Master the five remote function call (RFC) types in ABAP

Part 1 — A comprehensive guide for SAP programmers and administrators

by Masoud Aghadavoodi Jolfaei and Eduard Neuwirt

Masoud Aghadavoodi Jolfaei
Development Architect,
SAP AG

Eduard Neuwirt
Senior Developer,
SAP AG

(Full bios appear on page 106.)

Although new communication protocols, such as HTTP, HTTPS, and SOAP, have been gaining traction in recent years, remote function call (RFC) remains the protocol of choice for SAP developers, system architects, and administrators who need to connect their SAP systems together. This is not an accident: Since its invention by Gerd Rodé and Rainer Brendle in 1993, RFC has become a trusted, widely supported protocol that is deeply embedded into nearly all major enterprise systems.[1]

Within SAP landscapes, RFC continues to serve as the basis of nearly all major integration points, including Application Link Enabling (ALE)/Intermediate Documents (IDocs), ArchiveLink, and links to more recent systems, such as SAP NetWeaver Portal and SAP Master Data Management (MDM). If you're a developer, you've probably used RFC to call function modules on other SAP systems, or used one or more of SAP's standalone RFC libraries[2] to connect to an SAP system from Java, C, C++, or .NET.

But did you know that your SAP system offers a transactional RFC model — that is, one that allows you to specify several RFC calls that should either succeed or fail as a group? Did you also know that to achieve balanced distribution of the system load you can have the system execute multiple RFCs at the same time and process them in parallel and then call your program back when the task-specific results become available? If not, then read on!

This article, the first part of a two-article series, takes you on a brief tour of the five RFC types available to you within your SAP Basis 4.0B or higher systems, and teaches you how to use them within your ABAP programs. This

[1] RFC's strength lies primarily in its simplicity and ease-of-use. Many other technology infrastructures from this time, such as DCE-RPC, COM, or CORBA, need huge books to describe their configuration and programming capabilities, whereas users can understand RFC intuitively and quickly.

[2] The RFC libraries are available on the SAP Service Marketplace (http://service.sap.com/swdc/).

knowledge is especially important for both developers and administrators who support SAP programs.

> ***Note!***
>
> Many of you may be familiar with one or two of these types. If so, feel free to skim those sections, or just jump ahead. Most sections are self-contained.

In this article, we begin by explaining some essential RFC terms and concepts, including the building blocks of the interaction between a client session and a server session, and then discuss the five types of RFCs available to you, and when, why, and how to use each of them. In the second article, which will appear in an upcoming issue of this publication, we cover how to design and develop remote function modules (RFMs), as well as how to perform user authorization checks and maintain RFC destinations, and more advanced topics, such as serialization and deserialization of ABAP types and the handling of exceptions and messages in your programs. For each of these topics we help you avoid some common pitfalls that plague even the most experienced developers by providing best practices — for example, how to effectively apply RFC authorization objects, nested structures, and compatible extensions of structures, and how to avoid errors due to character conversions. There's something for everyone here, so let's start with some basics.

Baseline terminology

Before diving into specifics on the RFC types, there are a few terms and concepts used throughout the articles that you need to know. Feel free to skip ahead as needed.

- **SAP Web Application Server (SAP Web AS):** Each SAP Web AS server consists of a gateway, Internet Communication Manager (ICM), dispatcher, and set of work processes (**Figure 1**). The gateway and ICM are interfaces for communication with other application servers within the same SAP system, other SAP systems, or non-SAP systems. The gateway process handles the physical RFC communication with the remote systems (the focus of this article). The ICM process handles the communication for standard protocols (e.g., HTTP, HTTPS, and SMTP) with the remote systems. The dispatcher is the link between the work processes, the gateway, the ICM, and users' front-end SAPGUIs. Older systems, such as SAP R/3 4.6C, run on top of SAP Basis, which does not have the ICM components. The gateway and dispatcher components are essentially the same, however, so this article still applies even if you're not running SAP Web AS.
- **Work processes:** Each SAP Web AS system contains several work processes running in parallel. These work processes perform the actual processing the system or user requests (e.g., dialog requests, locking/unlocking database tables, etc.). There are five types of work processes on a system:
  - Background work process for processing background jobs (e.g., long-running or resource-intensive reports or programs)
  - Dialog work process for dialog and RFC communication (explained next)
  - Enqueue work process for locking and unlocking database entries that can be accessed concurrently
  - Spool work process for spool and printing tasks
  - Update work process for processing update task function modules

  All work processes contain the same components (i.e., taskhandler, dynpro, and ABAP processors). The type of work process determines the kind of tasks for which it is responsible in the application server.
- **Dialog work processes:** Dialog work processes execute the dialog steps of ABAP programs, and are also used to process RFC sessions. The

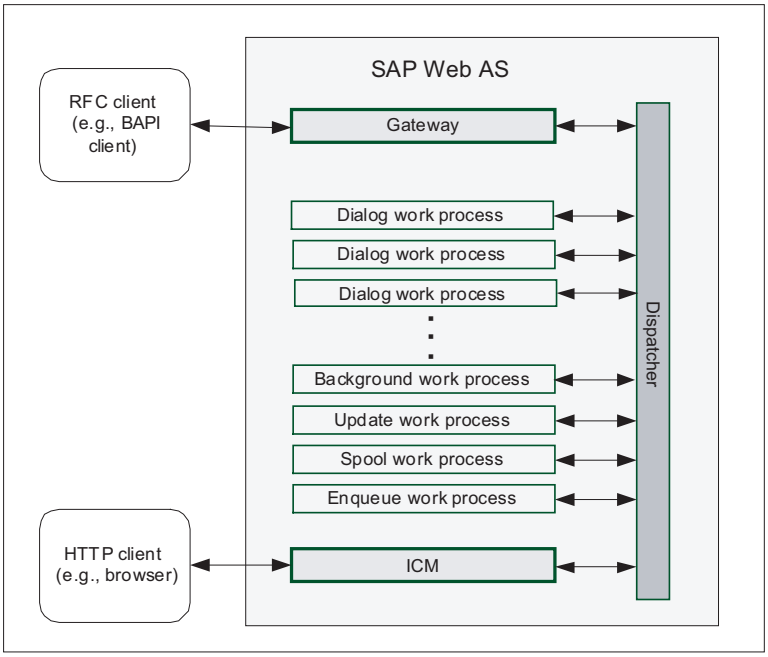


Figure 1 Work processes in an SAP Web AS

dispatcher distributes dialog and RFC processing requests among the available dialog work processes on the application server.[3]

- **Logon session:** A logon session is used to load and store all of a user's data after he or she has logged on. A logon session can be created either by a user explicitly logging on via SAPGUI (called a SAPGUI session) or by another SAP system via an RFC (called an RFC session). The user account for the RFC logon in the target system is called an RFC user.
- **Roll out, roll in:** On occasion, the processing of a program by a dialog work process encounters a situation in which the system must wait for some event before continuing — for example, when the system displays a screen to a user and waits for user input (i.e., at the end of a dialog step); when the program makes a synchronous RFC call, requiring the system to wait for the results before continuing; or when the system encounters a `WAIT` statement in the program's code. To optimize resources, SAP designed its systems to be able to "roll out" the program from memory while it waits, freeing the work process to work on another task. When the program processing can be resumed — such as when the results of a synchronous RFC call are received — the program can then "roll in" again. This design maximizes the utilization of the available pool of work processes.
- **Logical system:** This term is derived from the Application Link Enabling (ALE) technology, and uniquely identifies an application in a scenario in which multiple applications store their data in a single database. In the SAP world, a logical system corresponds to a client in a system — for example,

[3] You might be wondering why *dialog* work processes process RFC requests. This was an architectural choice by SAP, because dialog work processes have the unique capability of being able to "roll out" (or "set aside") a given task when it is waiting for some reason (e.g., a reply by the remote system to which the call is made). This design lets the SAP application server handle more RFC requests with a limited number of processes. The other reason is that dialog processes have a timeout feature, which helps to avoid work processes with long-running RFC tasks. By default, this timeout is set to five minutes (it can be adjusted via profile parameter rdisp/max_wprun_time).

client 000 in SAP system SRC represents a different logical system than client 800 within SAP system SRC.

- **Logical Unit of Work (LUW):** This is the time between two consistent states on the database or between the indivisible sequences of database operations that are concluded by a database commit. This span of time is commonly referred to in the database world as a "transaction" (you can understand why SAP chose not to use that word given its alternate meaning in the SAP context). An LUW is executed either completely or not at all by the database system. If an error is detected within an LUW, it is possible to revoke all database changes since the beginning of the LUW with the help of a rollback.
- **RFC client, RFC server:** An RFC client (*or* an RFC server on an SAP Web AS executed in a logical system) requests that an RFC server execute a function or program.
- **Remote function module (RFM):** This function module is remote-enabled and can be executed via RFC. The decision to remote-enable an SAP function module is made on an individual basis via the "Remote-enabled" radio button on the function module's attributes screen.[4]

With these definitions in hand, let's begin our journey into the world of working with RFC in SAP Web AS.

Note!

Interestingly, SAPGUI, which is typically thought of as a client, also acts as an RFC server because it can host user interface (UI) controls, which can be controlled via RFC. For example, the SAP application server loads the SAP Easy Access Menu tree control, with which you are undoubtedly familiar, with its contents by calling an RFC function resident within the SAPGUI software. Therefore, not only does SAPGUI make requests of the SAP application server, but it also listens for requests from the SAP application server. If you're interested in further reading on the subject, google "SAP Control Framework," or search for it on the SAP Help Portal at http://help.sap.com/.

The RFC communication process

The main task of the RFC runtime is to facilitate the execution of function modules in a remote system, be it an SAP system or an external program linked with one of the SAP-supplied RFC libraries, the Java Connector, or the .NET Connector. Fortunately, the details (such as specific mapping requirements of parameter values) of which type of system the user is calling are all handled by the RFC runtime. The ABAP code the system requires to call the external function is the same in all cases.

One of the first things you need to understand to work with RFC effectively is how RFC calls are actually communicated and executed. **Figure 2** illustrates the request-response phase through which each RFC call goes. The RFC client system (in the example, the SAP Web AS CLN) is shown on the left. The SAP Web AS SRV, shown on the right, hosts the function we want to call, and therefore is the RFC destination, or more specifically the RFC server.

