

---

# Increase the efficiency of your RFC communications with bgRFC — a scalable and transactional middleware framework

by Wolfgang Baur, Omar-Alexander Al-Hujaj, and Wolfgang Röder



**Wolfgang Baur**  
Developer,  
ABAP Connectivity,  
SAP AG



**Omar-Alexander Al-Hujaj**  
Developer,  
ABAP Connectivity,  
SAP AG



**Wolfgang Röder**  
Development Architect,  
SAP NetWeaver,  
IMS NW Foundation,  
SAP AG

Most ABAP developers have used remote function calls (RFCs) in one form or another to enable SAP and non-SAP system interoperability and process communication. In high-volume processing scenarios, the most commonly used types are asynchronous RFC (aRFC) for parallel function calls, transactional RFC (tRFC) for ensuring that grouped function calls are called once, and queued RFC (qRFC) for ensuring that grouped function calls are called once in a specified order.<sup>1</sup> However, aRFCs, qRFCs, and tRFCs have their limitations. aRFCs are not guaranteed, and while qRFCs and tRFCs follow the “transactional” model in which RFC calls succeed or fail as a group so that all calls are guaranteed, their scalability is limited. To bridge these gaps, with SAP NetWeaver 2004s SP14 SAP introduces background RFC (bgRFC), a new RFC type that improves runtime through efficient, highly scalable, transactional processing of large numbers of sequential function calls.

This article is an introduction to bgRFCs for ABAP Objects developers, and contains basic examples describing SAP NetWeaver integration scenarios based on ABAP Objects. The first section provides a brief introduction to reliable messaging and bgRFC terminology, including units, destinations, scenarios (outbound, inbound, and out-inbound), and Quality of Service (QoS) levels. Then you will learn how to program with bgRFCs, including how to create inbound, outbound, and out-inbound units, how to lock and unlock outbound units, how dependencies between the update task and bgRFC are handled, and how bgRFC units are processed. The article concludes with how to monitor bgRFCs and how to configure the bgRFC scheduler.

<sup>1</sup> For more information on the available types of RFCs, see the *SAP Professional Journal* articles “Master the five remote function call (RFC) types in ABAP: Part 1 — A comprehensive guide for SAP programmers and administrators” (September/October 2006) and “Explore the technical procedures and settings for creating and securing RFMs in ABAP: Part 2 — A comprehensive guide for SAP programmers and administrators” (November/December 2006).

*(Full bios appear on page 106.)*

Although experience in qRFC, tRFC, and aRFC is not mandatory for reading this article or for using bgRFC, it certainly helps in understanding the benefits of bgRFC. Knowledge of distributed computing is also very helpful in understanding the terminology.

## Key bgRFC concepts

There are some important foundational concepts you need to understand so that you can effectively use bgRFC in your programs — bgRFC units, destinations, scenarios, and Quality of Service levels. We'll look at these key concepts in detail in the following sections.

### bgRFC units

bgRFC supports two types of units — units of type T (transactional) and of type Q (queued). Both units follow a transactional model in which multiple RFC calls succeed or fail as a group, only units of type Q have additional features. We will come back to the details of both types of units in an upcoming section on bgRFC-supported Quality of Service levels, but first let's take a closer look at the definition of a *transaction* in the context of bgRFC units.

Taking a look on the Internet, we find the following definition for transaction: “When most database programmers refer to a transaction, they are referring to an ACID transaction. These transactions (logical units of work) are designed for short-lived decisions usually lasting under a few seconds. These transactions identify a logical unit of work that is performed either completely or not at all. That is, either a COMMIT WORK or a ROLLBACK WORK is performed on the operations. This enables the data to maintain a state of consistency.”<sup>2</sup>

ACID is an acronym:

- **A**tom**i**c: The transaction cannot be divided into parts and must be executed completely (or not at all).

- **C**onsistency: A consistent state of the data may be expected at all times; either all of the RFCs in the unit are processed or none.
- **I**solation: Transactions are isolated from each other; transaction serialization ensures that multiple transaction instances do not collide.
- **D**urable: Once a transaction commits, its updates survive, even if the system goes down.

This definition is related to database transactions, but with slight changes, you can reuse it for bgRFC units: “When ABAP Objects programmers refer to a bgRFC unit, they are referring to an ACID transaction. These transactions identify a logical unit of work that is performed either completely or not at all. That is, either a COMMIT WORK or a ROLLBACK WORK is performed on the operations. This enables the data to maintain a state of consistency.”

In a transactional system, you want to save some sets of work in their entirety or not save them at all. For example, you don't want the line items of an order to be successfully saved if the save of the header row containing the order total information fails. This complete set of work is commonly called a “logical unit of work.” Every time we use the term “unit” in this article, it is used as a synonym for “logical unit of work.”

Like qRFCs and tRFCs, a bgRFC unit is a container for a number of RFCs. What's different, however, and completely new in bgRFC is the possibility to create as many units in parallel as you like and to handle them separately. This provides a big advantage over the former tRFC and qRFC concept in which you could only handle one unit at a time (CALL FUNCTION . . . AS SEPARATE UNIT). This new feature is very helpful if your business transaction requires the creation of multiple independent messages for successive business steps (e.g., for accounting or inventory systems). With bgRFC, you are able to commit all of them or roll back all of them in a single transaction. This helps you keep your business system consistent.

The potential of this feature is that you can now easily group RFCs, and all such units are independent. What happens if one RFC in one unit fails? A bgRFC unit is atomic (ACID); either all of the RFCs in the unit are processed or none. Each bgRFC unit is

<sup>2</sup> See [http://www.serviceoriented.org/acid\\_transactions.html](http://www.serviceoriented.org/acid_transactions.html).

isolated. This means that all bgRFC units are executed independently. At the end of the execution of each bgRFC unit, there is an implicit COMMIT WORK, which makes the data manipulation persistent in your database.

## bgRFC destinations

One important piece of information is missing to complete the full picture of the potential of bgRFC: Similar to tRFC and qRFC, with bgRFC function modules are called remotely. (In a later section you will learn that a local system also can be used as a remote system.) For RFCs, we use the construct of a *destination* to define where to execute the function module. In the case of bgRFC, tRFC, and qRFC, the destination defines where the unit will be executed. Later, you will learn that bgRFC destinations are handled differently depending on whether they are outbound or inbound, and we're sure that you'll like the new way of handling destinations in your programs.

SAP's ABAP architecture is a three-tiered architecture with different types of work processes. All application servers with dialog work processes can be used as a destination for RFCs, whether the server is part of the same installation or a different installation. bgRFC is primarily designed for interoperability, but you can also use it for an asynchronous and load-balanced mass execution. A destination is usually not a single server, but it could be.

Let's take a closer look at the load balancing issue. bgRFC units are always executed in dialog work processes. Because dialog work processes are primarily used to handle the work processes of human users, what will happen if dialog users have to share the same dialog work processes with masses of bgRFC units? What will be the effect on the response times for the human users? What mechanisms prevent the overloading of servers and the use of idle servers? Not to worry — bgRFC provides functions for a fair share of system resources to automatically handle load balancing issues.

A bgRFC destination could also be a group of application servers — the automatic dispatching of

units to these servers is handled by the bgRFC runtime. Later in this article, you will learn how to define bgRFC destinations and server groups.

## bgRFC scenarios

You can use every application server with at least one dialog work process as a destination for bgRFC units. bgRFC supports three types of bgRFC scenarios — outbound, inbound, and out-inbound — which we will introduce in the subsequent sections. The main difference between these scenarios is which system controls the execution of units — i.e., in which system the bgRFC scheduler, which is in charge of dispatching all bgRFC units, is running (we will go into more detail on this later in the article). bgRFC scenarios are strongly related to the bgRFC destination. Every time you request a destination object, you use special factory methods<sup>3</sup> that are related to the three bgRFC scenarios. We introduce the complete bgRFC API in its own section, but let's first zero in on the three scenarios.

