WIS
Wellesley Information Services

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

Part 3: Advanced topics

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

SAP Professional Journal

This article was originally published in March of 2008 by SAP Professional Journal and appears here with permission of the publisher, Wellesley Information Services.

This article is the third in a three-part series on remote function call (RFC) communications between an SAP system and an external program written in C (or another low-level programming language that has a C interface). SAP's new software development kit (SDK) for RFC communications, SAP NetWeaver RFC SDK, is the successor to the classic RFC SDK for SAP R/3. You can use it inC/C++-based applications to communicate with SAP back-end systems.

In the first article,[1] we discussed the basic data structures (metadata descriptions and data containers) in use throughout SAP NetWeaver RFC SDK. We also discussed RFC client programs (i.e., those cases where an external program issues an RFC into the SAP system). In the second article,[2] we talked about RFC server programs (i.e., those cases where the SAP system issues an RFC to an external program). This type of functionality enables you to access any kind of system or functionality from an ABAP application, even assembly-language routines or hardware device drivers, if you need them.

Here, we look closely at the following advanced features that are not as commonly used but are, nevertheless, important:

- How to handle transactional RFC (tRFC) and queued RFC (qRFC)
- How to construct hard-coded metadata descriptions
- How to handle RFC callbacks
  - How to deal with a callback from the SAP system (when the external program is the client)
  - How to issue a callback into the ABAP system (when the external program is the server)
- How to use single sign-on (SSO) or Secure Network Communication (SNC) as an alternative logon mechanism (with the option of additional data encryption)

Some of these topics are indispensable. For example, a good knowledge of the tRFC protocol is essential because it is the underlying technology for sending and receiving IDocs. At present, IDocs still form the foundation for any asynchronous data exchange between SAP and external systems, such as queuing systems or Electronic Data Interchange (EDI) subsystems. Furthermore, any network that isn't 100% private and secured against possible attack should use SNC to encrypt the data and verify the communications partner. In today's Internet and open networks world, this is becoming increasingly important to keep intruders from gaining READ access to sensitive data or even changing sensitive data in |the SAP system.

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

[2] "Improve communication between your C/C++ applications and SAP systems with SAP NetWeaver RFC SDK — Part 2: RFC server programs" (*SAP Professional Journal*, January/February 2008).

>> Prerequisites

The current article assumes that you have a basic understanding of how RFC communications work from an ABAP point of view (i.e., you should be familiar with the concepts of *remote-enabled function modules* and *BAPIs* (to call ABAP functionality from external C/C++ programs), and with the CALL FUNCTION … DESTINATION statement (to call external functionality from ABAP programs). If necessary, refer to the corresponding chapters of the online SAP library. You should also take a detailed look at the API documentation that comes with SAP NetWeaver RFC SDK: the sapnwrfc.h header file and a programming guide in PDF form. Knowledge of the classic RFC SDK for SAP R/3 is not required to understand this article, but we assume you have read the first two articles in the series.

If you haven't already done so, download the source code examples mentioned in the first two articles from the *SAP Professional Journal* Web site at http://www.SAPpro.com/downloads.cfm (go to the download area for the November/December 2007 issue). In preparation for this article, you should also download the file callback.zip from the March/April 2008 download area. This file contains two sample programs: an ABAP function module using RFC callback and a C client calling that function module. We recommend that you have the examples available as you read this article because only portions of the code appear in print. To compile and run the sample programs, you need to download SAP NetWeaver RFC SDK from the SAP Service Marketplace.[3] SAP Note 1025361 describes where to find the SDK for various operating system platforms. Make sure that you have at least patch level 1 of SAP NetWeaver RFC SDK, because a few of the functions mentioned in this article were not available at patch level 0. If you have trouble compiling the sample programs included with this article, please review the tips for compiling and linking in SAP Note 1056696.

Special features

These advanced topics exceed standard client-side or server-side RFC communications. They can be tricky, and it's easy to get unexpected results if you make a slight mistake. For example, if you don't use the tRFC protocol correctly, you can easily create duplicate transaction calls or lose them altogether. Some of these features appear often in practice; for example, whenever IDocs are involved, you have to use tRFC. Other features, such as RFC callback, occur rarely; in nine years of RFC experience, we have only seen two SAP BAPIs that use the RFC callback feature. But if you run into one of them and you aren't aware of its implications, your attempt to call such a BAPI won't work. The most common of the advanced RFC features are tRFC and qRFC. Let's start by looking at them.

Transactional RFC and queued RFC

tRFC is an enhancement to plain RFC that guarantees an "exactly once execution" for the calls that use it. In this case a "call" may be a single function module or a chain of several function modules combined into a single logical unit of work (LUW). In the latter case, tRFC also guarantees that either all function modules in the LUW execute in the target system, or none does. Failed LUWs (calls) are stored on the sender side and retried until transmission is successful.

qRFC is added on top of tRFC and guarantees the preservation of the order of execution for the different LUWs. In other words, if the sender side uses qRFC to send two LUWs, A and B in that order, then A and B will execute on the receiver side in that order, A then B, guaranteed. The SAP system will wait until LUW A executes successfully before it attempts to send LUW B. Technically, tRFC and qRFC are quite similar. The only difference between the two is that the qRFC case needs to maintain a queue count.

When you use the qRFC protocol for calls between an SAP system and an external RFC program, the queuing is always on the SAP system's side. Therefore, the external program won't notice much difference between tRFC and qRFC. If the external program is the server, there is no difference at all between the two, because the SAP system automatically sends the two LUWs in the correct order. If the external program is the client, it only needs to specify the name of a qRFC inbound queue in RfcCreateTransaction() to use qRFC. This article keeps to the tRFC case; we trust that you can translate the concepts for the qRFC case yourself.

Let's examine how the tRFC protocol works. You need to know this to set up transaction-secure scenarios, because in all the situations we've seen that failed to achieve end-to-end transactional security, the implementer had misunderstood some subtlety in the protocol (see the sidebar on page 4). Just having a transaction ID (TID) doesn't guarantee transactional security.

How tRFC works

The tRFC protocol implements a double handshake using the following steps. During this handshake, the transaction goes through four, sometimes five, different statuses: Created, Executed, Committed or Rolled back, and Confirmed.

1. The client opens a connection to the server and sends it a unique 24-digit TID. (The TID needs to be globally unique; that's the only requirement. Some operating systems offer functionality for generating 24-digit globally unique identifiers — GUIDs — but some don't. In that case, you can use the SAP system's TID generation

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

feature.) The server has to store the TID in its status-keeping component, which is typically a database. Then, the server sets the transaction's status to Created and returns OK to the client. If the server cannot securely continue to use the TID for technical reasons (e.g., because the connection to its database is currently down), the server returns an error code and the client should retry the transaction later. However, if that TID is already in the server's storage and has been processed successfully before, the server should simply return OK o the client and let the process continue.

2. Then, the client sends the actual transaction data to the server. In theory, the server should insert the data into its database at this time; it shouldn't trigger a database Commit. In practice, however, especially if the server doesn't use a database, it processes the data completely at this point. If this happens, the server simply performs a "no operation" (NOOP) in the next step (step 3). That's OK and should cause no harm. If the data processes successfully, the server sets the TID's status to Executed and then returns OK. If the server has trouble processing the data, it returns an error code.

   However, if in step 1 the TID already exists in Executed or Committed status in the server's status storage, then the server immediately returns OK in the current step without reprocessing the data. This feature allows the client to re-execute the same transaction if it doesn't know whether the data previously reached the server. If the server behaves correctly, it ignores the data when it recognizes the TID as already having executed successfully. So, the client may retry a transaction again and again until it receives an OK code from the server, with no concern about creating duplicate transaction calls.

