorInfo);

// Get the performance values you are interested in.
RfcRemoveThroughputFromConnection(connection, &errorInfo);

RfcGetNumberOfCalls(throughput, &numberOfCalls, &errorInfo);

RfcGetTotalTime(throughput, &totalTime, &errorInfo);

RfcGetSerializationTime(throughput, &serializationTime, &errorInfo);

RfcGetDeserializationTime(throughput, &deserializationTime, &errorInfo);

RfcGetBackendTime(throughput, &backendTime, &errorInfo);

RfcGetSentBytes(throughput, &sentBytes, &errorInfo);

RfcGetReceivedBytes(throughput, &receivedBytes, &errorInfo);

// In the end, delete the throughput object.
RfcDestroyThroughput(throughput, NULL);
```

6.2 Measuring RFC Server Performance

When measuring the performance of a server program, there are a few differences to the client case, in terms of how to use the throughput feature as well as of what numbers can be measured. The steps for preparing and using a throughput object are as follows:

1. Open a server connection (in case of a dispatch-loop RFC server or a started RFC server) or create a server handle (in case of a managed RFC server).
2. Create an object of type `RFC_THROUGHPUT_HANDLE`.
3. Attach it to the server connection or to the server handle.
4. Start listening for incoming RFC calls.
   The NW RFC library aggregates the performance numbers of multiple function calls received via this connection or server handle for as long as the throughput object is attached to it.

5. Detach the throughput object from the server connection or server handle.
6. Evaluate the results you are interested in.

In the case of a server application, the collected numbers have the following meaning:

- **Number of Calls**: total number of function calls received
- **Total Time**: basically, the time between receiving the first byte of the RFC request and writing the last byte of the RFC response
- **Deserialization Time**: the time needed by the NW RFC library for converting the received byte stream of the function module's input parameters into C data types. This includes operations like codepage conversions, endianness conversions and decompression.
- **Serialization Time**: the time needed by the NW RFC library for converting the C data types of the function module's output parameters into a byte stream that can be sent over the network. This includes operations like codepage conversions, endianness conversions and compression.
- **Application Time**: the time needed by the server function implementations (type `RFC_SERVER_FUNCTION`) as well as by any tRFC/bgRFC handlers.
- **Sent Bytes**: number of bytes written to the network
- **Received Bytes**: number of bytes received from the network

All numbers are summed up over all calls received via this connection/server handle during the time the throughput object was attached to it. All time values are measured in milliseconds.

In the server case we don't have any information about how much time was spent on the network or in the backend system. You would need to use performance counters on backend side to get this information. However, if you are able to add timestamps directly before and after the CALL FUNCTION statement that calls our server, we can already make some observations about network and SAP Kernel time. Let's call this time "Call Time".

- Time spent on the network and in the SAP Kernel:
  "Call Time – Total Time"
- Time spent in NW RFC library coding:
  "Total Time – Application Time",
  which should approximately be the same as "Serialization Time + Deserialization Time"
- Time spent in your application coding:
  "Application Time"

The following code illustrates this, using the example of a managed server.

```
RFC_ERROR_INFO errorInfo;
RFC_SERVER_HANDLE server;
RFC_THROUGHPUT_HANDLE throughput;

SAP_ULLONG numberOfCalls, totalTime, serializationTime,
deserializationTime, applicationTime, sentBytes, receivedBytes;


/* Perform the usual steps for creating a managed server. */

throughput = RfcCreateThroughput(&errorInfo);

RfcSetThroughputOnServer(server, throughput, &errorInfo);

/* Start the server and wait until it received and processed the expected
 function calls. */
RfcLaunchServer(server, &errorInfo);
```

```
sleep(x);
RfcShutdownServer(server, 60, &errorInfo);

// Get the performance values you are interested in.
RfcRemoveThroughputFromServer(server, &errorInfo);

RfcGetNumberOfCalls(throughput, &numberOfCalls, &errorInfo);

RfcGetTotalTime(throughput, &totalTime, &errorInfo);

RfcGetSerializationTime(throughput, &serializationTime, &errorInfo);

RfcGetDeserializationTime(throughput, &deserializationTime, &errorInfo);

RfcGetApplicationTime(throughput, &applicationTime, &errorInfo);