### bgRFC outbound

Let's begin with the outbound scenario. Outbound scenarios are scenarios of RFCs between different installations, e.g., between SAP ERP and SAP Supply Chain Management (SCM) systems. In this scenario, one application creates information that must be synchronized with at least one other remote application. Because of the possibility that the remote system may not be up and running, the units are temporarily stored in the local database of the sender system prior to delivering them to the bgRFC scheduler, which is part of the sender system. In the outbound scenario, the bgRFC scheduler of the sender system is in charge of the execution of the units in the receiver systems. This leads to the situation in which the execution of the units is separated (asynchronous in a separate transaction) from the transaction that has created the units in the sender system. After a unit has been executed in the remote system, the bgRFC scheduler deletes this unit in the local database. This concept provides the service quality level Quality of Service Exactly Once

<sup>3</sup> An object-oriented design pattern that deals with creating objects without specifying the exact class of the object to be created.

## Quality of Service (QoS)

In computer networking, the traffic engineering term Quality of Service (QoS) refers to control mechanisms that can provide different priorities to different users or data flows, or guarantee a certain level of performance for a data flow in accordance with requests from the application program. QoS can be defined as “the measure of the degree of satisfaction of the user of the system.”\* This term is sometimes used as a quality measure with many alternative definitions, rather than referring to the control mechanisms. In computer networking, a good QoS may mean advanced QoS mechanisms, or high probability that the network is able to provide the requested level of performance. In the context of bgRFC, QoS is used to define the level of service for the dispatching of bgRFC units.

QoS types include the following, where “messages” can be considered a synonym for bgRFC units:

- **Quality of Service Best Effort (QoS BE):** If a system failure occurs, QoS BE messages may not be delivered. There is no guarantee for the right message order.
- **Quality of Service Exactly Once (QoS EO):** QoS EO messages are guaranteed to be delivered. There is no guarantee for the right message order.
- **Quality of Service Exactly Once In Order (QoS EOIO):** QoS EOIO message delivery and sequence are guaranteed. The information is published in time order, and the same order should be guaranteed at the receiver’s side (i.e., old messages should arrive at the receiver’s side before new ones).

\* See <http://en.wikipedia.org/wiki/Qos>.

(QoS EO). Quality of Service Exactly Once In Order (QoS EOIO) is also available. (We go into more detail on bgRFC QoS in an upcoming section; see the sidebar on this page for a brief introduction to this concept.)

In **Figure 1**, you see the global class CL\_BGRFC\_DESTINATION\_OUTBOUND, which provides two factory methods to get a reference to an object with the interface IF\_BGRFC\_DESTINATION\_OUTBOUND.

In **Figure 2**, we introduce the factory methods used to create instances of bgRFC unit objects. We will present more detail on these unit objects when we introduce the supported QoS levels in an upcoming section. You can see that there is a hierarchical relationship between bgRFC destinations and bgRFC units, and that there is a strong binding between bgRFC destinations and bgRFC scenarios.

### *bgRFC inbound*

For the inbound scenario, in contrast to the outbound scenario, the receiver system is identical to the sender system. This scenario is similar to the update task. The bgRFC inbound scenario is not a replacement for the update task, but provides new capabilities for developers.

If you call a function module using the statement `CALL FUNCTION ... IN UPDATE TASK`, the function module is flagged for execution using a special update work process. The actual execution of the function module may therefore be postponed. When you call a function module using the `IN UPDATE TASK` addition, the function module and its interface parameters are stored as a log entry in a special database table. You can then commit or roll back all of the update postings of a program at one time. The implication, therefore, is that you can only have one logical unit of work at a

<b>CL_BGRFC_DESTINATION_OUTBOUND</b>
+CREATE(in DEST_NAME : BGRFC_DEST_NAME_OUTBOUND) : IF_BGRFC_DESTINATION_OUTBOUND +CREATE_GROUP(in DEST_NAMES : BGRFC_DEST_NAME_TAB_OUTBOUND) : IF_BGRFC_DESTINATION_OUTBOUND

**Figure 1** Factory methods for outbound destinations

«interface» <b>IF_BGRFC_DESTINATION_OUTBOUND</b>
+CREATE_TRFC_UNIT() : IF_TRFC_UNIT_OUTBOUND +CREATE_QRFC_UNIT() : IF_QRFC_UNIT_OUTBOUND +CREATE_QRFC_UNIT_OUTBOUND() : IF_QRFC_UNIT_OUTBOUND

**Figure 2** Factory methods to create instances of unit objects for the outbound scenario

<b>CL_BGRFC_DESTINATION_INBOUND</b>
+CREATE(in DEST_NAME : BGRFC_DEST_NAME_INBOUND) : IF_BGRFC_DESTINATION_INBOUND

**Figure 3** Factory method to create instances of inbound destinations

time. Every time you want to create a new and independent logical unit of work you have to close the existing one by using `COMMIT WORK`. This has a major impact on the design of your program in those cases in which you want to create multiple independent logical units of work. As you learned before, you can create and manage many bgRFC units in parallel. You can create additional units as long as you have not called `COMMIT WORK`. `COMMIT WORK` closes all existing bgRFC units and the application transaction. It stores the interface parameters of all function modules in bgRFC units plus the interface parameters of all function modules for the update task.

In contrast to update tasks, the system processes bgRFC units in dialog work processes. This provides you with more control over the dispatching of units. For example, you can define a special bgRFC destination for a specific purpose and use it for your units. You can use this for parallelization of the mass execution of units or to make use of special servers for a purpose or service (e.g., a tax calculation). You can also separate types of units by using different bgRFC destinations for better performance or throughput. Think about the usage of buffers, context switches, and so on. For example, you can use each bgRFC

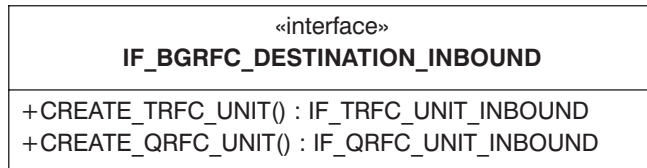
destination as a logical application to separate financial or HR transactions from others. The bgRFC inbound scenario is always an alternative to the update task, if you require more control over the execution of units. However, bgRFC units do not keep the current business transaction for used resources (i.e., master data records) locked. This could be one reason for staying with the update task. This does not mean that during the execution of the unit there is no locking mechanism, but each RFC is responsible for its own locking to guarantee system integrity.

Like the outbound scenario, the inbound scenario provides at least the service quality level QoS EO. This could be a reason for using bgRFC instead of aRFC, because aRFC provides only Quality of Service Best Effort (QoS BE), which means you don't have reliable messaging. Reliable messaging is a protocol that allows messages to be delivered reliably between system applications. This means that the delivery of messages is guaranteed based on a hand-shaking protocol between sender and receiver systems.

The factory method in **Figure 3** provides an instance of a destination object with the interface `IF_BGRFC_DESTINATION_INBOUND`.



As in the outbound scenario, bgRFC provides factory methods to create instances of unit classes, as shown in **Figure 4**.



**Figure 4** Factory methods to create instances of unit objects for the inbound scenario

### *Note!*

You should use the concept of the update task as long as it provides all required functionality and only switch to the bgRFC inbound scenario if it is really required. Do not misuse dialog work processes for asynchronous transactional updates.

### **bgRFC out-inbound**

The out-inbound scenario is a mixture of the outbound and the inbound scenario. The bgRFC scheduler in the sender system is only allowed to transfer the units from its outbound workload to the inbound workload of the receiver system. The units will be deleted in the workload of the sender system after the transfer. In the receiver system, the local bgRFC scheduler works as in the inbound scenario to execute the units. The main purpose for this scenario is so you don't give the sender system control over the execution of the units in the receiver system. Another purpose is that the working process of the sender system in an outbound scenario is blocked in idle mode during the execution time of the unit in the receiver system. In an out-inbound scenario, the sender system has less idle time. From the programmer's perspective, the out-inbound scenario is similar to the outbound scenario (refer

back to **Figure 2**). For this reason, bgRFC doesn't provide separate factory methods to create instances of destinations, but rather provides a factory class for creating instances of unit classes for the out-inbound scenario.