3. Depending on the server's return code (error or OK), the client triggers either a Rollback or a Commit event. If the server has a database, it should perform its database Rollback or Commit event; if not, it can ignore these events. Either way, the server should set the TID's status to Rolled back or Committed.

4. During the three previous steps, if the client receives an error response — or no response at all — it should set its transaction status to Rolled back and try again later. The client should keep sending the transaction until it receives an OK from the server. This guarantees transactional security even in the worst-case scenario, which is when the connection breaks down while the server is trying to send the OK code to the client, after the server received and processed the data successfully. At this point the client only knows that the call ended in a network error; it doesn't know whether the data reached the server or not.

   In this case, the tRFC protocol requires that the client resend the transaction to ensure the data isn't lost. After some period of time, the client resends the transaction. The server notices that it has already processed this TID and ignores the data, immediately returning OK. This time, the OK message gets through to the client. Now, the client knows that the transaction ended successfully, so it stops sending the transaction and proceeds to the next step.

   To sum it up: The fact that the client keeps sending the data until it receives an OK response guarantees the "at least once" part of the equation; the fact that the server ignores the data if it already processed the TID guarantees the "at most once" part of the equation. Together, the parts ensure "exactly once" execution.

5. The client deletes the data and the TID (to ensure that it won't resend the data) and then sends the Confirm event to the server. This signal tells the server that the client knows the transaction arrived and won't try to resend it. Since the server no longer needs to protect itself from duplicate transaction calls, it can now delete this TID from its status storage.

>> Tip!

Be aware that the code examples illustrate tRFC handling, not IDoc processing. They can handle only one simple type of IDoc: TXTRAW01, which is used to transport plain text documents between SAP systems. This IDoc has only one kind of segment, E1TXTRW, with one field, TLINE, of type CHAR 72. Coding and decoding IDocs is typically a complicated task, especially when dealing with IDocs from non-Unicode multibyte SAP systems, and is beyond the scope of this article. If you need to process complex IDocs, we recommend using SAP's Java IDoc Library.[4]

After so much theory, let's look at some code. We have provided two programs, a client and a server. In both cases, we have used an IDoc to illustrate the tRFC. In the client example, it's easy to verify that an IDoc has arrived in the SAP system. For the server example, it's easy to send an IDoc from the SAP system. The necessary instructions for both of these cases are in the section "Testing the iDocClient and iDocServer programs" on page 8. Using a function module other than an IDoc for the tRFC requires a lot of setup and ABAP coding in the SAP system.

The client program iDocClient.c

Now, let's look at how to send tRFCs to an SAP system and how to reprocess failed tRFCs. You can simulate failed transactions by, for example, pulling out the network cable during an ongoing transaction. For this purpose, the client program iDocClient.c offers two options on its initial screen:

- It lets you create and send new IDocs to the SAP system (option 1: Send a new IDoc).

[4] SAP's Java IDoc Library at http://service.sap.com/connectors.

When tRFC fails

When tRFC didn't work as expected, we found it was always due to some variation of the following two mistakes:

1. **Transactions lost:** The client used the same TID for two different transactions, either inadvertently, because of a bug in the TID generation algorithm, or on purpose, because, for example, the client tried to reuse TIDs. Either way, let's assume that the previous transaction has already executed successfully, but its Confirm event has not yet occurred. The server still has the TID marked as executed successfully in its status storage, because TIDs are only cleaned up during the Confirm event. Then, the data for the new transaction arrives at the server. The server still has the TID from the previous call, so it ignores the current call's data and immediately returns OK to the client. The client gets the expected OK code and assumes that both transactions successfully completed processing.

   In another instance, imagine that the client is a multi-threaded program processing a high volume of data. Transactions are lost, apparently at random, because the problem depends on a race condition between two threads. If the transaction in the second thread executes between the first thread's Execute and Confirm events, then the second transaction is lost. However, if the first thread's Confirm has already taken place when the second thread sends its transaction, then both transactions are OK because the server no longer recognizes the TID. The first thread's Confirm has deleted the TID from the server's status-keeping component. Debugging such a sporadic problem is difficult; that's why it's never a good idea to reuse TIDs.

2. **Transactions duplicated:** The client program generates a new TID for every tRFC call, even for those that repeat previously failed transactions. In most cases, this isn't a problem. But if the server has successfully processed the transaction and the client hasn't received the OK code, the new TID results in the server's processing the same transaction twice.

   This problem is more common than you might expect; it happens quite regularly when more than two systems comprise the end-to-end scenario. For example, some kind of non-RFC-capable system (e.g., a Web server front end or an MQSeries system) sends a message to a middleware program that can use HTTP as well as RFC. The middleware program translates the message into an IDoc and then sends it to an SAP back-end system using tRFC.

   Many people setting up such a scenario first put the TID generation into the middleware program, which performs the tRFC call. But, sooner or later, the following sequence occurs:

1. The front-end system creates a message and sends it to the middleware program via HTTP.
2. The middleware program receives the HTTP call, generates a TID, and sends the IDoc successfully to the SAP system via tRFC.
3. The SAP system processes the IDoc successfully and returns OK to the middleware program.
4. The middleware program tries to send OK to the front-end system, but the HTTP connection has gone down.
5. The front-end system sees that the message has failed and resends it.
6. The middleware program receives the HTTP call, generates a (new) TID, and sends the IDoc successfully to the SAP system via tRFC.
7. … and suddenly we have ordered 20 refrigerators instead of 10.

To achieve end-to-end transactional security, the first system (in the above example, the front-end system) should always generate the TID; it is then passed along the entire chain. However, that's not all. Suppose the above scenario changes in the following way:

1. The front-end system creates a message and a TID and sends both to the middleware program via HTTP.
2. The middleware program sends the IDoc to the SAP system via tRFC using this TID.
3. The SAP system processes the IDoc successfully and returns OK to the middleware program.
4. The middleware program performs the Confirm step and returns OK to the front-end system.

Continues on next page

Continued from previous page

If the HTTP connection breaks down during the last step, you will get a duplicate transaction even if the front-end system attaches the same TID to the second attempt. The problem is that the first attempt's Confirm event has already removed this TID from the SAP system's status storage. Thus, the front-end system also needs to trigger the Confirm event.

In other words, the front-end system needs to make two HTTP calls to the middleware program: one with the TID and the message, which triggers the IDoc; and the other with the TID only, to trigger the Confirm event after the OK comes back from the middleware program. Sometimes, three calls are necessary if the front-end system can't generate TIDs and needs to get one from the RFC library's RfcGetTransactionID().

If the front-end system can't fulfill this requirement, you might omit the Confirm step altogether. The negative effect of this choice is that the ARFCRSTATE table, which holds the tRFC status information in the SAP system, keeps growing. However, scheduling a job in the SAP system that removes all TIDs older than four weeks can remedy this concern. It's highly unlikely that a transaction has completed successfully without informing the original sender for four weeks. To set up such a cleanup job, you can use the RSARFCDL report.

- It can display a status overview of currently pending IDocs (i.e., IDocs that you or the program created previously, but that haven't yet been sent successfully to the SAP system). You can then pick one of the failed IDocs from the status overview and try to resend it (option 2: Display currently pending IDocs).

  To keep track of the TIDs and their statuses, iDocClient.c uses some helper functions from statusTracking.c; the server program reuses the same functions later. This was the main reason for putting them into a separate module.