The communication cycle begins with a work process (i.e., a dialog or background work process initiated by an application on the client) that generates an RFC request. The RFC request is serialized in the work process for transmission over the network and passed to the target system.

The RFC request can include:

- **Connection parameters:** These values include the host name and port number of the target server.

[4] To enable this option, display the function module in change mode via transaction SE37, and then navigate to the Attributes tab.

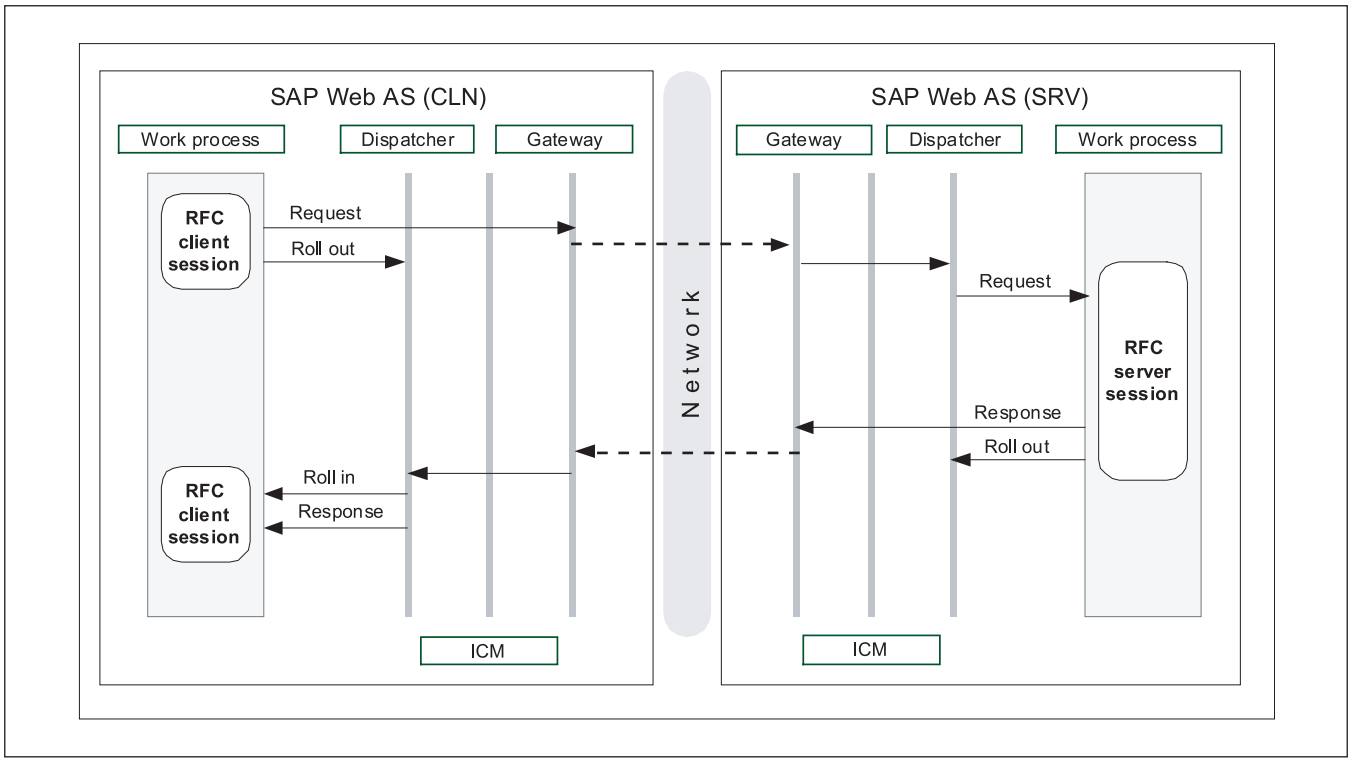


Figure 2 RFC process communication cycle

They are the technical settings maintained in the RFC destination.[5]

- **Logon information:** This information includes the associated user attributes (e.g., language, client, user name, etc.).
- **RFC runtime data:** This is the name of the function module, as well as any specified input parameters (e.g., exporting, importing, changing tables parameters).

The local gateway (on the RFC client) opens a TCP/IP connection to the target gateway (on the RFC server) and transmits the request to it. The target gateway then allocates a task at the target dispatcher, which identifies an available dialog work process to process the request. The work process on the target system then deserializes the request,[6] executes it, and returns the result following the same sequence in reverse (i.e., dispatcher, gateway, network, gateway, etc.). As indicated in Figure 2, rolling session data in and out as needed (e.g., while the client waits for the response, after the server issues its response, etc.), freeing both memory and work processes to perform

> ***Note!***
>
> For maximum performance and security, the RFC runtime transmits the connection parameters and logon information with the first request only.

[5] Typically we distinguish between two different destination types: the RFC destinations maintained in transaction SM59 and the so-called dynamical destinations (e.g., *<hostname>_<system id>_<system number>*, such as binmain_BIN_53), which are not available in transaction SM59. In that case, you then specify the connection parameters in the destination ID (e.g., binmain_BIN_53, where binmain is the hostname and 53 is the target service number).

[6] That is, the target system converts the request from a stream of bytes into a memory structure representing the inbound RFC call so that the target program can read parameters and other RFC metadata included in the request.

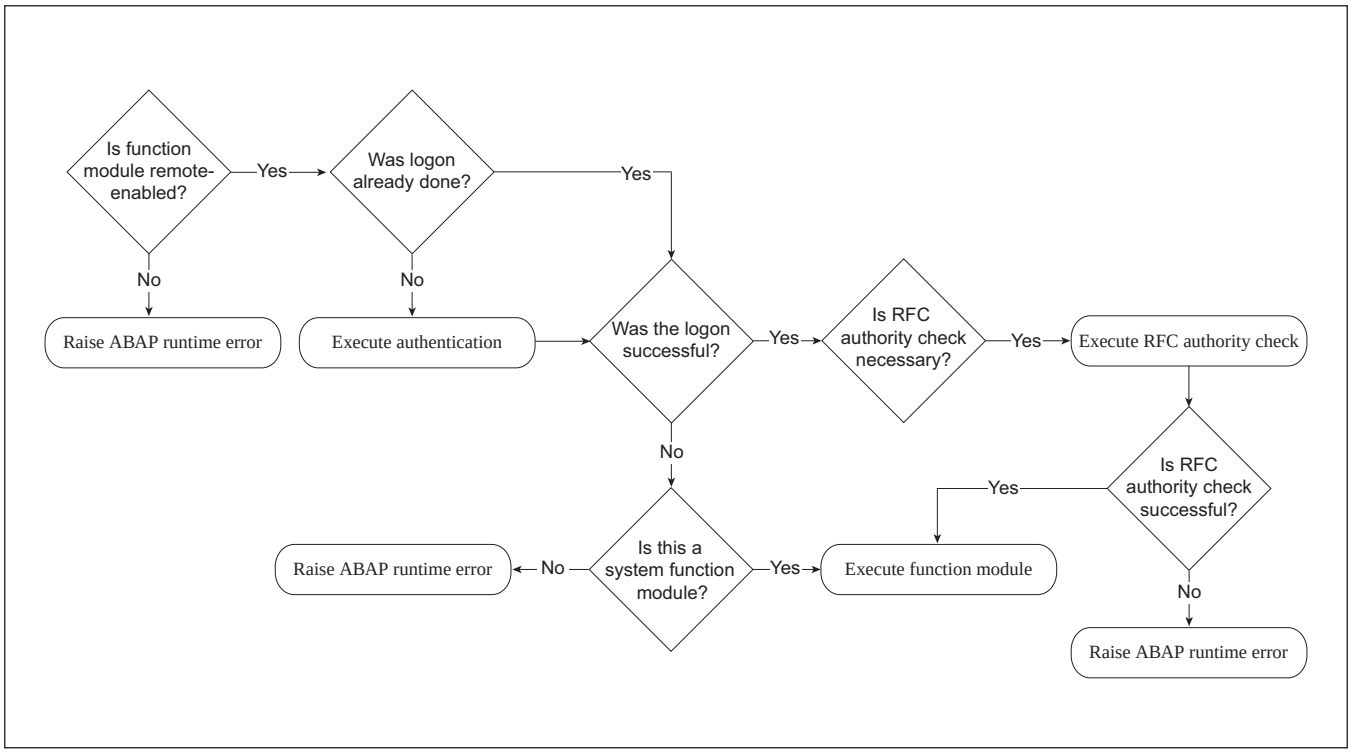


Figure 3 RFC work process flow on the target system

other tasks when it is not actively needed, optimizes performance on both the client and target systems.

Now let's take a closer look at what specifically takes place within the work process on the target system (RFC server). Using **Figure 3**, let's discuss the following actions the RFC server performs in the order in which they are executed:

1. **Checks that the function module is available and remote-enabled.** A function module can only be executed remotely if it is properly identified in the Function Builder as a remote-enabled function module.

2. **Creates an RFC session.** If this is the first RFC request, an RFC session is created with the delivered logon information. You may find it interesting that this happens *after* the function module availability check.

3. **Performs an RFC authority check.** Depending on the setting of the application server's profile parameter auth/rfc_authority_check, the RFC runtime checks to make sure that the users are authorized to execute the function module to be called.[7]

4. **Calls the function module.** If the previous steps are successful, the remote-enabled function module is executed. Otherwise, an appropriate ABAP runtime or error message is raised and passed back to the caller program.

> ***Note!***
>
> A few system function modules, those belonging to the function group SRFC, can be executed even if the request is missing logon information or includes invalid logon information (e.g., RFC_PING and RFC_SYSTEM_INFO).

[7] You can find a detailed description of the auth/rfc_authority_check profile parameter in transaction RZ11. The RFC authority check is active by default.

Note!

You manage authorizations at the function group level (and not at the function module level). Granting a user authorization to a function group (authorization object S_RFC) enables the user to execute all RFC-enabled function modules within that group.

Now that you have a better understanding of the RFC work process flow, let's delve into the five types of RFCs available on SAP Basis 4.0B and higher systems, including all SAP Web AS systems.