RfcGetSentBytes(throughput, &sentBytes, &errorInfo);

RfcGetReceivedBytes(throughput, &receivedBytes, &errorInfo);

// In the end, delete the throughput object.
RfcDestroyThroughput(throughput, NULL);
```

6.3 Tips and Tricks

If you are interested in how much time your program needs for fetching metadata from the backend DDIC, and whether it might be worth it in your scenario to use hardcoded metadata or to save it into a file and load it from there, you can attach a throughput object to the connection, before you perform your metadata lookups via `RfcGetFunctionDesc()` or `RfcMetadataBatchQuery()`.

You can also find out, how many roundtrips can be saved by using the `USE_REPOSITORY_ROUNDTRIP_OPTIMIZATION` feature, if your backend supports it. Run your scenario twice, once with the connection parameter `USE_REPOSITORY_ROUNDTRIP_OPTIMIZATION = 1` set, and once without it, and then compare the “numberOfCalls” value obtained in both cases.

If your program needs to make many independent performance measurements, you can reset a throughput object to zero in between the single measurements using `RfcResetThroughput()` and then reuse it for the next measurement instead of creating and destroying new objects all the time.

7 RFC Callback

Before going into the technical details of our next topic, “RFC Callback”, you ought to know that this is an old and deprecated feature which should no longer be used in modern applications. There are several reasons for this, the most important one being security. In fact, in most installations it has already been disabled for security reasons via profile parameter `rfc/callback_security_method`, it is available only with CPIC-based RFC connections, not with WebSocket-based RFC connections, and it may be removed entirely in future versions of the ABAP system and the NW RFC library. There are also technical reasons why it cannot be used in some situations (see the limitations listed below), and finally in many scenarios the desired functionality, for which RFC Callback is used, can also be achieved by alternative mechanisms, for instance:

- The interface or flow of data could be designed in a different way: instead of the server side “fetching” additional data from the client at a later point, the client could already send all data required by the server upfront in the first request.
- If the server really needs data that cannot yet be determined at the time the client makes its initial request, the server could open a separate RFC client connection to the other side, that needs to be configured in advance, and fetch the additional data via “normal” RFC requests over this connection.

After the above, it should be clear to you that in most cases you will not need the feature of “RFC Callback”, especially when designing new applications and interfaces. Only if you absolutely need to use old function modules and scenarios that require RFC Callback or if you have to maintain an old legacy application, which uses it, you should now read on.

In normal RFC communication, there is always the sequence “client sends request, server sends the corresponding response, client sends the next request, server replies with the next response, and so”. However, this sequence can sometimes be interrupted, and client and server can temporarily switch roles. The exchange of messages then looks like this:

1. The client sends request for function module A.
2. The server starts processing A and at some point sends a request for function module B to the client.
3. The client is now “server”, processes B and then sends the response for function module B to the server over the already existing RFC connection established in step 1.
4. Steps 2 and 3 can optionally be repeated.
5. In the end, the server finishes processing A and returns the response for function module A to the client, which ends the original request/response cycle.

This feature is called *RFC Callback*, because the server makes a callback to the client, before completing the client’s original request. If you want to use this feature in an NW RFC program, you should be familiar with the techniques for both, RFC client programming (chapter 4) as well as RFC server programming (chapter 5), because the external program – whether it is a client program that wants to be prepared for receiving a callback, or a server program that wants to execute a callback – has to combine the mechanisms for client and server programming in the correct way.

Note: there are a few limitations and potential pitfalls that need to be considered, before using RFC Callback:

- RFC Callback cannot be used, while a tRFC, qRFC or bgRFC LUW is currently executing, because a callback at this point would interfere with the status events being triggered on the same RFC connection.
- WebSocket RFC based communication does not support RFC Callback
- The called function module should not perform any action that resets the user session state on either side. So any functionality involving `SYSTEM_RESET_RFC_SERVER`,

RFC_SET_REG_SERVER_PROPERTY or RfcSetServerStateful() should be avoided inside the callback.

- The called function module should not trigger a database COMMIT WORK, as this might lead to quite unexpected results.
- If the “outer” function module A has a TABLES parameter, and the code of A changes the table before executing the callback to function module B, then the changed table records are already sent to the client as part of the request data of function module B. This is so that both sides always see the same state of the involved tables. One consequence of this feature is, that in an external client program you need to deactivate the delta manager by adding the connection parameter DELTA=0, because the NW RFC library’s server-side delta-manager cannot handle these out-of-band updates. The table must be transferred entirely in each request and response.

7.1 RFC Callback in Client Programs

So let’s assume you have an ordinary RFC client program that is calling function module A via RfcInvoke(). In order for an RFC Callback to take place, both sides need to perform a few additional steps:

The client program needs to be prepared for receiving a callback to function module B, before it invokes function module A. This is achieved by using the same mechanisms a server program would use for setting up its function modules:

1. Implement the server function (type RFC_SERVER_FUNCTION) that is to process function module B.
2. Obtain metadata for function module B.
3. Register the pair of function pointer and metadata handle using RfcInstallServerFunction().
4. Now the client program is ready to receive a callback for function module B and can execute function module A via RfcInvoke() as any ordinary client program would.

The ABAP code of function module A needs to perform the callback to function module B. This is achieved by an ABAP statement like this:

```
CALL FUNCTION 'B' DESTINATION BACK
    EXPORTING
      INPUT            = an_input
    IMPORTING
      OUTPUT           = an_output
    TABLES
      A_TABLE          = a_table
    EXCEPTIONS
      AN_EXCEPTION           = 1
      SYSTEM_FAILURE         = 2  MESSAGE rfc_message
      COMMUNICATION_FAILURE = 3  MESSAGE rfc_message.
```

The complete flow of events now looks as follows:

1. The client program opens a connection and calls function module A via RfcInvoke().
2. The NW RFC library sends the request data for function module A to the ABAP side.
3. The backend receives this request and starts the ABAP code of function module A.
4. At some point this code runs into the statement CALL FUNCTION 'B' DESTINATION BACK. The backend now sends the request data for B back to the NW RFC library over the connection established in step 1.
5. The NW RFC library receives the request data for function module B, decodes it and starts your server function implementation. Note that all this happens inside the call stack and the same thread of the original RfcInvoke() from step 1.

6. Once your server function has returned, the RFC library takes its output and sends the response data for function module B back to the ABAP side.
7. Now, function module A continues executing in the ABAP system, and once it has finished, the backend returns the response data for function module A to the NW RFC library.
8. The RFC library decodes that data and then returns from the `RfcInvoke()` call started in step 1. The client program can now read the data and continue with further calls, as necessary.

As we can see in the above, the client program can receive only callbacks for those function modules, for which it installed a server function; any attempt of the ABAP side to call a different function module, will be aborted by the NW RFC library with a FU_NOT_FOUND error. Hence, this constitutes kind of a white list of function modules that the ABAP side is allowed to call.

7.2 RFC Callback in Server Programs

Now let's assume you have a server program that wants to perform a callback to the backend side. Again, both sides need to do their part for this process to work:

The ABAP side needs to be prepared for receiving the callback to function module B. In earlier releases not much had to be done for this: all function modules for which the current ABAP user had enough authorizations, could be executed via a callback from an RFC server. Nowadays, however, more security mechanisms have been put into place. Depending on the value of the profile parameter `rfc/callback_security_method`, you must enter the names of function modules, that the C/C++ program shall be allowed to invoke as a callback, into a whitelist. This whitelist is maintained in the definition of the corresponding RFC destination. In the screen described in chapter 5.2.1 *Setting up the RFC Destination*, go to the *Logon & Security* tab and enter the combination of function names A ("Called Function Module") and B ("Callback Function Module") in the Callback Positive List.

Additionally, the ABAP program that is calling function module A, can always prevent the external server from performing callbacks by executing the function module `RFC_CALLBACK_REJECTED` before the `CALL FUNCTION 'A' DESTINATION 'EXTERNAL_SERVER'` statement. See the documentation of the function module `RFC_CALLBACK_REJECTED` as well as the documentation of the profile parameter `rfc/callback_security_method`.

The C/C++ code of the implementation of function module A needs to perform the callback to function module B. This is achieved via the same mechanisms a client program would use for calling a function module:

1. Obtain metadata for function module B.
2. Fill the necessary importing parameters.
3. Execute function module B by calling `RfcInvoke()` on the connection handle that the NW RFC library has passed into the server function The ABAP system calls our external server program via `CALL FUNCTION 'A' DESTINATION 'EXTERNAL_SERVER'`.
4. The NW RFC library receives the request data for function module A, decodes it and then passes it into your server function implementation of function module A.
5. After `RfcInvoke()` has returned, process its outputs and then continue processing function module A.

The complete flow of events in this case looks as follows:

1. Together with a handle to the connection, over which the request was received.
2. At some point this server function creates and fills a function handle for function module B and executes it via `RfcInvoke()` on the connection handle obtained in step 2.