To understand tRFC handling, you don't need to understand all the details of statusTracking.c. Just think of it as a "poor person's database." The header file statusTracking.h explains what the helper functions do. The comments in statusTracking.h should be sufficient to explain how the status tracking component works.

1. **Create step:** When sending a new IDoc, iDocClient collects the data for it and persists it in the file system. This is necessary so that the data will still exist later if a transaction fails and needs to be repeated. Then, iDocClient creates a TID using RfcGetTransactionID(). This function uses the SAP system's TID generation functionalities, so you can only use it when you have an open connection to the back-end system. If your program needs to create and maintain transactions while the back end is down, you have to use your own TID creation algorithm or solve the problem in a way similar to how iDocClient does it (i.e., by using a continually increasing IDoc number until a TID can be created).

2. **Execute step:** After getting a TID, iDocClient executes the transaction in the back-end system. The client case is simpler than the general mechanism of the tRFC protocol because the RFC library function RfcSubmitTransaction() does a lot of the work for you and processes the Create, Execute, and Commit/Rollback steps internally. iDocClient needs to create a data container (RFC_TRANSACTION_HANDLE, a new, special "data container" that we haven't met before) for the transaction via RfcCreateTransaction(), create and fill a data container for the function module data, insert one or more function module containers into the transaction data container via RfcInvokeInTransaction(), and send the transaction to the back-end system via RfcSubmitTransaction(). This process is bundled into a single function, fireIDoc(), which is shown in detail in **Figure 1**.

3. **Rollback or Commit step:** If the function RfcSubmitTransaction() completes with RFC_OK, you know that the SAP system has successfully processed and committed the transaction's data. This means that the iDocClient program can delete the persisted data on its side and then call the function RfcConfirmTransaction() to clean up the TID's status information in the back-end system's ARFCRSTATE table. If RfcSubmitTransaction() ends with an error code, you can't determine what happened, so iDocClient has to keep the persisted data for later retries. In addition, the iDocClient program keeps status information about these presumably failed transactions permanently in a file.

   To display a status overview of currently pending IDocs, choose option 2: Display currently pending IDocs from iDocClient's initial screen for a list of these transactions. If you try to resubmit a transaction from this list, the entire tRFC protocol process will repeat.

   To test this feature, start iDocClient.c in your debugger, and set a breakpoint just before RfcSubmitTransaction() (line 225). Send a new IDoc to the SAP system defined in iDocClient's sapnwrfc.ini file; this file is used to store back-end logon information (most prominently the application server's host name). When the debugger stops, log on to the application server you used in iDocClient's sapnwrfc.ini file, use transaction code SMGW (Gateway

```
01 RFC_TRANSACTION_HANDLE tHandle = RfcCreateTransaction(*connection, tid, NULL, errorInfo);
02
03 if (tHandle == NULL){
04     printfU(cU("Error creating a transaction handle: %s\n"),
               errorInfo->message);
05     goto rollback;
06 }
07
08 rc = RfcInvokeInTransaction(tHandle, iDoc, errorInfo);
09 if (rc != RFC_OK){
10     printfU(cU("Error adding IDoc to transaction: %s\n"), errorInfo->message);
11     goto rollback;
12 }
13
14 rc = RfcSubmitTransaction(tHandle, errorInfo);
15 if (rc != RFC_OK){
16     printfU(cU("Error sending IDoc: %s\n"), errorInfo->message);
17     goto rollback;
18 }
19
20 // At this point we can guarantee that the IDoc reached the back end safely.
21 printfU(cU("IDoc %s successfully sent using TID %s\n"), dnum, tid);
22 deleteMessageBody(iDoc);
23 store_rc = deleteEntry(tidStore, index);
24 if (store_rc != RC_OK){
25     RfcDestroyTransaction(tHandle, NULL);
26     return;
27 }
28
29 // If we get here, the TID entry has been deleted, so there's no danger of ever
30 // sending the same IDoc again. Now we can clean up the TID in the back end:
31 RfcConfirmTransaction(tHandle, NULL);
32
33 RfcDestroyTransaction(tHandle, NULL);
34   return;
35
36 rollback:
37 RfcDestroyTransaction(tHandle, NULL);
38 if (RfcSAPUCToUTF8(errorInfo->message, 100, bufferC, &length,
                       &dontCare, NULL) != RFC_OK)
39     strcpy_s(bufferC, 81, "Error firing the IDoc");
40 setTIDStatus(tidStore, index, Status_RolledBack, bufferC);
41
42 /* IDOC_INBOUND_ASYNCHRONOUS doesn't throw any ABAP Exceptions.
43 So normally, if an IDoc fails, we get a "hard error". This means
44 the connection is closed afterwards. Try to reconnect in these cases. */
45 if (rc == RFC_COMMUNICATION_FAILURE || rc == RFC_ABAP_MESSAGE ||
          rc == RFC_ABAP_RUNTIME_FAILURE)
46     *connection = RfcOpenConnection(loginParams, 1, errorInfo);
```

Figure 1 Code snippet showing fireIDoc( ) function

>> Note

Although the RfcInvokeInTransaction() function has "invoke" in its name, it doesn't actually perform a call to the back-end system. The function merely serializes the function module data into the special tRFC format (not the typical RFC format), and adds it to the transaction data container.

Monitor) to display all currently active connections, and delete the connection corresponding to the iDocClient program. (For more details, please see the first article in this series.)

When you continue in the debugger, the IDoc will fail with a network error. iDocClient automatically tries to reconnect, and since you only simulated the network problem, the reconnect should be successful. Then, go back to iDocClient's initial screen and display the list of pending IDocs by choosing the second option. The IDoc you just tried to send should be listed there with an error message, such as "Connection to partner broken," and you can resend it immediately. This method allows you to simulate and test error situations that typically don't occur when developing and testing tRFC programs.

In addition, the index in lines 23 and 40 already points to the stored status information entry corresponding to the current TID.

The code takes for granted that previous processing has created a TID and stored it safely in a status-keeping component. It also assumes that the IDoc data has been prepared and persisted. The iDocClient program uses helper functions in statusTracking.c to store TIDs and IDoc data. You can also use the statusTracking component to set the query status and error information about TIDs and to read and write IDoc data blocks. In addition, the index in lines 23 and 40 already points to the stored status entry corresponding to the current TID.

The code first creates a transaction object from the existing TID (line 1), adds the IDoc data to it (line 8), and then tries to execute the tRFC in the back end (line 14). If an error occurs during any of these steps, the program skips the rest and sets the TID's status to Rolled back. Later (even if the program is stopped and restarted in-between), the program can reread the TID and IDoc data from the TID store and resend it.

However, if the Submit step is successful, the code deletes the IDoc body and the TID from the TID store. If the TID delete is successful, the code confirms the TID in the back end. The successful deletion of the IDoc body isn't important, because the code only sends the IDocs for which it finds an entry in the TID store. It never sends "loose" IDoc bodies, so there's no risk of getting duplicates.

4. **Confirm step:** The Confirm step (line 31), as well as RfcDestroyTransaction() (line 33), which frees the memory that tRFC uses, are only clean-up steps; therefore, we don't care about any errors there. The IDoc has processed successfully at this point, so any error is irrelevant.

The server program iDocServer

The server, iDocServer.c, is more complicated than the client and needs to implement all four steps or events of the tRFC protocol. To implement them in the design of the SAP NetWeaver RFC library, you must install the four callback functions that the RFC library calls when the corresponding event arrives.