### **bgRFC Quality of Service levels**

In the prior sections, we used the term Quality of Service. This term is often used in the area of interoperability and networking, including Internet and Web services, to define levels of provided services (see also the sidebar on page 80). With bgRFC, we have to distinguish between the Best Effort (QoS BE), Exactly Once (QoS EO), and Exactly Once In Order (QoS EOIO) service levels. Even though QoS BE is not supported by bgRFC, you should be aware of the differences between it and other QoS levels.

QoS BE means that the service provider does not guarantee the delivery of the message because the service expends minimal (best) effort for the service. This means that the message could be transferred to the receiver once, multiple times, or never. In the case of aRFC, you get a service that is comparable with QoS BE (except the multiple delivery times), but this may not be enough for your business transactions. (Of course, if you do not need reliable messaging, you should think about using aRFC instead of bgRFC to save the additional effort of reliable messaging.)

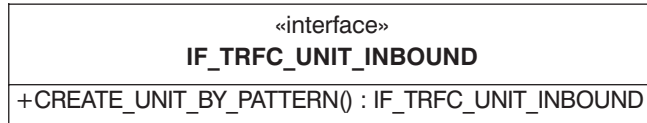
For reliable messaging, you need QoS EO and QoS EOIO. Let's take a look at how bgRFC supports these two service levels via bgRFC units of type T (transactional) and Q (queued), respectively.

### **bgRFC QoS EO**

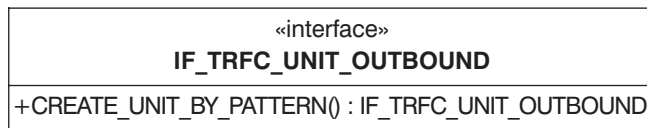
bgRFC units of type T (transactional) offer QoS EO. QoS EO means that the service guarantees that the message will be delivered once and only once. In the case of bgRFC, this means that the unit will be executed in the receiver system once. The unit can't get lost because a bgRFC unit is persistent, and the bgRFC protocol guarantees that the unit will be executed once. All bgRFC units with the attribute QoS EO are

independent from each other. All of them can be executed in parallel in a random order — which describes the use case of bgRFC QoS EO scenarios.

QoS EO is supported for both bgRFC inbound (see **Figure 5**) and outbound (see **Figure 6**) scenarios. You can identify the API for the QoS EO units by the term “tRFC” in the names of classes, interfaces, and methods.



**Figure 5** Interface of a bgRFC unit for the inbound scenario



**Figure 6** Interface of a bgRFC unit for the outbound scenario

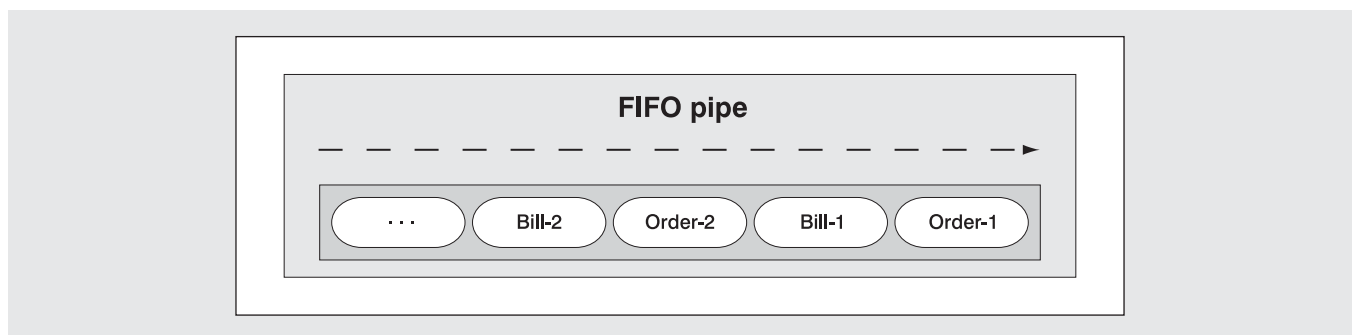
### bgRFC QoS EOIO

bgRFC units of type Q (queued) offer QoS EOIO — they are like units of type T, which are executed only once, but units of type Q are executed in a particular order.

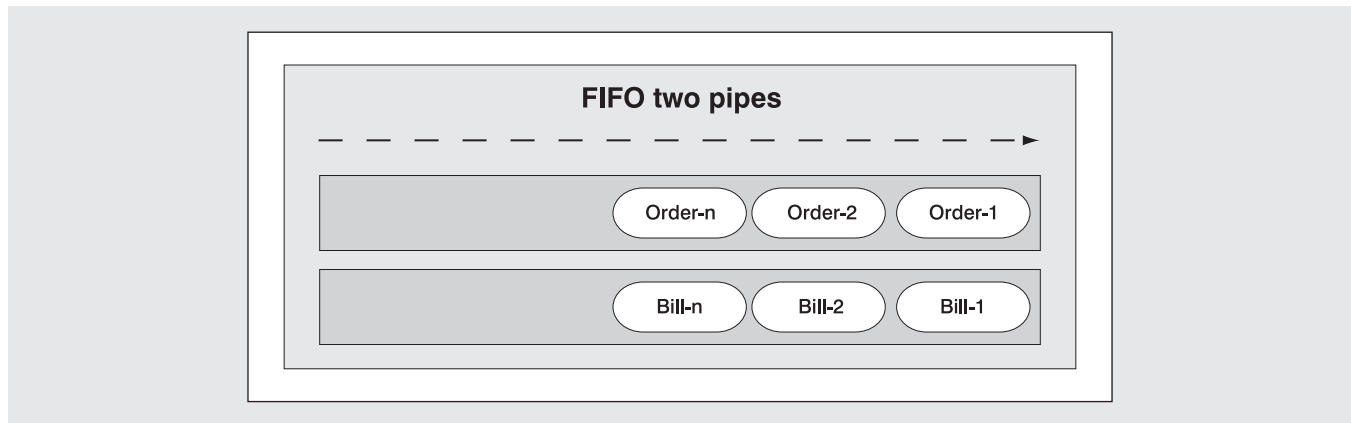
In this section we present the real power of the bgRFC concept. To help you identify scenarios in

which you can use bgRFC with QoS EOIO, we start with some examples that use non-bgRFC solutions and examine their disadvantages:

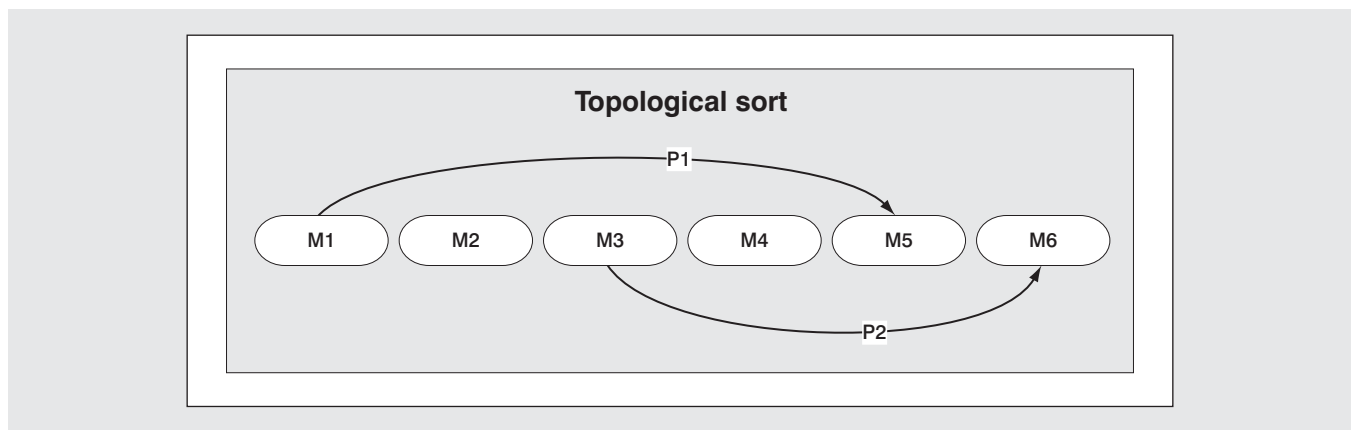
- **Example 1:** A given CRM application is designed for creating sales orders and billing documents. Your business process requires that both documents be forwarded to a back-end system. The back-end system requires that the sales order be imported before the related billing document. The easiest way to handle this requirement is to serialize the messages using a single queue, because it can be set up to work in a FIFO (first in, first out) mode, as shown in **Figure 7**. That’s fine for small solutions, but what happens in a high-volume system? The single queue can become a bottleneck because there is no option for scalability via parallelization. In that case, there is no option to add more processing power (e.g., by adding application servers) to handle a high volume of business transactions.
- **Example 2:** Let’s try to attack the problem from a different angle. Suppose we were to increase the number of queues to 10. We’d hope that such a change would result in running 10 times faster, but it doesn’t. There is one major issue: how to dispatch the messages to the queues. Round-robin dispatching doesn’t help, because we have to keep the dependencies between the sales orders and the billing documents in mind, and we then have to guarantee that the corresponding billing document is assigned to the same queue as the sales document. A simple heuristic could help (e.g., adopting a dispatch sequence based on the last digit of the



**Figure 7** Example 1 with a single pipe



**Figure 8** Example 3 with two pipes



**Figure 9** Basic topological sort

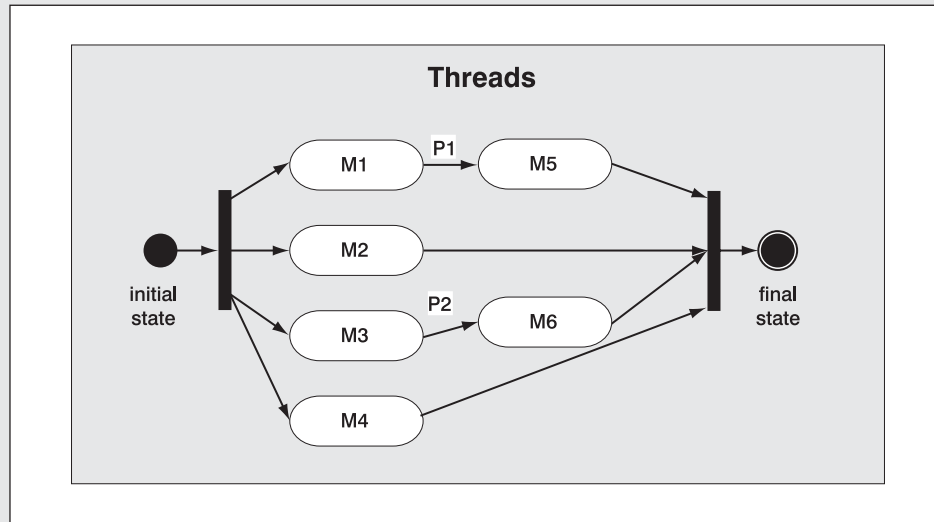
sales document number), but such simple solutions are generally not sufficient in real-world business scenarios, because the number of related documents may be high and the heuristic rapidly becomes too complex.

- **Example 3:** If 10 queues turn out to be unmanageable, maybe we could use two queues, one for sales orders and one for billing documents (see **Figure 8**). The sales order queue has a higher priority. Every time at least one message is in the sales order queue, the scheduler stops working on the billing document queue. That could work, but if you have a high volume of sales orders, you would have to have a good explanation for your customer waiting at the front desk for the billing document. Also, what would happen if the first

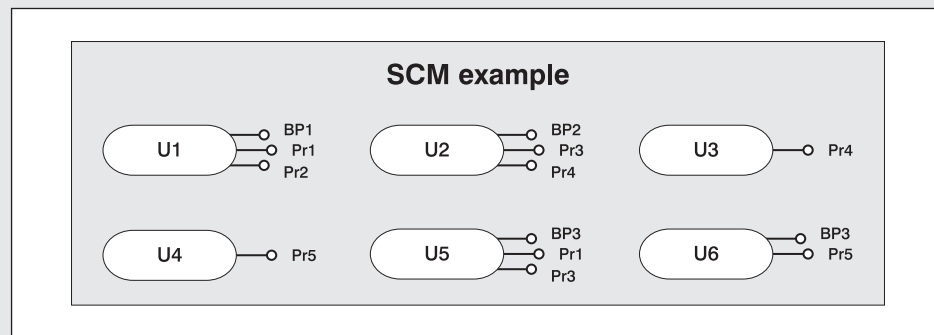
message in a queue fails? For better scalability, you could increase the number of queues, but this is not a realistic solution; daily business is not as simple as this example.

We could continue thrashing around devising clever queuing schemes to address bottlenecking issues, but bgRFC offers a simpler solution. To understand the bgRFC approach, it's helpful to look at the problem using a *topological sorting* scenario. You may not be familiar with this term, but as a developer of asynchronous high-volume business transactions, you certainly know the problem. Here's an example: Consider the interface between a CRM and an SCM system. We now enter the world of dependencies and "pipes" (also known as "queues"). In **Figure 9** you see the six messages M1 to M6.





**Figure 10** Resulting threads with the messages of Figure 9



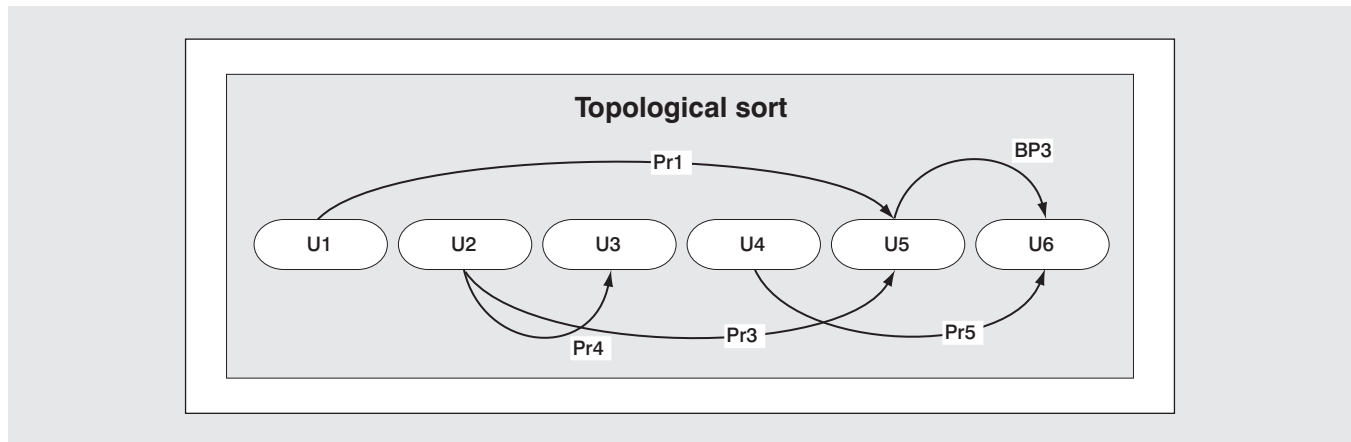
**Figure 11** bgRFC units and used business objects

We want to execute all of these messages in parallel, but we have two dependencies: (1) message M5 is related to M1 (marked as P1) and M6 is related to M3 (marked as P2). In **Figure 10** you can see the corresponding threads to the result of the topological sort with all of the messages that can be processed simultaneously or sequentially.

What we need is an engine that automatically recognizes the dependencies between messages and executes as many messages as possible in parallel but in the correct order. Here, we face the QoS

EOIO requirement. Message M5 can be executed only if message M1 is already executed. The same is true for the messages M6 and M3. bgRFC provides the solution we need for this.