The five basic types of RFC

There are five basic types of RFCs from which to choose, each well suited to specific scenarios:

- Synchronous RFC (sRFC)
- Asynchronous RFC (aRFC)
- Transactional RFC (tRFC)
- Queued RFC (qRFC)
- Parallel RFC (pRFC)

In the following sections, we explain the unique role of each RFC type, review the ABAP commands associated with each one, and the situations in which each excels. We follow up with a consolidated quick-reference matrix to which you can refer in your daily work. Let's begin with the most common type — synchronous RFC.

Synchronous RFC (sRFC)

The synchronous RFC (sRFC) is the most common RFC and is ideal when you need the results of a function module immediately after its execution. sRFC also means that both systems (i.e., RFC client and RFC server) must be available at the time the call is made. As shown in **Figure 4** on page 88, the sRFC is made by calling a function via the `CALL FUNCTION` statement with extension `DESTINATION`, which identifies the target system of the call.[8]

Note!

To improve performance, the system can roll out the context of the caller program while it waits for the response of the RFM. As soon as the local gateway receives the response, however, the activation (roll in) of the program context is triggered. For maximum performance, the connection to all target systems is retained for the entire duration of the process of calling an ABAP program (or ABAP transaction). An RFC session is created for each unique destination ID encountered in your program (see `myFirstDestination` and `mySecondDestination` in Figure 4).

There are situations in which it is not reasonable to keep the RFC pipe (i.e., the communication) and its associated remote session open for a whole transaction. For example, when the RFC is used as a stateless call on a different server or in a separate session, this call still has some influence on the transaction flow of the caller execution time because the call consumes considerable application server resources. In this case, you can close the RFC pipe and the associated remote session with the help of the function module RFC_CONNECTION_CLOSE.[9]

[8] The `DESTINATION` clause appears in all of the RFC types, although it is optional in instances of aRFC, tRFC, and qRFC. If you omit it, the value of `DESTINATION` is set internally to `NONE`, which means that the function executes on the local system.

[9] A stateless call is one in which subsequent calls do not depend on the outcome of any previous call or on which user is making the call — an example of a stateless call is a catalog search.

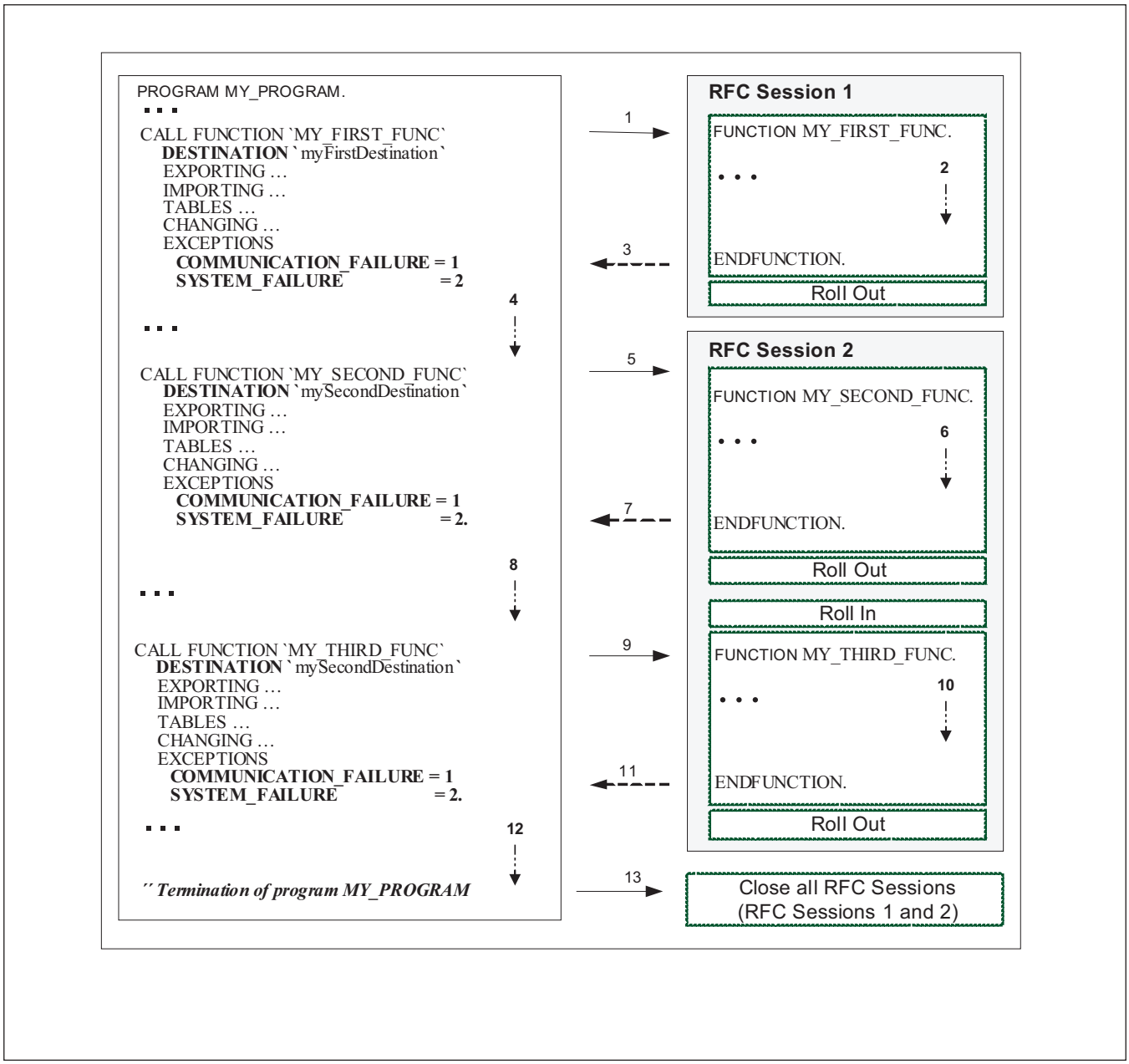


Figure 4 Calling a function module synchronously from ABAP (sRFC)

The sRFC call is passed immediately to the target system, and the caller program is halted until the response of the RFM is received. Processing then continues with the next line after the `CALL FUNCTION` statement, so the actual results of the function are available for use within the main program. This also means that you can place error-handling logic immediately after the `CALL FUNCTION` statement to prevent downstream calls if an exception occurs during a call.

Every remote call of a function module that is made using the sRFC interface defines a separate context in

the target system. The function group of the function module is loaded into an internal session of the context and retained. This means that if repeated calls of function modules belonging to the same destination and the same function group are made, the global data of this function group can be accessed collectively. A connection and its context are retained until the connection is explicitly closed, or until the calling/caller program is finished. To close a connection explicitly, you can use the function module `RFC_CONNECTION_CLOSE` or the API function `RfcAbort` or `RfcClose`.

Note!

A nice feature of the system is that, upon termination of an RFC client program, the system automatically closes all active sRFC connections for that program and eliminates the associated RFC sessions.

Note!

If the RFC destination is omitted or left blank (e.g., `CALL FUNCTION <function name> DESTINATION <blank>`), the function module gets executed locally, like a normal function call, and not as a remote function call.

Asynchronous RFC (aRFC)

Asynchronous RFC (aRFC) is a great option when you want to improve the performance of your ABAP program by having the system call one or more function modules in parallel rather than force your program to wait for results before continuing. In other words, with aRFC, the system immediately passes control back to the caller program so it can continue processing after the `CALL FUNCTION` statement. The system calls the function modules immediately. There are two forms of aRFC, and the specific form of aRFC you choose depends on whether you want to process values from the function module or keep a connection to the remote system open for subsequent calls (which you might do for performance reasons).

Note!

Both sRFC and aRFC are distinct programming approaches, each with a unique ABAP syntax. However, we still refer to sRFC and aRFC when introducing the other RFC types (tRFC, qRFC, and pRFC) because these other types use sRFC and aRFC behind the scenes — adding their specific own functionalities (i.e., queuing, transactional grouping, and parallel processing) to the process.

Let's further explore aRFC by tackling the question of whether or not to process the parameter values from the function module. The easiest option, as you might imagine, is not to make the system return the values to you. This form of the process is called *asynchronous RFC (aRFC) without response* (**Figure 5** on page 90). This means that, when the RFM is triggered, the return values are not made available to downstream calls or to the calling program. This process is useful in many situations, in which, for example, neither the caller program nor the downstream function calls need to receive or evaluate the output values of the RFM.

All you need to do to use this form is add a `STARTING NEW TASK <task name>` clause to the `CALL FUNCTION`, where <*task name*> can be any unique name you want (its significance will become clear shortly). As shown in Figure 5, the system begins to execute a call immediately in a dialog work process. A connection is made to the remote system, and the RFM executes immediately within a new session. Once the function is complete, both the RFC session and the connection to the remote server close. This is, therefore, less than optimal from a performance