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

The following describes how the four-step tRFC process works and what the demo implementation does in each step:

1. **Create step:** If the search for transaction-handler callback functions is successful, the RFC library calls the function of type RFC_ON_CHECK_TRANSACTION, providing the TID from the back end as input. (Function types are like blueprints and define how the function's inputs and outputs must look.) The user of the RFC library has to write such a function and hand it over to the RFC library, which will call it whenever the corresponding event arrives from the back end. This function should check the program's status management to see whether the tRFC server already "knows" this TID. The demo implementation uses a file-based TID store (statusTracking.c), but typically the TID store is a database. This step has three possible outcomes:

– The status management component is currently unavailable due to technical problems (e.g., the database used to store the status information is currently down). In this case, the server can't guarantee transactional security, so it should refuse to process any tRFCs. The check function returns RFC_EXTERNAL_FAILURE, after which the RFC library aborts the current call and the back-end system tries again later.

– The TID already exists and is in Executed or Committed status. In this case, the check function returns RFC_EXECUTED. The RFC library then immediately sends an OK code to the back end without performing the Execute step.

– The TID doesn't exist or exists in Created or Rolled back status. If so, you need to process the data, so the check function returns RFC_OK and the tRFC machinery continues normally.

2. Execute step: This step processes the tRFC data. It uses none of the transaction-handler callback functions; it uses an ordinary server function — several of them if the LUW consists of several function modules. This server function must be installed using RfcInstallServerFunction() or RfcInstallGenericServerFunction(). The RFC library uses the same procedure to find a suitable handler for the current function module name (see the synchronous RFC case described in the section "Designing a simple server" in the second article in this series). If the tRFC LUW consists of several function modules, the Execute step is repeated for each function module and the RFC library calls the corresponding server functions in the correct order. There are a few subtle differences between normal synchronous RFC and tRFC, which you need to keep in mind when implementing a server function that handles tRFCs:

– A tRFC has no return values. Your server function implementation may set the data container's EXPORTING parameters or change the CHANGING and TABLES parameters, but the RFC library will ignore it.

– A tRFC call can't throw any ABAP exceptions or ABAP messages. You can only use system failures 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 RFC library will translate this as a system failure and preserve as much error detail as it can.

– If you process a tRFC LUW consisting of several function modules, then your implementations need to be able to roll back their work. One of the later function modules in the chain may run into an error and abort the tRFC call, depending on what is implemented in your server functions. If an error occurs, the work of the preceding function modules needs to be undone. When the back end tries to process that tRFC again later, the whole LUW repeats. The concept of an LUW is all or nothing. The Rollback step rolls back the entire work.

   iDocServer has installed one server program for the function module IDOC_INBOUND_ASYNCHRONOUS, which prints the received IDoc data to the console and then asks the user whether to send an OK or an error message to the SAP system (so you can test resubmitting a "failed" transaction).

3. **Rollback or Commit step:** After the server functions in the previous step return RFC_OK, the RFC library calls the function of type RFC_ON_COMMIT_TRANSACTION. If the server uses a database or if it processes "multiple function module" LUWs, you should commit or persist the work of the current transaction at this point. iDocServer sets the transaction status to Committed.

   However, if a server function in step 2 returns an error code, the LUW's chain of function modules aborts immediately and the RFC library calls the function of type RFC_ON_ROLLBACK_TRANSACTION. This function rolls back all changes executed so far for the current transaction. The demo implementation sets the transaction status to Rolled back. This is OK for LUWs consisting of only one function module because this single server function can roll back its work before returning the error code.

   Then, the RFC library sends either an acknowledgement to the SAP system or a system failure with a detailed error message. After RfcListenAndDispatch() returns, you need to loop over and execute RfcListenAndDispatch() again to process the final Confirm step.

4. **Confirm step:** Once the SAP system gets the OK code, it triggers the Confirm event. When the RFC library receives this event, it calls the function of type RFC_ON_CONFIRM_TRANSACTION to delete the TID entry from the TID store.

Understanding the meaning of the four callback functions above is important.

Testing the iDocClient and iDocServer programs

After the iDocClient program sends an IDoc successfully, simply log on to the receiving system and execute transaction code WE02 (Display IDoc). Enter the message type TXTRAW, and press F8. You should see your IDoc and the data in its segments. The IDoc's Application Link Enabling (ALE) status[5] is probably an error (status 56). This means that although the IDoc was received successfully and the underlying tRFC was OK, the SAP system had no idea what to do with that IDoc because no ALE partner profile was defined for the sender and receiver values that the iDocClient program uses. For our purposes, however, this is OK. The fact that the IDoc can be seen in WE02 shows that the tRFC was OK. If we had an error on the tRFC level, we would have seen an error message in the iDocClient program, and WE02 would be empty.

Testing the iDocServer program is a more work-intensive process than trying out the client. The server trial requires that you:

1. Create a tRFC port via transaction code WE21 (Port Definition). An ALE port links the logical layer of the ALE framework to the technical communications channel for sending the IDocs. In this case, it tells ALE to send the TXTRAW01 IDocs via tRFC. As you define that port, enter the RFC destination from iDocServer's sapnwrfc.ini file. This ensures that the iDocServer program will receive the IDocs.

[5] ALE is a standard SAP technology for loosely coupling multiple systems (including SAP and non-SAP systems).

2. Create a logical system via transaction code SALE (Display ALE Customizing). Although the exact location of the entry will vary from release to release, somewhere on the initial SALE screen you'll find an entry named "Define logical system" (SAP R/3 4.5B or higher) or "Maintain logical systems" (SAP R/3 4.0B). You can give the logical system any name, for example, SPJ.

3. Create a partner profile, which will connect the port, the logical system, and the message type TXTRAW together.

– To create the profile, use transaction code WE20 (Partner Profiles) and click on Create. Enter the logical system from step 2 as the Partner No., the value "LS" as the Part. Type, and then save. Next, click on the yellow plus sign below Outbound Parameters, which allows you to add your TXTRAW01 IDoc to the partner profile.

– On the next screen, enter the message type TXTRAW, the receiver port from step 1, and the basic type TXTRAW01. Also, switch the output mode to "Transfer IDoc Immed." (or "Send Immediately," in some releases). You can leave the other fields on this screen at their defaults.

– After pressing Enter once, another field mysteriously appears on the screen: Pack. Size. Enter "1" and save the partner profile.

Now you're ready to send the first IDoc from the SAP system to the iDocServer program. Using transaction code WE19 (Test Tool), enter the Basis type TXTRAW01, and click on the Execute button. This generates an IDoc with one empty E1TXTRW segment. Fill the single field in that segment with some data and, if you wish, add a few more segments to that IDoc. Be sure to add them on the same level as the first segment, however, not as child segments. When you have finished entering the data, you need to fill the control record. Click on the first line of the IDoc, and enter the values shown in **Figure 2** and the message type TXTRAW.

Then, click on OK and click on the button "Standard outbound processing" ("Test outbound processing" in some releases) to send the IDoc to iDocServer.

If this program doesn't receive the IDoc, then either the ALE settings from steps 1 through 3, above, or the RFC destination registration is still wrong. First, display the IDoc using transaction code WE02. If its status is "03," the ALE settings are OK and the IDoc has been passed to the tRFC layer. Otherwise, you will find a detailed error message in the IDoc's status segments that tells you what's wrong with your ALE settings.