Let's use a more sophisticated example of a generic SCM scenario to illustrate how bgRFC meets the QoS EOIO requirement. Take a look at the **Figure 11**. Note that we have to switch the terminology used in the previous example — in bgRFC we don't work with "messages" but rather with "bgRFC units."



**Figure 12** SCM example topological sort

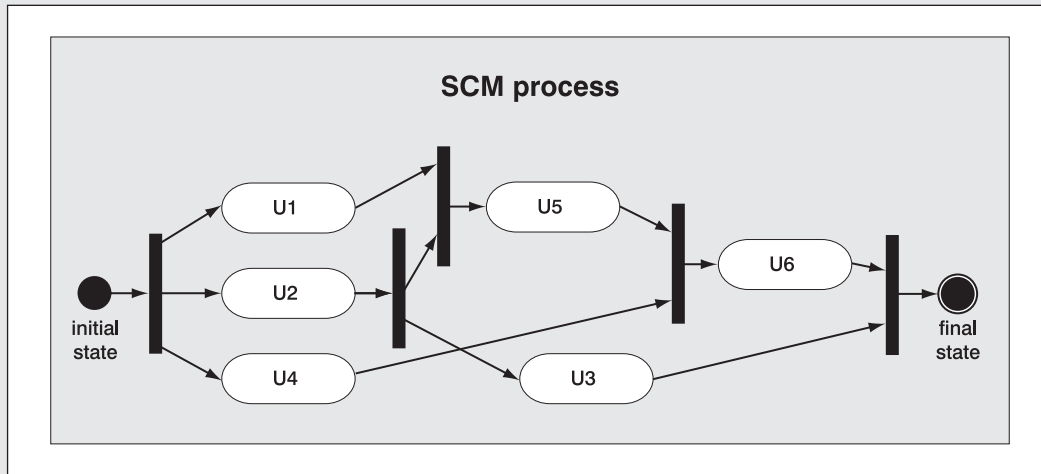
In this example, we have a multi-user application that creates bgRFC units for an SCM destination. A unit is a container for many RFCs, and each of these function calls is a manipulation of business objects. For example, the unit U1 manipulates business partner BP1 and the two products Pr1 and Pr2. The following three units U2 to U4 are manipulating different business objects. This means that there is no dependency between unit U1 and the following three units. The unit U5 is using the business objects BP3, Pr1, and Pr3. Therefore a serialization of the units U1 and U5 is required because both are using or manipulating the product Pr1. Without a serialization, the business process could fail, and the database consistency is in danger. In this example, there are also dependencies between the other units.

**Figure 12** visualizes the dependencies of all units. Each edge of the topological sort represents the reason for the dependency. You can determine the required business process as a result of a topological sort of the units.

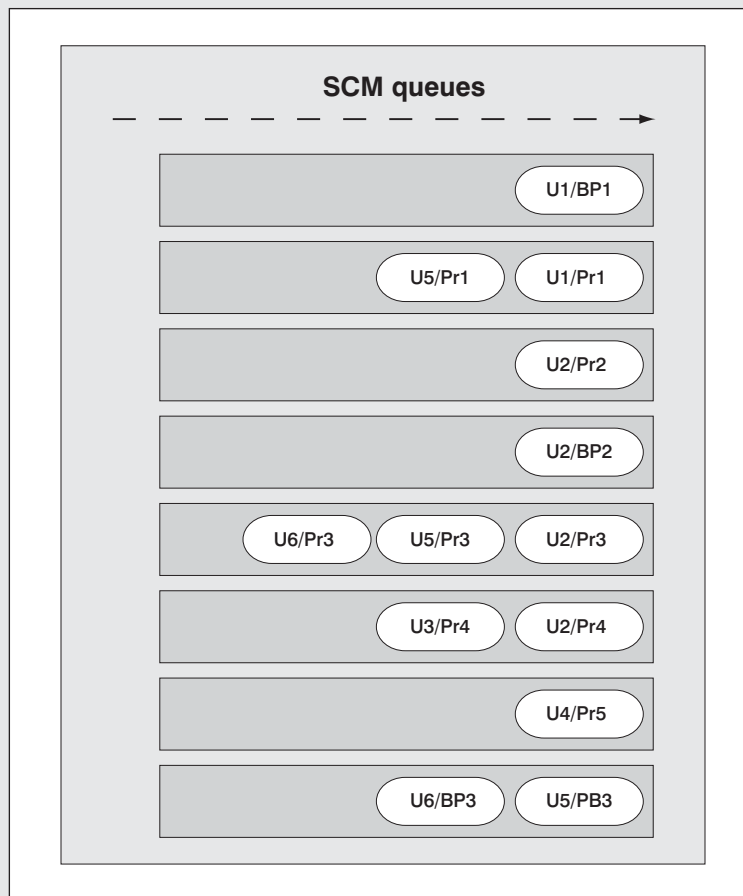
Again, to satisfy the requirements of this scenario, you need an engine that recognizes all dependencies between all units and executes as many units in parallel as possible. A number of predefined queues is not helpful because an SCM scenario is usually a high-volume scenario and our example is only a small fragment. In reality we will have thousands of units that could be processed in parallel, providing that you

have the right engine for the dependency recognition. This engine must be very fast and effective; otherwise, the engine itself can become the limiting factor for high-volume scenarios such as SCM. With the assumption that a unit can only be assigned to one queue, we will lead with our example in only one queue, because we have indirect dependencies between all units. These indirect dependencies are visualized as the forks and joins (i.e., the black vertical bars) in **Figure 13**.

What we need is a solution to assign units to multiple queues. However, a bgRFC unit is atomic, and we are not allowed to split bgRFC units. The solution is to assign units to *virtual queues*, which are identified by name and destination. The reason for the identification by destination is that there cannot be a dependency between bgRFC units for different destinations. Remember that a destination is a pointer to a target system for bgRFC units. Every time you need an additional virtual queue, you can use it — simply by using the appropriate name. The main challenge here is to define the rules for the names of the virtual queues. All applications or transactions that require a serialization of bgRFC have to share the same bgRFC destination and queue names. The dependency recognition is automatically handled by the bgRFC API. The bgRFC runtime analyzes the names of the virtual queues of each unit and automatically inserts the units in the bgRFC process for a destination (e.g., U1, U2, U3) as shown in **Figure 13**. **Figure 14** shows how the



**Figure 13** SCM example business process using standard notation



**Figure 14** Usage of virtual queues

«interface» <b>IF_QRFC_UNIT_INBOUND</b>
+CREATE_UNIT_BY_PATTERN() : IF_QRFC_UNIT_INBOUND +ADD_QUEUE_NAME_INBOUND(in QUEUE_NAME : QRFC_QUEUE_NAME, in IGNORE_DUPLICATES : bool = false) +ADD_QUEUE_NAMES_INBOUND(in QUEUE_NAMES : QRFC_QUEUE_NAME_TAB, in IGNORE_DUPLICATES : bool = false) +GET_QUEUE_NAMES() : QRFC_QUEUE_NAME_TAB

**Figure 15** Interface of virtual queued units for the inbound scenario

«interface» <b>IF_QRFC_UNIT_OUTBOUND</b>
+CREATE_UNIT_BY_PATTERN() : IF_QRFC_UNIT_OUTBOUND +ADD_QUEUE_NAME_OUTBOUND(in QUEUE_NAME : QRFC_QUEUE_NAME, in IGNORE_DUPLICATES : bool = false) +ADD_QUEUE_NAMES_OUTBOUND(in QUEUE_NAMES : QRFC_QUEUE_NAME_TAB, in IGNORE_DUPLICATES : bool = false) +GET_QUEUE_NAMES() : QRFC_QUEUE_NAME_TAB