3. The NW RFC library encodes it and sends the request data for function module B to the backend.
4. The backend receives the request, executes the ABAP code of function module B and then returns the response data of function module B to the NW RFC library. Note that function module B runs inside the same ABAP user session as the code (for example a report) that originally executed the `CALL FUNCTION` statement in step 1.
5. The NW RFC library decodes the response, fills it into the function handle from step 3 and then returns from `RfcInvoke()`.
6. Your server function implementation now continues, and once it is finished, the NW RFC library takes its outputs from the function handle of step 2, encodes them and sends the response data for function module A back to the ABAP system.
7. The ABAP system now returns from its original `CALL FUNCTION` statement in step 1, and the code of that report continues running.

8 Adding Supportability Features

In long running applications, for example server or middleware-type applications, it may be useful to reproduce and trouble-shoot occurring problems on the fly without having to restart the application. For this, you need to design a few features into your application right from the beginning:

1. In order to be able to turn RFC tracing on or off on the fly while the program is running, you can use the API
```
RfcSetTraceLevel(RFC_CONNECTION_HANDLE connection,
                 SAP_UC* destination,
                 unsigned traceLevel,
                 RFC_ERROR_INFO* errorInfo);
```
   In the user interface of your application, there could be a section that allows to set the trace level interactively. If a user/administrator changes the value here, it should trigger an action that calls the above API and changes the trace level for all currently active RFC connections (or for the one currently of interest).

2. Next, it may sometimes be necessary to add a special technical connection parameter when opening the connections used by the application. Here are a few imaginable use cases:
   a. In order to investigate a certain problem, SAP support wants to see the table contents in the RFC trace in uncompressed form. To enable this, the application needs to open a connection with the parameter NO_COMPRESSION=1
   b. Normally, your application calls ordinary function modules that don't have screen output, but in one special case you need to call an older BAPI that still has screen output and currently fails with the dump DYNPRO_SEND_IN_BACKGROUND "Screen output without connection to user." In this case you need to attach a SAPGui to the RFC connection used for calling this BAPI, and this is done via connection parameter USE_SAPGUI=1 or 2.
   c. For one of the function calls, the backend system returns broken characters in one of the response parameters, which causes the call to abort with a codepage conversion failure. But the application still shall process this one call and just have the broken characters be replaced with #-characters. This can be achieved by setting ON_CCE=2.

   For all of these cases, you could offer a screen in your user interface, which closes the currently used connection(s), temporarily adds the required connection parameter to the set of parameters and then opens a new connection.
   Here, you could either have a screen where a user can activate/fill a set of "predefined" parameters that you think may be useful for your application, or where the user can enter a generic name/value pair, which allows to specify every possible connection parameter.
   Once the problem is solved or the special situation is over, the user/admin can then remove the extra parameter again and return the application to "normal productive mode".

3. If your application uses the *sapnwrfc.ini* file for storing most of its parameters, it may be nice to be able to modify the contents of the file and have your application instantly use the changed values. This can be achieved with the API
```
RfcReloadIniFile(RFC_ERROR_INFO* errorInfo);
```
   The user would for example, change the hostname value in the .ini file from the PROD to the DEV instance and then log into your program and click a button that closes the currently open connections (if any), calls the above API, and then opens a new connection.

Similarly, if your application has to keep its configuration not in the current working directory, but in a separate “config” directory inside its installation directory, you can achieve this by calling

```
RfcSetIniPath(const SAP_UC* pathName,
              RFC_ERROR_INFO* errorInfo);
```

at the startup of the program.