Profile setting	Receiver	Sender
Port	Your port	SAP*XYZ*, where *XYZ* is the system ID of your SAP system
Partner No.	Your logical system	T90CLNT090
Part. Type	LS	LS

Figure 2 Values for the IDoc

If the IDoc status is "03," check transaction code SM58 (Asynchronous RFC Error Log). You should find an entry for the function module IDOC_INBOUND_ASYNCHRONOUS, your user ID, and your RFC destination. The error message in that line should tell you why the IDoc wasn't sent to the iDocServer program. Fix the problem (the error messages, for the most part, are self-explanatory), and resend the tRFC from SM58 by highlighting that line and pressing F6. If you abort the IDoc within the iDocServer program, you will also find that IDoc in SM58 and you can resend it from there. This enables you to test the error case, even if everything works OK otherwise.

To understand the differences between how the classic RFC SDK and SAP NetWeaver RFC SDK handle tRFC and qRFC, see the section "tRFC and qRFC" in the sidebar that begins on page 16.

Hard-coding metadata descriptions

Although hard-coding metadata information was common practice using the classic RFC SDK, it shouldn't be necessary with SAP NetWeaver RFC SDK. If you can avoid it, do! It's a lot of tedious work, and it's quite error-prone. However, in at least two scenarios, it's necessary to hard code the metadata (see the sidebar on the next page for some tips):

- If you are writing an RFC server that absolutely must not store any logon credentials for the back-end system because, for example, the RFC server is running on a publicly accessible host outside your firewall where it would be too dangerous to store logon credentials.
- If you are writing an RFC server that needs to offer function modules that don't exist in the back-end system.

Let's look at the file hardCodedServer.c as an example. The constructFunctionDescription() function hard codes the metadata for a non-existent function module named I_DONT_EXIST. **Figure 3** on page 11 shows the structure of the metadata and the metadata's values.

The rest of the program installs a server function for that function module and waits for an incoming call. When it receives the call, it prints the input values to the console and returns a few hard-coded output values. On every third call, it throws the exception NO_MORE_FOOD to illustrate that non-existing function modules can also include ABAP exceptions.

To call the I_DONT_EXIST function module, you can use the ABAP report Z_CALL_DOC_DOLITTLE contained in Z_CALL_DOC_DOLITTLE.abap (see the sidebar on page 11).

Tips on writing metadata descriptions

Here are some general tips to make it easier to write correct metadata descriptions.

- If the function modules that the server implements exist in the back-end system, you can write a program similar to the printDescription example available at http://www.SAPpro.com/downloads.cfm (go to the download area for the November/December 2007 issue) and use the printout of that program as a guide when hard-coding the metadata for your server program. This print program connects to the back-end data dictionary (DDIC) and prints the correct field lengths and field offsets to the console. Then, you just copy these values into your server program. It's fast, and it guarantees correct values for all field lengths and offsets.
- When setting up structures, keep in mind that sometimes the memory address for a field isn't just the next free address after the end of the previous field. A field's offset must always be divisible by its length. For example:
  - An INT field must start at an offset divisible by 4.
  - A FLOAT field has to begin at an offset divisible by 8.
  - In Unicode metadata, CHAR-like fields must start at an even offset (2, 4, 6, …).
  - STRING and XSTRING fields always have a length value of 8; technically, they are realized via a pointer-like mechanism that is 8 bytes long.
  - When a field is a structure or table, the calculations become more complicated:
    - If the substructure is a Type 1 structure (i.e., a structure with a fixed pre-calculable length: a structure containing no STRING, XSTRING, or table fields within its "tree"), then that structure's data is "inlined" in the parent structure (i.e., it's actually located within the parent structure). First, you need to calculate the structure's complete length and use that value in the parent structure's metadata description.
    - If the substructure is a Type 2 structure (i.e., a structure that has at least one field of indeterminable length), that field is realized with a pointer-like mechanism so, again, you use "8" as its length.
- After you finish writing the metadata, write a simple server program that uses it and test it thoroughly by sending the full range of data from an ABAP report.

RFC callbacks

Another advanced feature of RFC communications is the RFC callback, which works in the following way. A client opens an RFC connection to a server and starts executing a function module. While the client is waiting for that function module to complete processing, the server can use the open RFC connection to execute another remote-enabled function module on the client. But the server can only do this during the short time period after it has received the original request from the client and before it has sent the response to that request. Basically, during an RFC callback, client and server switch roles for a moment.

You can perform an RFC callback regardless of whether the server is an SAP system or an external program. However, there are a few limitations:

The client needs to be prepared to accept the callback. An ABAP client can refuse to accept callbacks by using the function module RFC_CALLBACK_REJECTED before it issues the CALL FUNCTION statement. A C client based on SAP NetWeaver RFC SDK can only receive those callbacks that it actively permits. To do that, the C client must use RfcInstallServerFunction() and install an implementing function for the function module it wants to accept.

During a tRFC LUW no callback is possible, because the tRFC double handshake exclusively blocks the RFC connection.

You should avoid using anything that changes the session state on the client side (e.g., a SYSTEM_RESET_RFC_SERVER or something that triggers a Commit Work) in a callback, as it can lead to quite unexpected results.

In an external RFC program based on SAP NetWeaver RFC SDK, you can implement both scenarios: a client program accepting a callback from an SAP system, and a server program issuing a callback to an SAP system. You just need to combine the client features and the server features in the correct way.

Structure ANIMALS	
LION	CHAR 5
ELEPHANT	FLOAT
ZEBRA	INT
IMPORTING	
DOG	INT
CAT	CHAR 5
ZOO	STRUCTURE ANIMALS
BIRD	FLOAT
EXPORTING	
COW	CHAR 3
STABLE	STRUCTURE ANIMALS
HORSE	INT
EXCEPTIONS	
NO_MORE_FOOD	

Figure 3 Structure and metadata for the non-existent function module I_DONT_EXIST

Clients

Let's look at a client example, which requires more effort than the server does. A small number of BAPIs use the RFC callback feature (e.g., BAPI_DOCUMENT_CHECKOUTVIEW). A client using these BAPIs needs to know how the RFC callback works. For BAPI_DOCUMENT_CHECKOUTVIEW, the complete process is as follows:

- First, the client program needs to implement the function module RFC_START_PROGRAM and install it via RfcInstallServerFunction(). That function module will be used later during the callback. It should locate the program sapftp and start it as a child process when requested to do so. (The sapftp program is a standard SAP program that acts as a temporary FTP server and needs to be installed with the client program.)
- Now, the client program opens a connection to SAP and executes BAPI_DOCUMENT_CHECKOUTVIEW.
- Then the BAPI issues a client callback and starts the function module RFC_START_PROGRAM, requesting that sapftp be started. The ABAP statement CALL FUNCTION 'RFC_START_PROGRAM' DESTINATION 'BACK' performs this RFC callback.
- The implementation of RFC_START_PROGRAM now starts executing within the client program. Here it locates sapftp and starts it as a child process.
- When RFC_START_PROGRAM returns successfully, the execution of BAPI_DOCUMENT_CHECKOUTVIEW continues, and then the BAPI opens an FTP connection to the sapftp program on the client host. Then, the BAPI uses FTP to transmit the requested document file to that host. Afterwards, sapftp ends.

Calling Dr. Doolittle

In the CALL FUNCTION statement in Z_CALL_DOC_DOLITTLE, we used intermediate variables to assign values to the IMPORTING parameters and obtain the values of the EXPORTING parameters. This is necessary because ABAP doesn't support literals for data types other than CHAR and INT. We also recommend that you use this mechanism to avoid unpleasant surprises.