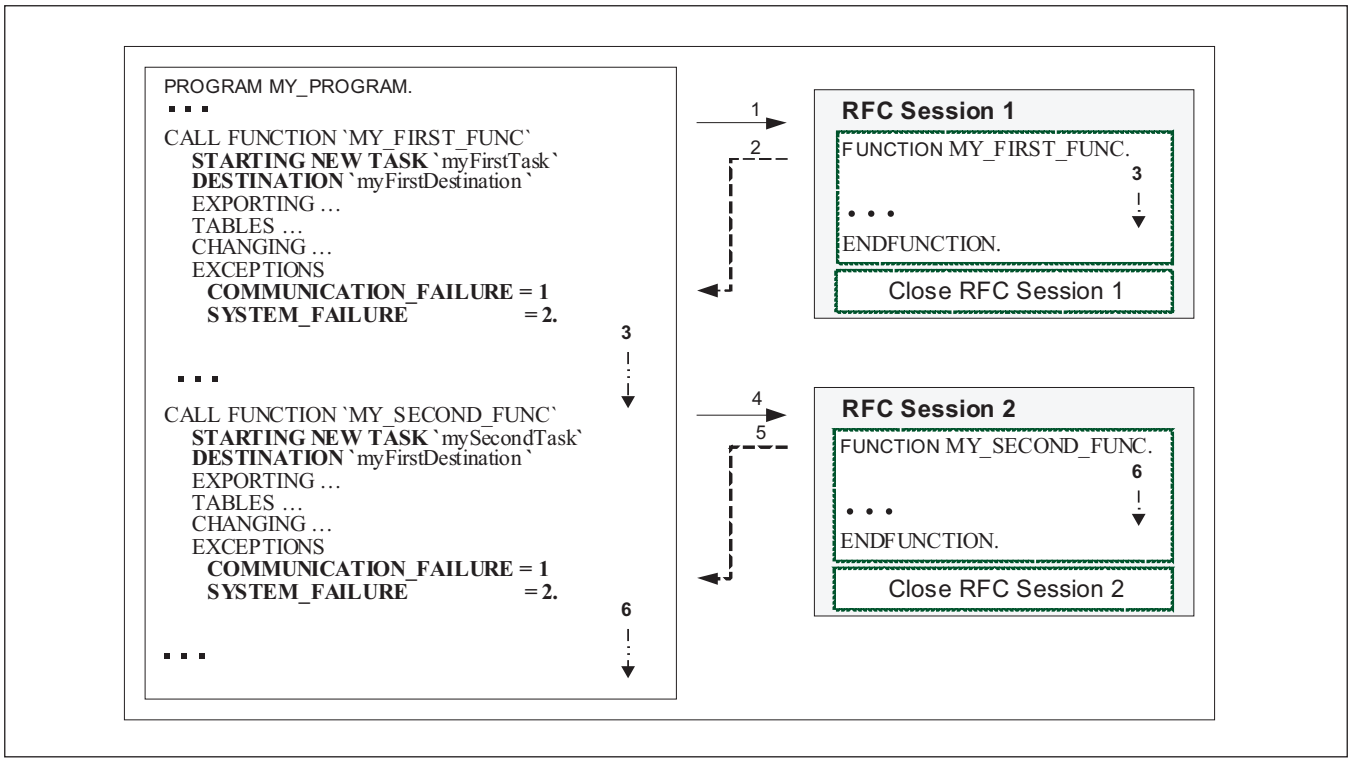


Figure 5 ABAP statements for aRFC without response

perspective if you need to make multiple aRFC calls to the same destination.

If you want to handle the results of your function calls — so you can, for example, take some conditional, follow-on action based on a function module's output, such as emailing an administrator if a failure occurs, or posting additional records using a dynamic order number generated by one of the functions — then *asynchronous RFC (aRFC) with response* is for you. Like the "without response" form, the `CALL FUNCTION` returns immediately, freeing the caller program from having to wait until the calls complete before continuing — in other words, the caller program executes the function immediately. Unlike the "without response" form, however, here you specify the name of a subroutine ABAP FORM (in the example, `FORM RESP_1`) that the system should invoke once *both* the function results are available and the caller ABAP program reaches a "synchronization point" statement, such as the `WAIT UNTIL` <*`logical expression`*> statement shown in **Figure 6**. You add a `PERFORMING` <*`callback form`*> `ON END OF TASK` clause to the `CALL FUNCTION` statement (in addition to the `STARTING NEW TASK` command we discussed earlier), and the callback subroutine includes a `RECEIVE RESULTS FROM FUNCTION` <*`function name`*> clause. (For more information on aRFC, see the sidebar "Things to keep in mind when choosing an aRFC with or without response" on page 92.)

Figure 6 also illustrates how the RFC process comes together. The RFC client sets up the connection and transfers the RFC data to its application server's gateway, without waiting for any acknowledgement. The caller program resumes processing after the `CALL FUNCTION` statement until the ABAP statement `WAIT` is reached. The associated task and destination names identify the RFC connections belonging to the aRFCs (in the example, the task names are `myFirstTask` and `mySecondTask`, and the destination is `myFirstDestination`). The call executes, and the subroutines are called in the order in which the responses arrive, assuming the main program has

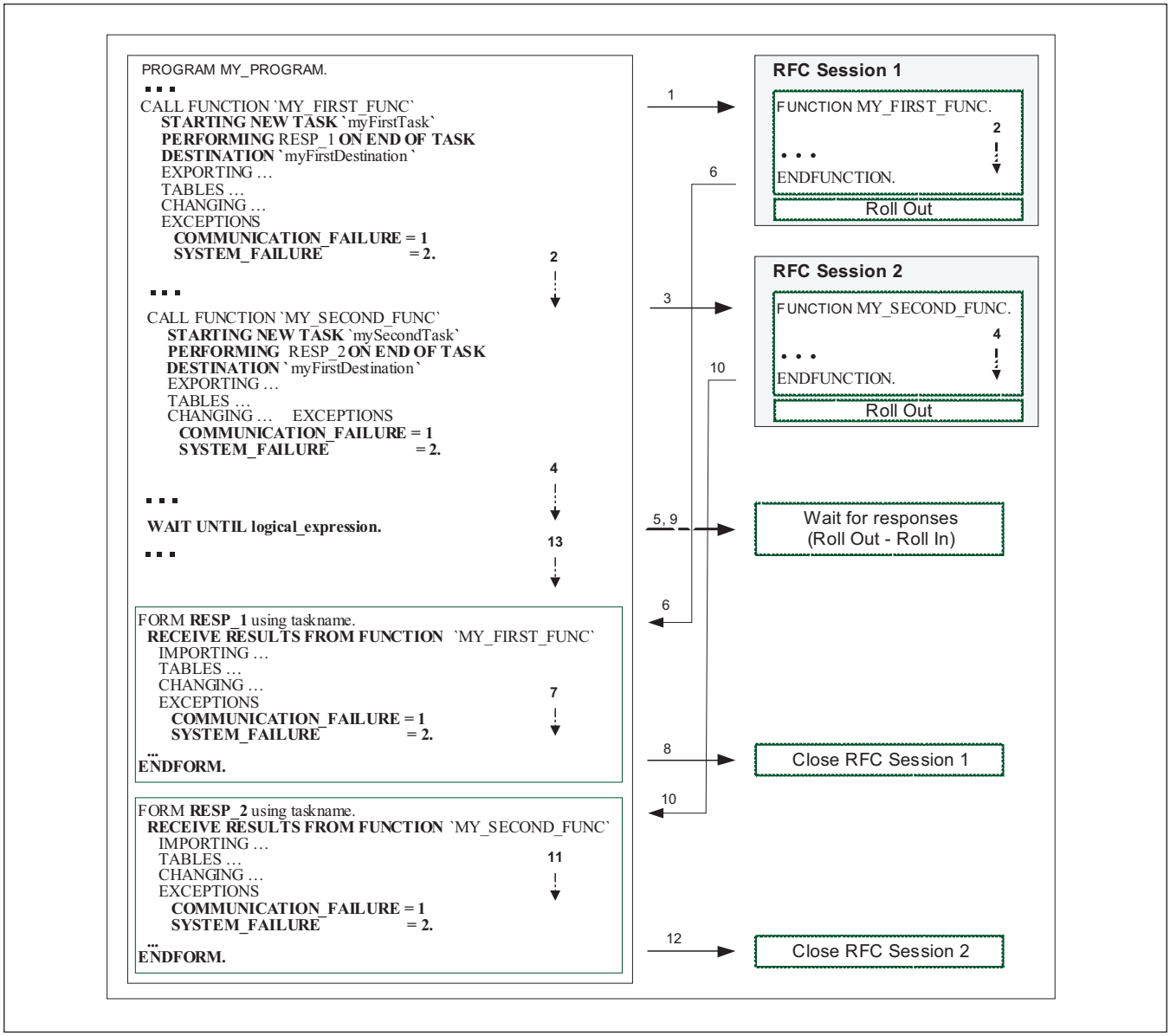


Figure 6 ABAP statements for aRFC with response

Note!

You cannot predict in what order the responses will come, and therefore in what order your subroutines will be executed. Make sure you take this into account when coding, and do not assume one code will execute before the other. If you need to control this sequence, either check the order internally or use qRFC (which we'll discuss shortly). Also note that you can only use qRFC if no responses are required or if an immediate processing of the remote function module is not required.

Things to keep in mind when choosing an aRFC with or without response

- In case of an aRFC without response, any error messages or exceptions raised during the execution of the RFM are not propagated to the RFC client and vanish. If this error information is essential for the RFC client processing, you have to choose the aRFC with response form.
- Remember, for the "call me back when the results are in" method (i.e., aRFC with response) to work, your ABAP program must give the system an opportunity to call the ABAP FORM to handle the results. The ABAP statements that trigger this type of interruption in processing include commands associated with dialog interactions (e.g., `CALL TRANSACTION` and `CALL SCREEN`), as well as an sRFC `CALL` or a `WAIT` statement. The recommended statement to initiate a synchronization point for the receipt of asynchronous responses is `WAIT UNTIL <expression>` (Figure 6). If a caller program terminates before receiving all the outstanding responses of the aRFCs, this might result in ABAP runtime errors on the RFC server.
- The need for a synchronization point has two implications: First, an aRFC with response will only save you processing time when you have processing to do while the aRFC call executes. Second, your callback subroutines have access to the global memory variables of your main program because they execute in the same process. This is good news if you need this capability; however, you can't predict the state of these variables since you don't know exactly when the callbacks might come.
- Consistent with SAP's move to object-oriented (OO) coding, SAP Web AS 6.40 or higher allows you to call a class method instead of a form routine in the `PERFORMING <callback form> ON END OF TASK` addition. For the method specification, the same types are permitted as for the `CALL METHOD` statement. Dynamic calls, in particular, are also supported.

reached the `WAIT` statement. RFC sessions and associated connections close as soon as the callback subroutine completes.

Your other option is to have the system retain the RFC session on the remote server after each RFC call, rather than discarding it. This yields improved performance when multiple functions are executed against the same destination in close succession. This is where the task name within the `CALL FUNCTION` statement gains importance.

If you compare **Figure 7** with Figure 6, you can see two key differences:

- On the left side, the addition `KEEPING TASK` of the `RECEIVE RESULTS` statement specified for the first function call tells the RFC runtime to keep the connection alive after the first call.

Note!