**Figure 16** Interface of virtual queued units for the outbound scenario

"interface" <b>IF_QRFC_UNIT_OUTINBOUND</b>
+CREATE_UNIT_BY_PATTERN() : IF_QRFC_UNIT_OUTINBOUND +ADD_QUEUE_NAME_OUTBOUND(in QUEUE_NAME : QRFC_QUEUE_NAME, in IGNORE_DUPLICATES : bool = false) +ADD_QUEUE_NAMES_OUTBOUND(in QUEUE_NAMES : QRFC_QUEUE_NAME_TAB, in IGNORE_DUPLICATES : bool = false) +ADD_QUEUE_NAME_INBOUND(in QUEUE_NAME : QRFC_QUEUE_NAME, in IGNORE_DUPLICATES : bool = false) +ADD_QUEUE_NAMES_INBOUND(in QUEUE_NAMES : QRFC_QUEUE_NAME_TAB, in IGNORE_DUPLICATES : bool = false) +GET_QUEUE_NAMES_OUTBOUND() : QRFC_QUEUE_NAME_TAB +GET_QUEUE_NAMES_INBOUND() : QRFC_QUEUE_NAME_TAB +LOCK_AT_INBOUND()

**Figure 17** Interface of the virtual queued units for the out-inbound scenario

units are distributed across virtual queues by using the business partner key and the product key as the virtual queue name (e.g., U1/BP1, U5/Pr1, U1/Pr1, etc.).

**Figure 15** and **Figure 16** provide the interfaces for the virtual queued units for the inbound scenario and the outbound scenario, respectively. **Figure 17** provides the interface for the virtual queued units for the out-inbound scenario.

You can identify the classes, interfaces, and methods of the APIs of the units with QoS EOIO by the term “qRFC” in the name. For each bgRFC scenario, we provide a single interface. Using this interface you can add queue names to bgRFC units. Because you get the reference to an instance of a unit class via a factory method of a destination class,

there is no need for the developer to include the destination in the queue name. bgRFC handles this automatically.

Here, we used basic examples to describe the problems addressed by the bgRFC solution. In reality, a high-volume business scenario like the SAP SCM example could easily create millions of units with hundreds of queue names per unit and many overlaps between virtual queues. This would lead to an enormous business process graph requiring as many simultaneous threads as possible for maximum throughput of unit executions combined with reliable messaging. bgRFC is a scalable solution that fulfills all of these requirements. Next, we show you how to use bgRFC in your own developed solutions.

```

data:
  l_lock_id type bgrfc_lock_id,

  l_dest      type ref to if_bgrfc_destination_inbound,
  l_unit      type ref to if_trfc_unit_inbound.

* Create the destination object references
  l_dest = cl_bgrfc_destination_inbound=>create( 'TEST_BGRFC' ).

* Create new bgRFC inbound unit type T object reference
  l_unit = l_dest->create_trfc_unit( ).

* Register first function module
  call function 'RFC_FUNCTION_1' in background unit l_unit.
* Register second function module
  call function 'RFC_FUNCTION_2' in background unit l_unit.

* Lock unit if requested
  l_lock_id = l_unit->lock( ).

* Save the call
  commit work.

```

**Figure 18** Create inbound unit of type T

## Programming with bgRFCs

So how do you program using bgRFCs? You will see that the API has a clear object-oriented design and that it is easy to program. This API is designed to support the different scenarios we just described.

### Creating inbound units

The first scenario is the QoS EO — also called *transactional bgRFC*. We generate an independent unit that can be executed in parallel with other units.

In the example shown in **Figure 18**, we generate a bgRFC unit of type T, which contains two function modules. First, you must create an object reference for a destination. Use transaction SBGRFCMAINIDST to configure the inbound destination. A destination handle can be created only if the inbound destination is maintained in the table — otherwise, the system raises an

exception of type CX\_BGRFC\_INVALID\_DESTINATION. We'll describe how to maintain inbound destinations and their meaning in more detail in the section “Inbound destinations” later in the article.

In the second step, we generate an object reference for a specific unit. With this object reference, it is possible to call as many function modules as you want. Each function module that is called with the same unit reference lies within the same logical unit of work. All of the function modules taken together form the bgRFC unit. Within the application program, you can create as many destinations and unit references as you want in parallel. If you want to bundle function modules from within different methods, function modules, or form routines, you have to take over the object reference. For powerful and mass data-relevant applications, it is necessary to build appropriate packages. The creation of a high number of short-running bgRFC units increases the overhead of the



```

data: l_lock_id TYPE bgrfc_lock_id.

* Release the lock of a transactional inbound unit.
cl_trfc_lock_inbound=>if_bgrfc_lock~release( lock_id = l_lock_id ).

commit work.

```

**Figure 19** Release a lock on a unit of type T in inbound scenario

```

data:
  l_lock_id type bgrfc_lock_id.
  l_dest      type ref to if_bgrfc_destination_inbound,
  l_unit      type ref to if_qrfc_unit_inbound.

start-of-selection.
* Create the destination object references
l_dest = cl_bgrfc_destination_inbound=>create( 'TEST_BGRFC' ).

* Create new queued bgRFC inbound unit type Q object reference
. l_unit = l_dest->create_qrfc_unit( ).

* Assign the unit to a queue
l_unit->add_queue_name_inbound( 'BASIS_BGRFC_IN_Q1' ).

* Register first function module
call function 'RFC_FUNCTION_1' in background unit l_unit.
* Register second function module
call function 'RFC_FUNCTION_2' in background unit l_unit.

* Lock unit
l_lock_id = l_unit->lock( ).

* Save the call
commit work.

```

**Figure 20** Create an inbound unit of type Q

bgRFC runtime environment and therefore probably decreases the overall performance.

But what is the meaning of the statement `l_unit->lock( )`? If it is necessary to delay the execution of a unit, the programmer can create a lock within the

application program. In this case, the bgRFC scheduler will not touch the unit until it is unlocked. Two use cases exist for this task. The first one is that you write a test program and you want to see the unit before the scheduler executes it. In this case, you can unlock the unit from the monitor (SBGRFCMON2). After you

**Note!**

All unit references that are generated in the program will be invalidated after the commit, so it is not possible to reuse a unit handle following a commit work.

release the lock, the scheduler takes the unit as soon as possible to execute it. It is neither necessary nor possible to explicitly trigger the scheduler — it is done automatically. The second use case is the prevention of the immediate execution of a unit because the program logic demands this. In this case, our API allows the unlocking of the unit within the application program. To do this, you must store the `lock_id` (in this example, `l_lock_id`) to unlock the unit at a later time. For releasing a lock, the API provides the interface method `if_bgRFC_lock~release( )`. This use case makes sense when there are dependencies in other parts of the application program that are inspected at a later point. **Figure 19** shows how to unlock a transactional inbound unit. The interface is implemented in four different classes that incorporate the two possible scenarios (inbound and outbound) as well as the service type (transactional as well as queued) of the bgRFC.

In the second example program, depicted in **Figure 20**, we create a bgRFC unit of type Q in the inbound queue. There are only a few differences in the coding between this and the previous example — the main issue is that for queued bgRFC, we need a queue name and must use a different factory method to create the unit object. To program a unit of type Q in the inbound queue, you have to proceed as mentioned before. First, create an object reference for a destination object, and then create a unit reference. To provide the unit with a queue name, you have to assign the queue name using the method `add_queue_name_inbound`. It is not important whether you assign the queue name before or after the function module call. After that step, you have a complete bgRFC unit object in which you can bundle as many function modules as you want. To store the data in the database table, you have to call a `commit work`. If you do

not call `commit work`, a rollback happens and no data is saved. After the `commit work`, the bgRFC scheduler can pick the units from the database to execute the units locally.

## Creating outbound units

In **Figure 21** on the next page we create a bgRFC unit of type T in the outbound scenario. As you can see, the call structure is the same as **Figure 18** — only the interface names are slightly different. So you have only to replace the word inbound by outbound within the interface names and you can easily change a program from inbound to outbound bgRFC.