For example, see what happens when you replace the line

```
BIRD = float_value.
```

with

```
BIRD = '1.234567'.
```

These appear to be equivalent but, surprisingly, they aren't. In the server program, you get a completely different floating-point number. Why? For example, imagine an ABAP interpreter received the line:

```
float_value = '1.234567'.
```

The ABAP interpreter would "know" that float_value is a variable of type FLOAT and, therefore, would convert the literal '1.234567' to an IEEE floating-point number (8 bytes), which is then passed to the RFC layer in the line

```
BIRD = float_value.
```

However, when writing something like

```
BIRD = '1.234567'.
```

the ABAP interpreter interprets the literal "1.234567" as a CHAR 8 value and passes the 8 bytes of the ASCII representation of this string to the RFC layer. This layer then sends those 8 bytes to the hardCodedServer program. There, they are interpreted as the 8 bytes of an IEEE floating-point number, yielding quite surprising results. In a Unicode SAP system "1.234567" will be converted to a 16-byte UTF-16 representation of that string. The RFC server program will then truncate the second 8 bytes and interpret the first 8 bytes as type FLOAT, leading to similar nonsense.

- If the FTP transmission is successful, BAPI_DOCUMENT_CHECKOUTVIEW returns with an OK message. The client program can now read the file that the BAPI transferred to the client program's host from the file system and continue processing it.

For security reasons, the implementation of RFC_START_FUNCTION should make sure that only the SAP user who handles document checkout is allowed to call this server function and that this function starts only the sapftp program — no other programs. However, if the external program acts only as a client (i.e., it doesn't register at any RFC destination), then you can only execute RFC_START_FUNCTION during a callback, so this process is really quite safe.

Function modules and BAPIs that use the callback feature exist only in certain SAP applications, such as SAP Document Management System (DMS). Here is a general outline of how to set up a client program that needs to accept a callback from the server. As we mentioned earlier, you can download the file callback.zip that contains two sample programs, an ABAP function module using RFC callback and a C client calling that function module, from the March/April 2008 area of http://www.SAPpro.com/downloads.cfm.

- Open a connection to the back end and fetch the metadata for the function modules the client wants to call and for those it expects as callbacks.
- Install the implementing server functions for these callback function modules, together with their metadata descriptions, as you would in a normal RFC server program.
- Invoke the original function modules the same way you would in a normal RFC client program.

Servers

Let's look at a callback example in which the external program is the server. When the RFC library invokes a server function, it passes an RFC_CONNECTION_HANDLE to the server function (in addition to the data container containing the request data and the errorInfo structure). This is a handle to the RFC connection over which the request came from the SAP system. So far, we have only used it to obtain the RFC_ATTRIBUTES of that connection and read some information about the caller on the other side. But you can also use the RFC_CONNECTION_HANDLE to call `RfcInvoke()` to achieve a callback to the SAP system.

To make a callback, the server function implementation only needs to create and fill a data container for the function module that it wants to call in the callback. Then, it sends the request over the server connection handle using RfcInvoke().

The sample program callbackDemo.c illustrates this: It installs a handler for STFC_CONNECTION, and when receiving a request for it, callbackDemo makes a callback to BAPI_USER_GET_DETAIL to read the calling user's real name and return it in the STFC_CONNECTION response.

To try it, log on to SE37 (ABAP Function Modules) and call STFC_CONNECTION for the RFC destination pointing to callbackDemo. This is fun only if you have maintained some meaningful data for the user in SU01 (User Maintenance). The SAP user who triggers the STFC_CONNECTION call needs to be authorized to execute BAPI_USER_GET_DETAIL, because this BAPI will execute within the user session for that user (i.e., the same user session that triggers the STFC_CONNECTION call to the external server).

To learn the differences between how the classic RFC SDK and SAP NetWeaver RFC SDK handle RFC callback, see the section "RFC callback" in the sidebar that begins on page 16.

>> Note

The callback mechanism is similar to the one shown in the generic server in our second article to fetch the function module description from the back end: an RFC request arrives at the server program, the server makes another request back into the client system to obtain more information, and then the server continues to process the original request. In the **repositoryLookup()** function, however, we opened another RFC connection to make the metadata query instead of using the original connection.

So, why can't the RFC library, when receiving an RFC request, automatically make a callback over the original connection and look up the necessary metadata via this callback? Because you can only send a callback request after the original request's data is cleared off the line (i.e., after the data is read and the underlying CPIC connection[6] is switched from **READ** to **WRITE** state).

Before the RFC library can read the data from the open connection, it needs the metadata description for that data, but before it can use the open connection for the metadata lookup, it first needs to read the pending data from it. There-fore, you need a second, fresh connection for the metadata lookup.

[6] Common Programming Interface for Communications (CPIC) is an old IBM standard for communications between IBM mainframe systems in an SNA-LU 6.2 network. In the times of SAP R/2, CPIC controlled communications between SAP R/2 systems. Later, with the rise of Unix systems and TCP/IP networks, SAP wrote its own CPIC adaptation for TCP/IP, so that it could also communicate in a heterogeneous land-scape consisting of mainframe-based SAP R/2 systems and Unix-based R/3 systems. Later, SAP added the RFC protocol on top of CPIC to make communications in a homogeneous Unix landscape easier. Consequently, today CPIC is still the underlying protocol of RFC communications.

Alternative logons

Let's take a look at alternative logons.

Single sign-on (SSO)

One topic that plays a role across all the communications protocols used to connect to SAP systems (SAP GUI, HTTP, and RFC communications) is that you log on via SAP SSO tickets. While widely used in SAP to SAP communications, SSO tickets are subject to a restriction that limits their usefulness for external programs: Only an SAP system can create an SSO ticket because you need access to user management to authenticate the user before creating the ticket.

Therefore, an external program needs to log on via a user ID and password (or via SNC) before it can request an SSO ticket for that user. This makes it useless as a logon mechanism for external client programs. However, SSO tickets may be quite useful, even for external programs, in the following three scenarios:

1. If your program needs to connect to many different SAP systems, you only need to keep logon information for one of them and you can request an SSO ticket while logging on to that system. You can then use this ticket to log on to the other SAP systems, provided the same user ID exists in all systems and a kind of trust relationship has been set up between them. You can accomplish this via transaction SSO2 (Workplace Single Sign-On Admin).

2. If your program acts as a kind of middleware component between two SAP systems (e.g., it receives calls from one system, does some kind of processing on the data, and then forwards the call into another SAP system or back into the same one), you can set up this program so it doesn't need any kind of user credentials. It simply registers itself at the sending SAP system's RFC destination, which is customized so that the SAP system sends an SSO ticket with every call; then, the program reads the ticket from the server connection and uses it to open the client connection to the receiving SAP system. Again, the prerequisite is that both systems use the same user management and that they trust each other.

3. A server program can also evaluate a logon ticket when it receives one. For this you need an additional SAP library: SAPSSOEXT.[7] You can use this library to verify the ticket and extract the SAP user ID. Then, you can use the user ID for authorization checks similar to those described in the section "Protecting the server against unauthorized access" in the second article in this series. The advantage of this method is that this ticket is almost impossible to forge, and in conjunction with SNC encryption, the server should be 99.9% secure against unauthorized access.

To make use of SSO tickets, you need to know three things:

1. How to ask the SAP system to send you such a ticket:

   – When logging in as a client, you need to specify a special parameter in addition to the usual logon parameters: `getsso2 = 1`. Upon successful logon, the SAP system then returns an SSO ticket for that user.