In the case of an aRFC without response, the connection closes directly after the successful check for availability of the remote function module and the logon procedure. If any errors occur after the connection closes (e.g., if the RFC authority check based on the S_RFC authority object fails), the caller program won't be informed (Figure 3).

- On the right side, you can see that the second function call is executed using the same task name and destination as the first function, and thus is in the

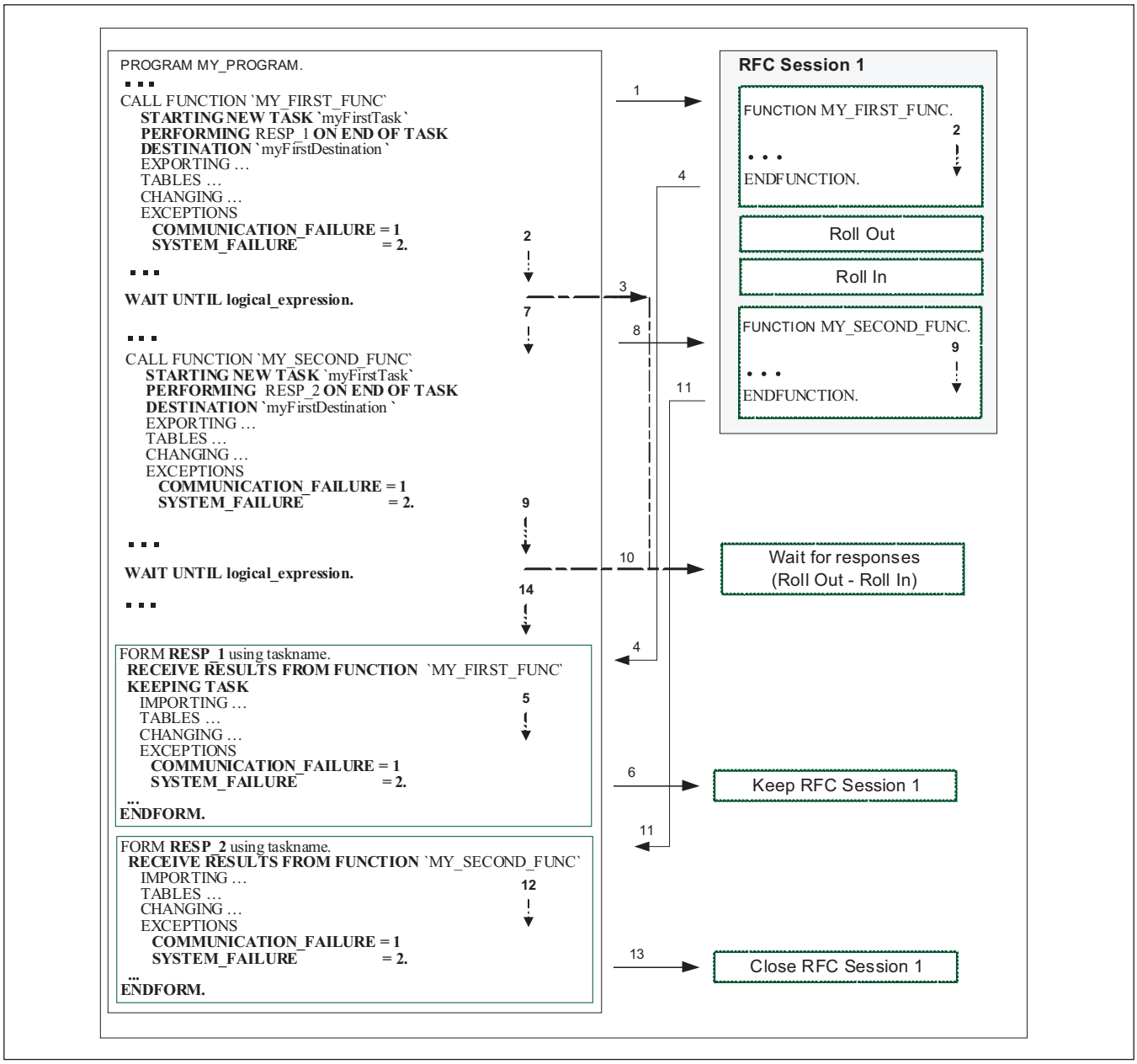


Figure 7 RFC session handling and keeping the remote session for aRFC with response

same RFC session as the first. When you use aRFC with response, the task name and destination identify the RFC connection and its RFC session. Since these attributes are available because of an open connection within the client context, this connection is retrieved and used for the subsequent call.

> ***Note!***
>
> As long as the associated RFC connection to a task remains open, you are not allowed to reuse the task name in the program.

Now let's look at what takes place in a transactional RFC.

Transactional RFC (tRFC)

A transactional RFC (tRFC) lets you group one or more function module calls together to a tRFC LUW, and ensures that each function module within the LUW is called exactly once — or not at all. In contrast to aRFC and sRFC, in tRFC the function modules belonging to a tRFC LUW are executed in order — if one issues an exception, the others do not get executed. tRFC is particularly important in scenarios in which you're posting business documents (e.g., invoices, shipping information, orders, etc.) to a system that might not be available at the time of posting. In this case, the tRFC framework tries to execute the RFC call at a later time. The tRFC infrastructure provides a monitoring transaction (transaction SM58) to track errors that occur during either the transmission of the call (communication errors) or the execution of the function module on the target system. In contrast, when dealing with aRFC and sRFC, you need to handle communication errors and to build your own mechanism for reissuing calls at a later time.

In summary, tRFC ensures that calls execute only once or not at all on each system, a requirement known as *Exactly Once (EO) execution*. Some business scenarios are even more stringent, however, requiring *Exactly Once In Order (EOIO) execution*, which is what qRFC provides over tRFC, as explained in the next section.

Figure 8 shows the syntax for calling a function module via tRFC. You simply add an `IN BACKGROUND TASK` clause to the `CALL FUNCTION` statement.

Here's how this works. When a program encounters a tRFC `CALL FUNCTION` statement, the RFC runtime records the call in a concealed internal table in the computer's memory, and then immediately returns control to the caller program, rather than calling the function module immediately. This is similar to aRFC, except that the RFC call is *not* executed immediately but rather held in a queue. As with aRFC, the `CALL FUNCTION` statement returns immediately, always succeeding with sy-subrc=0. Internally, the system assigns a unique ID (known as the transaction ID) to the calls for each destination (in the example, transaction IDs 4711 and 4712).

> ***Note!***
>
> Keep in mind that you don't have to call all function modules within your program using only one approach. You can call one module via tRFC, then the next via sRFC or aRFC, then the next via tRFC. Any tRFC calls you make will be grouped together for execution as described next.

The truly exciting part happens when the program issues a `COMMIT WORK` command. The recorded calls are executed asynchronously[10] in a special set of tRFC processing sessions created for each transaction ID (in the example, Session 1, Session 2, etc.).

> ***Note!***
>
> If a `ROLLBACK WORK` is issued instead of a `COMMIT WORK`, the internal queue of pending RFC calls is flushed — none of the calls are performed.

To summarize, tRFC is a great choice to use when the execution of function modules in a remote system has to be guaranteed, and when the progress of those calls should be monitored, especially for error behavior. The tRFC should always be used when the EO execution of the tRFC LUWs needs to be guaranteed,

[10] tRFC processing sessions are almost always executed asynchronously, except in the rare case when the caller program (the tRFC client) is running in a dialog work process (i.e., online) and the application server cannot afford the extra resources for the asynchronous session. In this case, the tRFC LUW is executed synchronously.

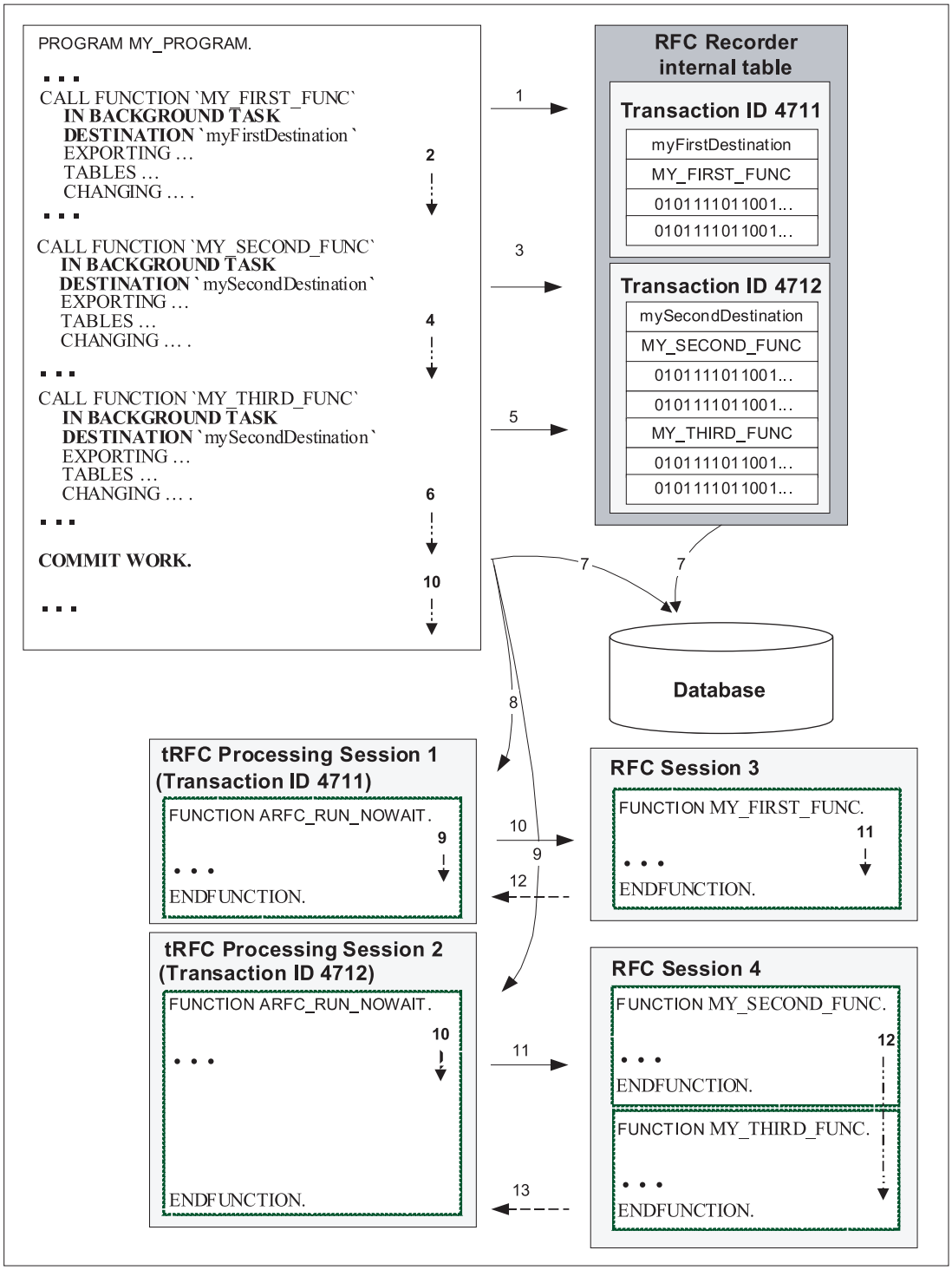


Figure 8 ABAP statements for calling function modules via tRFC

Note!