**Note!**

The inbound and outbound scenarios are very different use cases. Also, the term “destination” has a different meaning for the outbound and inbound scenarios. An outbound bgRFC destination must be maintained using transaction SM59. More details how to customize the destination are given in the upcoming section on configuring bgRFC destinations.

## Unlocking a locked outbound unit

You can release a lock within the monitor or with a separate program. To release it with a program, you have to store the `lock_id` at creation time. **Figure 22** on the next page depicts how this is done.

## Creating a bgRFC outbound unit of type Q

In the example program depicted in **Figure 23** on page 93, we create a bgRFC unit of type Q in the outbound queue. There are only a few differences in the coding compared to the bgRFC outbound unit of type T (see **Figure 21**). The main issue is that for queued bgRFC, we need a queue name. Also, you use a differ-

```

data:
  l_lock_id  type bgrfc_lock_id.
  l_dest     type ref to if_bgrfc_destination_outbound,
  l_unit     type ref to if_trfc_unit_outbound.

start-of-selection.
* Create the destination object references
  l_dest = cl_bgrfc_destination_outbound=>create( 'TEST_BGRFC' ).

* Create new bgRFC outbound unit type T object reference
  l_unit = l_dest->create_trfc_unit( ).

* Register first function module
  call function 'RFC_FUNCTION_1' in background unit l_unit.
* Register second function module
  call function 'RFC_FUNCTION_2' in background unit l_unit.

* Lock unit if requested
  l_lock_id = l_unit->lock( ).

commit work.

```

**Figure 21** Create outbound unit type T

```

data: l_lock_id TYPE bgrfc_lock_id.

* Release the lock of a outbound unit.
  cl_qrfc_lock_outbound=>if_bgrfc_lock~release( lock_id = l_lock_id ).

commit work.

```

**Figure 22** Release an outbound lock

ent factory method to create the unit object. To program a unit of type Q in the outbound queue, you have to proceed as mentioned before. First, you have to create an object reference for a destination object, after which you can create a unit reference. To provide the unit with a queue name, assign the queue name using the method `add_queue_name_outbound`. It is not important whether you assign the queue name

before or after the function module call. After that step, you have a complete bgRFC unit object in which you can bundle as many function modules as you want. To store the data in the database table, you have to call a `commit work`. If you do not call `commit work`, a rollback happens and no data is saved. After the `commit work`, the bgRFC scheduler can pick the units from the database to execute them.

```

data:
  l_lock_id type bgrfc_lock_id.
  l_dest    type ref to if_bgrfc_destination_outbound,
  l_unit    type ref to if_qrfc_unit_outbound.

start-of-selection.
* Create the destination object references
  l_dest = cl_bgrfc_destination_outbound=>create( 'TEST_BGRFC' ).

* Create new bgRFC outbound unit type Q object reference
. l_unit = l_dest->create_qrfc_unit( ).

* Assign the unit to a queue
  l_unit->add_queue_name_outbound( 'BASIS_BGRFC_OUT_Q1' ).

* Register first function module
  call function 'RFC_FUNCTION_1' in background unit l_unit.
* Register second function module
  call function 'RFC_FUNCTION_2' in background unit l_unit.

* Lock unit
  l_lock_id = l_unit->lock( ).

* Save the call
  commit work.

```

**Figure 23** Create outbound unit type Q

## Creating a bgRFC out-inbound unit of type Q

**Figure 24** on the next page shows an example for the out-inbound scenario. As depicted in the code, we decided to deliver a separate interface to create out-inbound units to distinguish them from the outbound and the inbound scenario. As already mentioned, the main advantage of the out-inbound scenario is the separation of communication and execution of a bgRFC.

As in the previous examples, we start with creating a reference to a destination object. The destination object provides the corresponding factory method, which supplies the reference to the unit interface. In the out-inbound scenario, you can choose two queue

names, one for the outbound queue and one for the inbound queue. The queue names need not be different — you can use the same queue name for both the outbound and the inbound queue. This makes it easier to track the units as they move from the outbound queue to the inbound queue. After the queue names are assigned, you can register as many function modules as you want. In contrast to the outbound and inbound scenarios, you may lock the unit on both sides if you want. To lock the unit on the outbound side, use the unit method `lock( )`, and to lock the unit on the inbound side, use the method `lock_at_inbound( )`.

With the `COMMIT WORK` statement, all units are saved to the database.

```

data:
  l_lock_id_in      type bgrfc_lock_id.
  l_lock_id_out     type bgrfc_lock_id.
  l_dest            type ref to if_bgrfc_destination_outbound,
  l_unit            type ref to if_qrfc_unit_outinbound.

start-of-selection.
* Create the destination object references
  l_dest = cl_bgrfc_destination_outbound=>create( 'TEST_BGRFC' ).

* Create new queued bgRFC OutInbound unit type Q object reference
  l_unit = l_dest->create_qrfc_unit( ).

* Assign the unit to a queue
* Outbound
  l_unit->add_queue_name_outbound( 'BASIS_BGRFC_OUTIN_Q1' ).
* Inbound
  l_unit->add_queue_name_inbound( 'BASIS_BGRFC_OUTIN_Q1' ).

* Register first function module
  call function 'RFC_FUNCTION_1' in background unit l_unit.
* Register second function module
  call function 'RFC_FUNCTION_2' in background unit l_unit.

* Lock unit
* Outbound
  l_lock_id_out = l_unit->lock( ).
* Inbound
  l_lock_id_in = l_unit->lock_at_inbound( ).

* Save the call
  commit work.

```

**Figure 24** Create queued out-inbound unit

## The update task and bgRFC

If bgRFC and the update task are used within the same application commit work cycle, you get dependencies between the bgRFC units and the function modules called in the update task. In **Figure 25**, the function module UPDATE\_FUNCTION\_1 is called in the update task. The bgRFC function modules will be executed automatically only if the update function module is

executed. Until the update task has finished, the bgRFC units are locked.

The bgRFC lock will be released and the unit picked up from the bgRFC scheduler only if the update task is executed correctly. While the update process is processing, the unit is locked with the state wait for update. This behavior is the same for all bgRFCs (transactional inbound, transactional outbound, queued inbound, and queued outbound).



```

* Create bgRFCs
...
* Call update task
  call function 'UPDATE_FUNCTION_1' in update task.

commit work.

```

**Figure 25** Call a function module in update task

## How bgRFC units are processed — the bgRFC scheduler

We can now address the issue of *how* bgRFC units are processed. After they have been written to the database at COMMIT WORK, they are processed by a scheduler. There are two types of schedulers: one for inbound units and one for outbound units. For each of them, one or more program instances can be configured to run on a specific application server. The schedulers have to cope with several tasks: load balancing issues, resource management, and execution coordination to avoid a double execution.

The AUTO ABAP process reads the configuration tables, checks the actual number of running schedulers, and then restarts or kills them. If there are no units to process, the inbound or outbound scheduler goes into a sleep mode for a configurable time. It is awakened by an event if new units are written to the queues. In addition, if the idle time has elapsed, the scheduler actively checks the database tables for executable units. In this way, the possible loss of an event can be counteracted.

Before we discuss the details of unit processing, we should mention what are *not* scheduler tasks. Schedulers are optimized for the fast execution of units. Therefore the scheduler does not check dependencies of units at execution time. The dependencies are determined at the creation time of units, i.e., when COMMIT WORK is processed. This ensures a linear scalability of execution, especially for strongly dependent queues. After execution of a unit, its successors are updated. The ones without further predecessors and locks are copied into a special, central database table

for immediate execution. All active schedulers pick their work package of units from this table.

For each unit of the work package, the first thing the scheduler does is ensure that the destination of the unit is up and available. After this precondition has been fulfilled, the scheduler checks to see if the local and the remote gateway have enough resources for the unit transfer.

Furthermore, the scheduler has to ensure that both the local and target systems have enough resources to execute the units, i.e., a free dialog work process on each system. At this point some intelligence comes into play. To optimize the runtime and to reduce network load, the scheduler does not always poll information about the receiving system. Each time a unit is processed, the resource information of the destination system is sent back to the sending system. It is cached together with a timestamp on the sending system. All scheduler tasks that are working with the same destination system use this stored data to decide whether the unit can be started or not. Polling of resource information is necessary only if no up-to-date cached information is available.