   – When registering at the SAP system as a server, you need to mark the checkbox "Send SAP Logon Ticket" when you define the RFC destination in SM59 (RFC Destinations: Display/Maintain); this feature is available with version 6.40 and later. The SAP system then sends an SSO ticket for the current user with each RFC request.

2. To gain access to the ticket once the SAP system sends it, either client or server, you can read the ticket from the connection handle using the API call RfcGetPartnerSSOTicket().

3. Once you have the ticket, you can open other client connections to any SAP system that shares the same user data. Just include the parameter `mysapsso2 = <value of ticket>` in the RFC_CONNECTION_PARAMETER array. This replaces the user ID and password parameters.

A ticket is usually valid only for a limited amount of time, such as one minute. You can customize the time span during which SSO tickets are valid for logon by setting the profile parameter login/ticket_expiration_time in the SAP system that creates the tickets. (Also, see the documentation for the profile parameter login/create_sso2_ticket.)

Secure Network Communication (SNC)

In our opinion, SNC is one of the most important features of external RFC communications. Unfortunately, it is seldom used. Nearly everyone uses HTTPS instead of HTTP when transmitting sensitive data, but in the world of RFC communications, the use of SNC is not yet as widespread as it should be. One reason may be that setting up SNC in an SAP system is complicated. It is, however, worth the effort. For more information, see the *SNC User's Guide* available at http://service.sap.com/security, following the menu path Security in Detail → Secure User Access → Authentication & Single Sign-On. Once you have set up SNC for the SAP system, it's easy to use for the RFC library. Before you can use SNC in an external program, you need an extra third-party security library that provides one or all of the following features, depending on the product you choose:

1. Safeguarding the integrity of the transmitted data with digital signatures.

[7] For more information, search for "SAPSSOEXT" on the SAP Help Portal at http://help.sap.com under SAP NetWeaver.

2. Protecting the secrecy of the transmitted data by using encryption.
3. Providing secure user-authentication mechanisms, for example, through the verification of X.509 certificates.

Common products you might use are Secude IT Security, MIT's Kerberos 5, or Microsoft's NT LAN Manager (NTLM). Typically, the extra library you need comes with the security solution. So, if a security solution is already installed on the front end where the external program will run, this library is automatically available to your RFC program. If the solution is installed only on the SAP host, you need to ask the SAP system administrator for a copy of it.

To use SNC in programs based upon the SAP NetWeaver RFC library, you need to set the environment variable SNC_LIB to point to the location of the third-party security library in the environment where the RFC program will run.

The Network Interface (NI) layer included in the RFC library then loads the security library and uses the security library's functions to sign, verify, encrypt, or decrypt each data packet written to or read from the network sockets that correspond to your RFC connections. All third-party security libraries provide a common interface for these functions, known as the General Security Services-API (GSS-API), which enables you to plug-and-play the libraries of different vendors without having to change your RFC program.

When opening a client connection to the back end or when registering at an RFC destination, you need to include the following additional properties in the logon parameters to activate SNC:

- SNC_QOP = x, where *x* can have the values shown in **Figure 4**:

  The values "8" and "9" need to be explained in more detail. They don't denote additional SNC_QOP values, but will be "mapped" to one of the first three SNC_QOP values at runtime. You should use the value "8" in your external program if you want to let the back end make the decision about which SNC_QOP value to use. Then, the RFC library will first ask the back end which SNC level to use ("1," "2," or "3") and then continue with that value. The value "9" can be used if your external program will run within several different landscapes, where different security solutions are in use. One of these security solutions might only support digital signature and encryption, "2," while another also supports user authentication, "3." By using the value "9," the external program can run in both environments without adjustment: it will automatically use "2" in the first environment and "3" in the second.

SNC_QOP value	Meaning
1	Digital signature
2	Digital signature, encryption
3	Digital signature, encryption, authentication
8	Default value defined by back-end system
9	Maximum value current security product supports

Figure 4 Possible values of x

>> Note

SNC_LIB needs to contain the path of the library plus its file name; the path alone will not suffice. This is the only environment variable that influences the behavior of the SAP NetWeaver RFC library, in contrast to the classic RFC library, in which close to 100 environment variables can have an effect.

- SNC_PARTNERNAME: This is the identity by which the back-end system is known within the landscape. It could, for example, be the Distinguished Name of the back end's certificate, if the security solution uses X.509 certificates. (See the security product's documentation as well as the *SNC User's Guide* for more information.)

  If the back end's identity, determined during a security handshake while opening the network connection, differs from the value in this parameter, then the RFC library — specifically, the NI layer — will refuse the connection. This is true for both the client and server.

- SNC_MYNAME: This is the identity by which the back end knows the external system. If the external program acts as a client, you must define this identity in the back end's security management, and you must map it to an SAP system user. Otherwise, the back end will refuse the logon.

  If the external program acts as a server, this identity must be the one defined in the SNC settings of the RFC destination in SM59. Otherwise, the back end will refuse the registration attempt at that destination.

  In most cases, the SNC_MYNAME parameter is optional because the third-party security library, which is part of the complete security solution installed on that machine, knows where to find its certificate store or key store and will use, for example, the Distinguished Name of the certificate being set up for the current operating system user if the security solution is certificate-based.

It doesn't take much effort to set up SNC on the external side. However, a few more things need to be done if your program is a server connected to several back-end systems and you want to set up a kind of trust relationship to some of them (e.g., if calls from one SAP system need more authorizations than calls from the other systems).

At runtime you can determine the SNC name of the system currently making the call from within your server function or checkAuthorization() function. Instead of taking the back end's system ID from RFC_ATTRIBUTES, you can use the back end's SNC name, which is much harder to forge because it has been verified through the security

solution. When you set up the server program, you need to obtain the SNC names of the SAP systems that you want to trust (or grant special access). Then, you need to pass these SNC names to the function RfcSNCNameToKey(), which passes them on to the corresponding GSS-API of the security library and obtains a unique key in a canonical form for each back end. (The *canonical* form is a kind of vendor- and platform-independent representation of the SNC names.) These keys are in binary format, and your program should keep them in a list. When an RFC request arrives in the function checkAuthorization(), you should call RfcGetPartnerSNCKey() on the current RFC_CONNECTION_HANDLE to get the identity of the current calling system and compare the key to the stored list of trusted keys. If the key is on the list and the fields that you want to check from RFC_ATTRIBUTES or the SSO ticket also match, then checkAuthorization() should grant access to this function module. Please see SAP Note 934507 for a code sample for the classic RFC library. The SAP NetWeaver RFC library works about the same.

Performance

We have done a few benchmarks using the Secude solution on SNC level 3 (signature, encryption, and X.509 authentication) and compared the results to a plain (non-SNC) RFC connection. The results of these tests were as follows:

- The logon process with Secude SNC (opening the network connection, certificate handshake, and the logon procedure within the SAP kernel) takes about 100 times longer than the plain user ID and password logon process.
- However, the transmission of data only takes 3% to 4% longer with Secude SNC than with a plain RFC connection.

So, if your program is designed to repeatedly open a new connection, make a single RFC call, and then close the connection again, and it does this thousands of times, you will experience a dramatic performance decrease by switching to SNC. Most of the time will be spent exchanging and verifying certificates.

Therefore, when using SNC, it is vitally important to reuse the connections (e.g., by setting up a connection pool). Then, the time to process one RfcOpenConnection() will be negligible compared to the high number of RfcInvoke() calls, and you will see only a small 3% to 4% increase in overall network time. In our opinion, that's acceptable. It's similar to comparing the overhead of HTTPS to that of HTTP.