An important point to remember: Only calls to functions at the same destination are grouped together within a given transactional unit. So if you were to make three tRFC calls, two of which were to the same destination, those two would constitute one transaction, while the third would be placed in its own transaction. This means, unfortunately, that you can't group calls across multiple systems, which is sometimes an important business requirement (and is known as a "two-phase commit" in the IT world).

even when an RFC destination is not accessible at the time of processing, or when communication interruption can occur and automatic re-execution might need to be done. See the sidebar "Important considerations before choosing tRFC" below for more information on implementing this type of RFC.

Queued RFC (qRFC)

Transaction RFC (tRFC) is a great option when you want to have a guaranteed transactional execution (EO execution) of LUWs, but it does not let you control the execution order of multiple LUWs with any certainty, which is a common requirement (see the note on page 91). In these situations, you need to use qRFC.

Important considerations before choosing tRFC

You should be aware of a few important considerations before choosing tRFC:

- A `COMMIT WORK` is called on each remote system after *all* of the functions have executed without raising an exception. This is so that all function calls either succeed or fail as a unit. If one of the function calls raises an exception, downstream calls in the queue are ignored, and a rollback is issued for all previous calls.
- Errors returned by functions in parameters, such as BAPI `RETURN` structures (which might return an E in the `RETURN-TYPE` field) are *not* treated as failures by the RFC runtime. Only *exceptions* — i.e., messages of types E (Error messages), A (Abort or Termination messages), and X (Exit messages) issued via an ABAP `MESSAGE` statement — are treated as failures. So, carefully investigate how your remote function module returns errors if transactional behavior is important to you.
- For the automated rollback mechanism (see the first bulleted item) to work, the function modules called must *not* perform their own `COMMIT WORK` internally. You can check this by searching the function module code for `COMMIT WORK`, `CALL TRANSCATION`, or sRFC or aRFC statements (transactions called from within a function module perform a database commit at the end, making their changes irreversible by a subsequent rollback).
- Because the `CALL FUNCTION` statement always succeeds (with sy-subrc=0 as mentioned earlier), there is no point in using code error handling logic after this type of `CALL FUNCTION`. If the asynchronous call

Note!

Consider the following actions that must be executed in sequence: creation of a purchase order, modification of the purchase order, and deletion of the purchase order. Since each action needs to be committed before the next is executed, and because the steps must be executed in order, tRFC would not be the right choice for this process. With tRFC, each action is placed in its own session, and you cannot control the order of execution of the sessions. In this case, you need qRFC.

Queued RFC (qRFC), an extension of tRFC, solves the problem by offering fine control over the sequence in which tRFC sessions are executed. **Figure 9** on page 98 shows how the ABAP syntax for qRFC differs from tRFC: all that's needed is to call function `TRFC_SET_QUEUE_NAME` before the regular tRFC-style `CALL FUNCTION` statement. The call to function `TRFC_SET_QUEUE_NAME` tells the system into which queue to insert the next RFC call.

Figure 9 also illustrates how qRFC works operationally. Like tRFC, the recorded calls are saved into the database when a `COMMIT WORK` occurs in the calling program. In contrast to tRFC, however, a separate scheduler process controls scheduling and execution of qRFC LUWs.

fails, you have no opportunity programmatically (without using Workflow) to perform any follow-on processing. This is not to say that you can't monitor when these errors occur — all failed tRFC calls appear in the tRFC administration log (transaction SM58), where they can be re-executed if necessary. If you need to programmatically intercept failed asynchronous calls, use the aRFC with response approach.

- Perhaps you've noticed that tRFC LUWs are executed in parallel, instead of sequentially. This can cause a problem if the function calls in the second transaction need to be called after those in the first. One solution to this problem is to use qRFC instead of tRFC — qRFC goes a step beyond tRFC to let you specify a sequence in which various transactions can be executed. Another approach is to register the RFC destinations used in tRFC LUWs via transaction SMQS.

Caution!

This latter approach (registration in SMQS) is not guaranteed to work in all cases. For example, if, for some reason, one of the tRFCs fails (due to the target system being unavailable for example), the remaining (downstream) transactions will still execute. The transactions are not conditional upon one another, and execution in order is done on a "best efforts" basis.

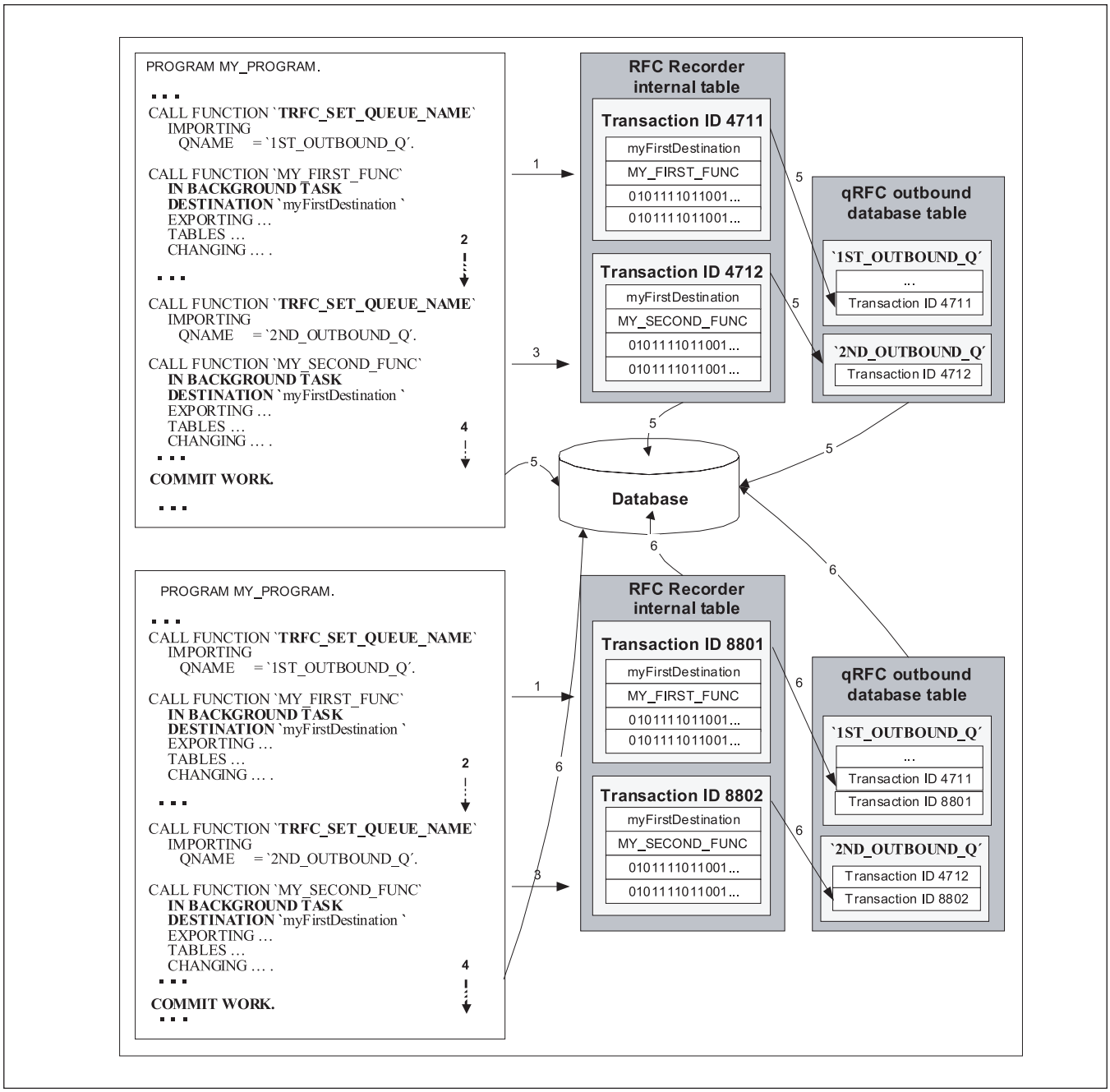


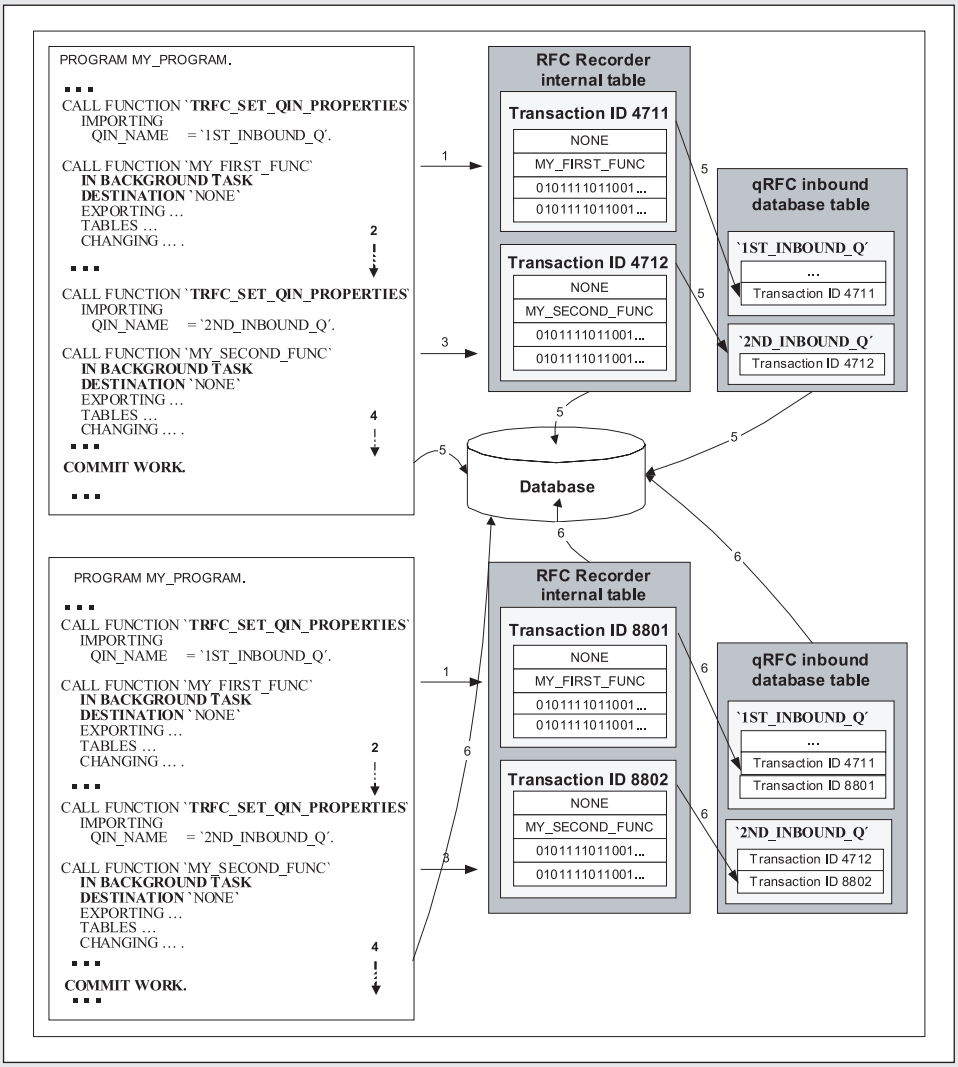
Figure 9 ABAP statements for calling function modules via Outbound qRFC Scheduler

The qRFC Scheduler serves two purposes:

- To ensure the execution order of qRFC sessions in a queue
- To hinder/stop the execution of subsequent LUWs in the same queue when a qRFC LUW is executed with an error, and to continue processing the queue after the erroneous LUW is removed or the error situation is resolved. For more information on the qRFC Scheduler, see the sidebar on the next page.

Outbound vs. Inbound qRFC Scheduler

Technically, there are two queue scheduler processes: an Outbound Scheduler and an Inbound Scheduler. The Outbound Scheduler controls the execution of LUWs in another logical system (i.e., when logging onto another client or with another user ID). The Inbound Scheduler controls the qRFC execution in the same system, user, and client (e.g., a qRFC call with destination `NONE` or no destination specification in the `CALL FUNCTION IN BACKGROUND TASK` statement). Figure 9 shows the ABAP syntax for using the Outbound Scheduler. The screenshot below shows the ABAP syntax for the inbound queue. To configure the qRFC outbound queues, use transaction SMQS, and to configure the qRFC inbound queues, use transaction SMQR.



In summary, you should use qRFC when EOIO execution of LUWs needs to be guaranteed (e.g., if a target system becomes inaccessible, or if communication is interrupted for some reason).

Parallel RFC (pRFC)

This final type of RFC, pRFC, is actually an extension of aRFC. Because it improves system performance, using pRFC is a prudent and responsible choice when you plan to make a large number of RFC calls asynchronously (SAP R/3 Material Resource Planning and Controlling applications use this kind RFC to speed up the execution of their business scenarios).

What do we mean by "prudent and responsible"? Well, in some cases, considering that executing a function module asynchronously requires much more system resources (i.e., memory, processing time, etc.) than synchronous execution, aRFC can overload your SAP server. After all, you're asking the system to initiate and manage the RFC calls while *continuing* to process your main program, and perhaps queuing up several more sessions to execute at the same time.[11] What's more, at any given point in time, the destination SAP server might run out of available memory or other resources it needs to run these asynchronous sessions as soon as they arrive. aRFC does not take any of this into account — it just continues to make its calls, without considering how the overload might impact other processes or users on the system.

pRFC, on the other hand, combats this in two ways:

- Allowing you to specify a *group* of application servers on which to execute the RFC calls, instead of just a single RFC destination. The RFC runtime has a built-in, load-balancing mechanism that distributes the RFC calls to application servers based on their available resources.
- Alerting you — by raising an exception — when none of the servers in the group has sufficient resources to process the request, so you can choose another processing option (we'll discuss what your choices are shortly).

Figure 10 shows the ABAP syntax for pRFC. There are two items to note in the `CALL FUNCTION` statement: First, the usual `DESTINATION` clause has been replaced with a `DESTINATION IN GROUP <group>` clause. Second, a `RESOURCE_FAILURE` exception has been added to the exception list.[12] The purpose of the `<group>` in the `DESTINATION` clause is to identify a group of application servers that the system should use together to process the parallel RFC calls based on availability. The RFC runtime raises the RFC exception, `RESOURCE_FAILURE`, if, at the time you dispatch a call, none of the application servers in the group has enough resources to execute the remote function module.

> ***Note!***
>
> When you design your application, another thing that can constrain performance is your parallel processes, which can require simultaneous access to system resources. For example, two processes might need to modify an entry in a database table, but one may need to wait until the other releases the lock on the entry.

> ***Note!***
>
> pRFC can only be used to execute remote function modules in the same logical system — i.e., on the same system and client.

> ***Tip!***
>
> RFC server groups are maintained in transaction RZ12.

[11] If you're using the tRFC or qRFC variations, the impact is even worse due to the additional queuing, scheduling, and other logic involved.

[12] Note that, because pRFC is an extension of aRFC, the same ABAP additions (like performing and receiving results) for remote session handling are available for the aRFC.

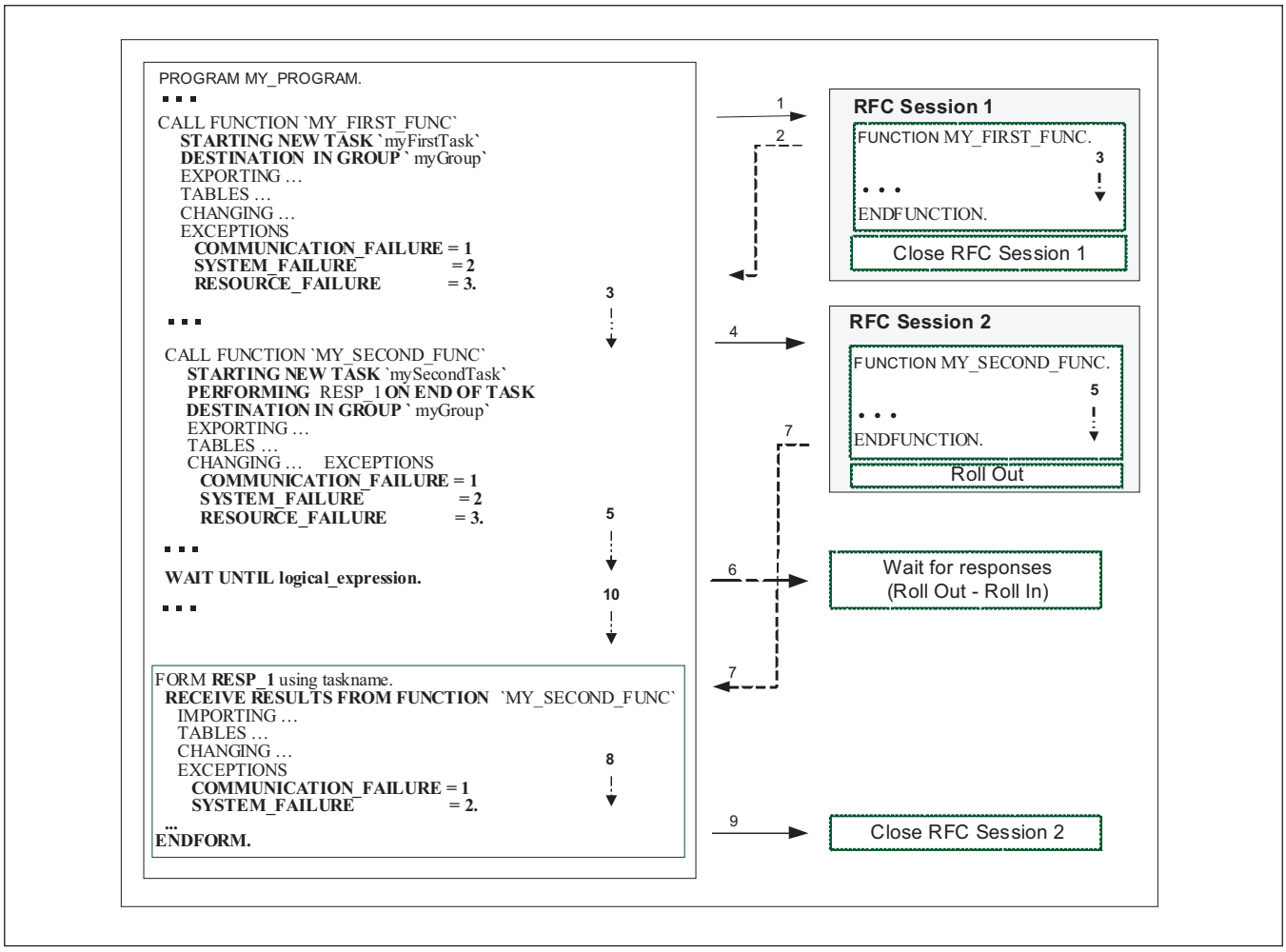


Figure 10 ABAP statements for processing pRFC

At this point, you're probably asking, "What should I do if the call raises the exception `RESOURCE_FAILURE`?" You have a couple of options:

- The best option is to allow the remote system time to free up system resources by pausing your program with a `WAIT` statement. This gives the system time to complete existing aRFCs (e.g., to initiate the receiving phase of outstanding asynchronous responses, if any are being used). In this case, the `WAIT UP TO <seconds>` command should be used.[13]
- Another option — if you are executing the function calls locally — is to call the function modules synchronously instead of in parallel. This should avoid the overload situation because the system can roll your (calling) program out of memory temporarily while it proceeds with each of the RFC calls in turn. When each of the calls complete, the system brings your session back into memory.