If the time spent processing the data in the back-end system or the external program is a few magnitudes higher than the pure network time, then the slightly increased network time won't have any significant influence on overall performance. To illustrate this with an example, let's say the network time is 100 milliseconds (ms) and the processing time 100 seconds (a factor of 1,000 higher or a magnitude of 3). The total time would then be 100.1 seconds. If SNC increases the network time by 4% to 104 ms, the total time only increases to 100 seconds + 104 ms = 100.104 seconds, which is negligible.

>> Note

You should always perform the comparison based on the binary SNC *keys*, not on the SNC *names*! The SNC names are only a string representation of the underlying SNC key (e.g., for display in a UI or a log) and are not necessarily unique. Also, when you compare a string of two SNC names, they may differ while the underlying "identity" is the same.

To see the differences between how the classic RFC SDK and SAP NetWeaver RFC SDK handle SSO and SNC, see the section "SSO and SNC" in the sidebar that begins on page 16.

Conclusion

SAP NetWeaver RFC SDK is a big step forward in the area of RFC communications with external non-SAP components. One big advantage of this new tool is its enhanced usability and the clarity of the concepts being used. Many concepts that proved successful within SAP Java Connector (JCo), such as data containers, metadata descriptions, and automatic caching, have also been applied in the C world. This should significantly reduce the time and pain involved in developing an RFC application and getting it to work.

Another big advantage with this new SDK is that for the first time the external side supports virtually every RFC communications feature that the latest SAP kernel version supports. Therefore, every feature that you can use for communicating between two SAP systems you can now also use to communicate between an SAP system and an external component, including a few features that are not yet available with JCo 2.x. The only limitations we have seen in the current article are due to the SAP NetWeaver RFC library's not having its own queuing engine (it relies on the queuing capabilities of the back end to handle qRFCs) and not having its own user management, so it can't use the full functionality of SSO tickets.

In the future, the SAP NetWeaver RFC library will most likely be kept up-to-date with recent developments in RFC communications — background RFC (bgRFC), class-based exceptions, or Remote Method Invocation (RMI). Once SAP builds a feature into the kernel and delivers it to the customer base, it should also be available very soon thereafter to the SAP NetWeaver RFC library (for more information on bgRFC, see the article "Increase the efficiency of your RFC communications with bgRFC — a scalable and transactional middleware framework" in the May/June 2007 issue of this publication).

RFC remains a reliable and highly efficient communications protocol that you can use to exchange any kind of data with an SAP system, whether it's high-level business data (BAPIs and IDocs) or low-level binary data (via RAW or XSTRING types).

Comparing SAP NetWeaver RFC SDK to the classic RFC SDK

This sidebar compares three advanced features of SAP NetWeaver RFC SDK programming to the classic RFC SDK.

- tRFC and qRFC
- RFC callback
- SSO and SNC

Please note the following differences and similarities.

tRFC and qRFC

The processing of tRFCs is quite similar to that in the classic RFC SDK. The main difference is that the classic RFC SDK supports only transactions (LUWs) that consist of a single function module, while the SAP NetWeaver RFC SDK can also process transactions in which multiple function modules are bundled into a single LUW. Other than that, the two SDKs feel pretty much the same. The following functions are approximately equivalent:

Client case:

Classic RFC SDK	SAP NetWeaver RFC SDK
RfcCreateTransID()	RfcGetTransactionID()
RfcIndirectCallEx()	RfcSubmitTransaction()
RfcQueueInsert()	None. Use queueName != NULL in RfcCreateTransaction()
RfcConfirmTransID()	RfcConfirmTransaction()

Equivalents for the data-container functions RfcCreateTransaction(), RfcInvokeInTransaction(), and RfcDestroyTransaction() don't exist in the classic SDK. They enable the user to bundle several function modules into one LUW, and the classic SDK doesn't support that feature.

Server case:

Classic RFC SDK	SAP NetWeaver RFC SDK
RfcInstallTransactionControl2()	RfcInstallTransactionHandlers()
RFC_ON_CHECK_TID_EX	RFC_ON_CHECK_TRANSACTION
RFC_ON_COMMIT_EX	RFC_ON_COMMIT_TRANSACTION
RFC_ON_ROLLBACK_EX	RFC_ON_ROLLBACK_TRANSACTION
RFC_ON_CONFIRM_TID_EX	RFC_ON_CONFIRM_TRANSACTION

There are two more features we should mention here:

- In the classic RFC SDK, you can only install the four transaction-control callback functions globally. In SAP NetWeaver RFC SDK, you can install them either per system ID or globally (systemID=NULL).
- In the classic RFC SDK, RfcQueueInsert() has an additional parameter, qcount, which is supposed to allow you to specify a certain position for the LUW within a queue when you use qRFC. However, the back-end systems never supported this feature, so it's always been ignored. Therefore, in SAP NetWeaver RFC SDK, SAP has removed the qcount parameter.

Continues on next page

Continued from previous page

RFC callback

The classic RFC SDK handles RFC callbacks in a different way from the SAP NetWeaver RFC SDK:

- On the client side, the client needs to install a server function for the callback function module. That part is pretty similar between the two SDKs. But to execute the callback function module, the client needs to check the return code from `RfcReceiveEx()` or `RfcCallReceiveEx()`. In the case of `RFC_CALL`, the client first needs to start its internal server-processing machinery (e.g., `RfcDispatch()`), and afterwards continue with its own client call by calling `RfcReceiveEx()` again.
- On the server side, the server function in the classic RFC SDK first has to call `RfcGetData()` to read the original client request from the connection the client has opened. Then, the server function can use `RfcCallReceiveEx()` to execute a callback via this client connection. However, the server must perform the callback before it sends any response data for the original client request over that connection via `RfcSendData()`. If you use `RfcCallReceiveEx()` before `RfcGetData()` or after `RfcSendData()`, the server program will crash because the underlying CPIC connection will be in an incorrect state at that point (e.g., the connection will be in `READ` state when you try to issue a `WRITE` to it). In the SAP NetWeaver RFC SDK, the developer has been relieved of the burden of personally handling these low-level details. The SAP NetWeaver RFC library handles them internally, so it always guarantees a consistent state of the underlying CPIC connection.

SSO and SNC

The handling of SSO and SNC is actually almost identical between the classic RFC SDK and the SAP NetWeaver RFC library. The following functions are equivalent:

Classic RFC SDK	SAP NetWeaver RFC SDK
RfcGetTicket()	RfcGetPartnerSSOTicket()
RfcSncPartnerAclKey()	RfcGetPartnerSNCKey()
RfcSncPartnerName()	RfcGetPartnerSNCName()
RfcSncNameToAclKey()	RfcSNCNameToKey()
RfcSncAclKeyToName()	RfcSNCKeyToName()

>> Note

SAP NetWeaver RFC SDK does *not* support SSO version 1; it *only* supports SSO version 2.

The logon parameters remain the same (e.g., `GETSSO2` and `MYSAPSSO2` for using SSO and `SNC_PARTNERNAME`, `SNC_MYNAME`, and `SNC_QOP` for using SNC).

The `SNC_MODE` parameter isn't necessary in the SAP NetWeaver RFC library. If the other SNC parameters (`SNC_PARTNERNAME`, `SNC_MYNAME`, and `SNC_QOP`) are found by the SAP NetWeaver RFC library, it will use SNC; otherwise, it won't. Also, the function `RfcSncMode()` no longer exists: If SNC is not active, then a call to `RfcGetPartnerSNCKey()` or `RfcGetPartnerSNCName()` returns `NULL`.

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

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