In short, pRFC is appropriate when you need to process a large number of RFC calls asynchronously, such as during data loads. The built-in load balancing and overload-avoidance features make it a responsible choice that can help avoid angry emails or knocks at your door because of system slowdowns.

[13] Refer to SAP Note 597583 for more on this topic.

	sRFC	aRFC	tRFC	qRFC	pRFC
Execution time	immediate	immediate	unspecified	unspecified	immediate
Processing type	synchronous	asynchronous	asynchronous and EO*	asynchronous and EOIO*	asynchronous
Output parameters	supported	supported	not supported	not supported	supported
User switch	supported	supported	supported	supported	not supported
Dialog interaction	supported	supported	not supported	not supported	not recommended
Monitoring	not available	not available	available	available	not available
* EO = Exactly Once; EOIO = Exactly Once In Order					

Figure 11 Decision matrix for choosing the "right" RFC type

> ***Caution!***
>
> ABAP statements that may lead to a roll out action — such as `CALL SCREEN`, `CALL TRANSACTION`, sRFC calls, or `WAIT` statements — may trigger a database commit. The use of ABAP statements that lead to a roll out within a callback (response) routine of an aRFC with response call is not permitted. Instead, the processing of those statements is initiated in the caller program after the `WAIT` statement.

In order to fulfill the condition in the `WAIT UNTIL` expression, you need to set some variables in the callback routine.

Choosing the "right" RFC type

Learning about the different RFC types available can be a bit overwhelming at first, but it is a wise and fruitful investment. To help you along, we've assembled a table (**Figure 11**) that summarizes some key characteristics of the types of RFCs to help you choose the type best suited for your situation.

These key characteristics include:

- **Execution time:** With sRFC, aRFC, and pRFC, execution occurs immediately when the remote function is called. In contrast, tRFC calls execute when a `COMMIT WORK` occurs.[14]
- **Processing type:** sRFC is undoubtedly the most frequently used RFC type, and will continue to be, because it is easy to code and debug, and because you can use the results of RFC calls in your main program for the majority of situations. In situations in which you do not need the results (i.e., output parameters) or in which performance is particularly important, you need to decide if one of the other types is a better choice. For example, tRFC and qRFC are appropriate choices when you need EO and EOIO execution of function modules, respectively (assuming you can live with some of their restrictions, which you will learn as you continue to read this section).
- **Output parameters:** You cannot retrieve function module output parameters with either tRFC or

[14] The qRFC Scheduler manages destinations that are registered in transaction SMQS.

qRFC (for reasons that go beyond the scope of this article). aRFC and pRFC provide a callback mechanism that you can use to make the system call a subroutine (`FORM`) in your program to receive the output parameters when the program becomes available (because of a `WAIT` statement, dialog step, or sRFC call, for example), although you have to be careful not to assume the sequence or point in time at which these parameters will return.

- **User switch:** With all of the options except for pRFC, you can explicitly specify a user ID or a client number to use when logging on to the remote system. (Further discussion of the logon process in relation to pRFC goes beyond the scope of this article.)
- **Dialog interaction:** As explained in the sidebar "What happens if an RFM spawns a dialog?" below, unsuppressed dialog interactions[15] within function modules are possible with sRFC and aRFC, but not with tRFC and qRFC. Dialog

[15] That is, the use of statements such as `CALL TRANSACTION` or `CALL SCREEN`

What happens if an RFM spawns a dialog?

Most function modules you call don't have user interfaces — they simply execute, perform some action, and return a result. Occasionally, however, you may encounter a function module that calls one or more transactions and does not suppress the transaction screen (by supplying the system with batch data input values, for example). Sometimes this happens if a transaction needs additional user input to complete its task (e.g., the input of an order number).

The good news is that, in the case of sRFC and aRFC calls, the RFC runtime can display these screens to your users on the local RFC client. In other words, when screen outputs are processed during the execution of a synchronous remote function module call (e.g., by the ABAP statements `CALL TRANSACTION` or `CALL SCREEN`), the generated SAPGUI data is sent back to the RFC client, rather than forcing the function module error out. Pretty neat, right?

Note!

Transactional and queued calls do not pass dialogs through because there can be multiple calls being made at the same time, and the connection to the caller program might not be available at the processing time of LUWs. Although with pRFC a dialog interaction is technically allowed, you must be very careful using it. Dialog interactions are executed in external sessions and are limited to six at a time (the profile parameter rdisp/max_alt_modes controls this number). The system will generate error messages if you try to process any more than six dialog sessions concurrently. Although you may want to use pRFC when processing a large number of current sessions, it is not recommended.

Continues on next page

Continued from previous page

Note!

There is one additional, logical requirement for displaying dialog screens locally — there must be someone to whom the dialogs display. That is, your program must be executed in the foreground via a "regular" SAPGUI user (e.g., a dialog user). If you use a dedicated RFC user account for the target system, the RFC communication user must also be a dialog user. If this condition is not met, the system raises the ABAP runtime error `DYNPRO_SEND_IN_BACKGROUND` when it encounters a screen on the remote system.

The manner in which the dialog screens are displayed differs depending on whether the function modules are executed synchronously or asynchronously. In the case of sRFC, the output of the remote dialog screen is processed inline, on the same screen (i.e., the caller screen is temporary replaced by the remote screen). In contrast, each screen from the asynchronous calls pops up in a new SAPGUI window.

Note!

There are two important issues that you should be aware of when relying on this "remote screen" mechanism:

- The execution of any command field codes in the form of `/n<transaction code>` in a full screen (i.e., a screen with a command field area) leads to interruption of the RFC connection to the client program, with the exception of the aRFC without response. In other words, with /n, in the case of sRFC, the remote session terminates and the caller program continues its processing after the RFC.
- From the security point of view, with the help of a full screen, users can execute other commands, such as `/o<transaction code>` (subject to the RFC user's authorizations, of course). This can be a problem in a scenario in which a static RFC user ID has been hard coded into the RFC destination, and this user ID has authorizations that exceed those appropriate for that user. The user could theoretically access all data and transactions available to the RFC user.

To avoid these remote screen issues, we recommend one of the following alternative approaches:

To avoid the `/n<transaction code>` issue:

- Use pop-up screens (i.e., screens without command fields), instead of full screens so users don't have access to the command field in the screen.*
- Use aRFC without response instead of sRFC, if possible, since the RFC connection to the server is no longer available.

To avoid security issues:

- Suppress remote screens whenever possible by passing the necessary data through the remote function module.
- Solicit the additional necessary information via your own screens, and pass the data to the remote function.
- Provide the RFC user, maintained in the RFC destination, with restricted, well-tailored authorities for his or her allowed activities in the target system.**

Note!

In an RFC session, the list processing outputs, which are generated by ABAP `WRITE` statements, are ignored because the RFC server runtime deactivates list processing. The processing of WRITE statements can be activated indirectly, however, by placing WRITE statements within a report called by the RFM through a `SUBMIT <report> AND RETURN` statement.

* For information on how to control these screens, refer to the ABAP documentation at http://help.sap.com/.

** For more information, go to the SAP NetWeaver '04 documentation at http://help.sap.com/ and navigate to Application Platform → Connectivity → Components of SAP Communication Technology → Classical SAP Technologies (ABAP) → RFC → RFC Programming in ABAP → Maintaining Remote Destinations → Trusted Systems: Trust Relationships Between SAP Systems.

interaction is generally possible with pRFC, but is not recommended. Refer to the sidebar for details behind these restrictions.

- **Monitoring:** Monitor transactions are only available for tRFC and qRFC. tRFC monitoring and debugging are accomplished via transaction SM58. Transaction SMQ1 provides a monitor for the qRFC outbound queues, and transaction SMQ2 provides a monitor for the qRFC inbound queues. There are no monitoring transactions for the sRFC, aRFC, and pRFC connections available, other than the rudimentary transaction SMGW, which shows the present RFC connections on the application server.

Conclusion

This is the first in a two-article series on RFC capabilities within SAP Basis 4.0B or higher systems. This article has taken you on a tour of five key types of

RFC. It has presented some basic, yet key terms and concepts you need to know to work effectively with RFC on your SAP systems. You have also had an opportunity to explore the RFC communication process and to step through each RFC type you can choose from in daily practice. We've found that most people are surprised to learn how rich and powerful this functionality is, and have only ever used one or two pieces before reading this article. You can now begin trying your hand at the more advanced RFC types, yielding better performing and more robust ABAP applications.

In the next article, we explore how to create and secure remote function modules, how RFC serialization/deserialization works, and how to handle exceptions and messages within your ABAP code using best practice techniques.

Masoud Aghadavoodi Jolfaei studied computer science and received his doctorate in the area of satellite communication at Aachen University of Technology. He joined SAP AG in 1994 and became a member of the ABAP Connectivity group, where he works as a development architect on the design, tools, and rollout of the ABAP communication infrastructure. In addition, Masoud is responsible for the integration of Internet protocols (HTTP, HTTPS, and SMTP) into the ABAP runtime. You may reach him at masoud.aghadavoodi.jolfaei@sap.com.

Eduard Neuwirt joined SAP AG in 1999 and became a member of the ABAP Connectivity group, where he worked on the design, tools, and rollout of the ABAP communication infrastructure. Eduard was also responsible for the development of the remote function call (RFC) tools on the external side, including the RFC library and JRFC. Since September 2005, Eduard has worked for the SAP Defense and Public Security Department. He is responsible for the interfaces to external military non-SAP systems. You may reach him at eduard.neuwirt@sap.com.