If enough resources for sending and executing a unit are available, the scheduler will send the function modules to the target system and execute them asynchronously. Therefore every scheduler will execute several independent units in parallel. At a later time, the scheduler will wait for the responses of the target system and ensure that the units have been executed completely and successfully. If, on the other hand, some resource is lacking, the unit will be locked for a limited time and the scheduler will try to execute the unit subsequently.

Of course, the scheduler also has to check for those units for which a response has been missing for a long time. For those units, it must verify whether they are still running or if their execution has been aborted. This may be the case, for example, when an unrecoverable error has occurred on the receiving system. These occurrences must be distinguished from those cases in which the network may have temporarily disconnected and therefore the response has been lost. (This holds true for both outbound and inbound schedulers.)

It may occur to you to configure as many schedulers as possible to maximize the overall throughput. But the situation is unfortunately not that simple — the more schedulers that are configured, the higher the effort for coordination. Furthermore, the I/O capacity of the database server restricts the maximum number of executable units.

## Advantage of the out-inbound scenario

To appreciate the advantage of the out-inbound scenario, let's examine a common example of an ABAP communication. Normally, an ABAP system sends and receives requests from different ABAP instances. Neither the time of occurrence nor the runtime of a task can be evaluated. If, for example, a transactional bgRFC is triggered on ABAP instance A and should be executed on a second ABAP instance B, and some other ABAP instances send a lot of requests to ABAP instance B, it is possible that not all sent requests can be executed on the server system immediately. In this case, all incoming requests that cannot be executed immediately will be queued in the dispatcher of ABAP instance B until a work process becomes free. On the sending machine, the work processes will be kept blocked until the dispatcher gets free work processes for the task. In the case where a lot of applications are executed on ABAP instance B with long runtimes, you can easily imagine that the work processes on the sending system will be blocked for a long time for all following requests. Because these triggered transactional bgRFCs are stuck in the dispatcher queue and waiting for a free work process, an overflow of the dispatcher

queue could result as well as a huge amount of memory allocation on the server system.

Because a receiver system has no control over the incoming requests, it is beneficial to reduce the time to handle the incoming requests. You can do this using the out-inbound scenario.

In this case, the scheduler on the sending system sends the data to the destination system. Instead of being immediately executed there, the data is stored on the database tables to minimize the communication time.

The work process on the sending system is then blocked as long as the destination system needs to store the application data in the database. The inbound scheduler of the destination system determines whether the destination system has the resources (memory, work processes, etc.) required to process the application data. For the inbound scheduler, the resources check can be easily done, because it is only a check on the local instance. The scheduler will start processing the bgRFC units only if enough resources are available. The result is a more stable system load situation and the sender system will not be blocked by weak server systems during the whole application runtime.

Now that you now know how to use bgRFC in your programs, let's take a look at the bgRFC destinations that enable the functionality to work.

## Configuring bgRFC destinations

In the following sections, we look at how outbound and inbound destinations differ, and how they are configured.

### Outbound destinations

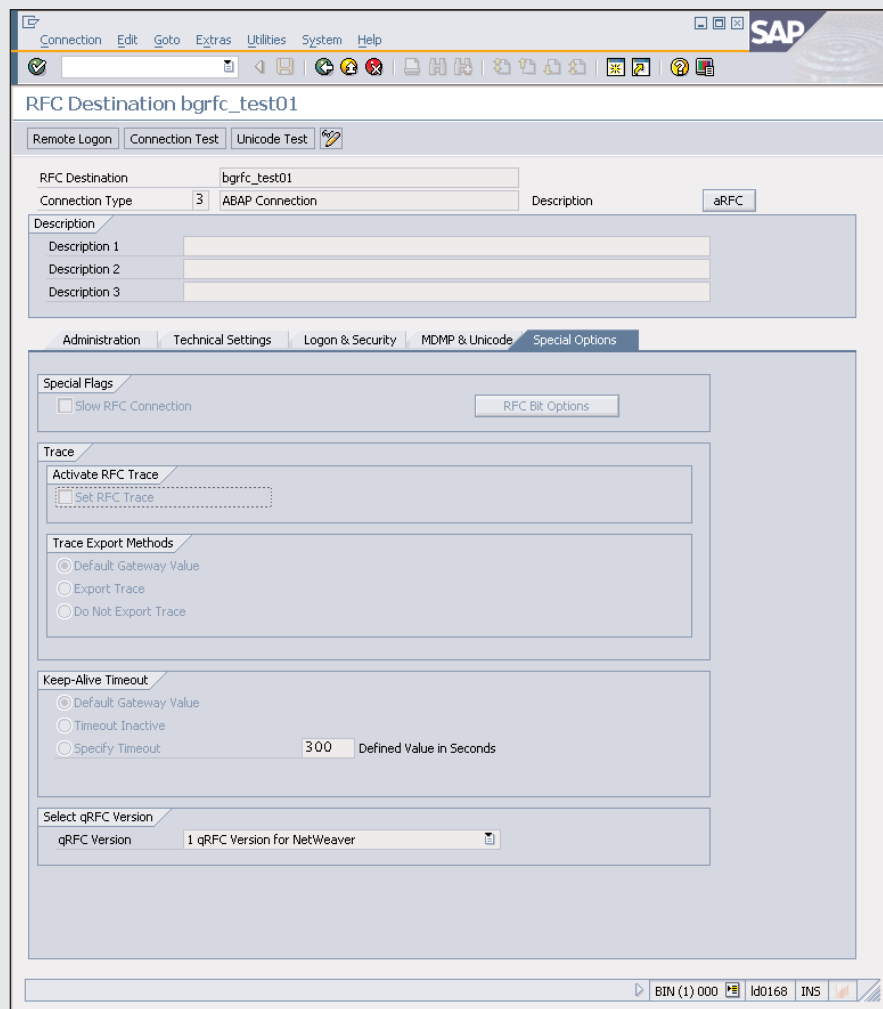
bgRFC units sent to an outbound destination will normally be executed on a remote system, i.e., on a system that is different from the calling one. But there is no need for calling and receiving systems to be different. It is also possible just to change the client or the user and to keep the system the same, or even to execute the bgRFC unit on the very same application server with the same user and the same client.

There are many ways to configure an outbound destination. For this task, call transaction SM59. We do not have room here to discuss all of the possibilities for configuring destinations in SM59; the important issue for bgRFC purposes is to enable bgRFC for a destination.

At this point, we need to take yet another little excursion. The bgRFC framework is a reimplementa-tion of an existing framework (classic tRFC and qRFC). The bgRFC framework and tRFC/qRFC framework and their schedulers exist now in parallel, because the APIs of these two implementations are

completely different. To avoid serializations between the schedulers, for every SM59 destination, the administrator must define whether the destination works with the legacy (tRFC/qRFC) or bgRFC implementation.

You make this distinction by selecting the qRFC version. It may be 0 (i.e., Classic qRFC version) or 1 (i.e., Version qRFC for NetWeaver), which is, in fact, the bgRFC implementation. (By default a destination will have the setting Classic qRFC.) You find the qRFC version on the Special Options tab, as shown in **Figure 26**.



**Figure 26** Define RFC destination (transaction SM59)

## Inbound destinations

Inbound destinations are completely different. An inbound unit is always executed on the calling system. You cannot change the client, the user, or the language. And you do not even maintain them using SM59. So of what use is the notion of a destination in this context? Before we discuss this philosophical point, let's concentrate on the practical side: inbound destinations must be maintained via transaction SBGRFCMAINIDST. In contrast to transaction SM59, there are only two things to maintain: the name of the inbound destination, and the server group. You maintain server groups using transaction RZ12, as shown in **Figure 27**.

The corresponding server groups are called *RFC server groups*, which differ from the logon groups you maintain via transaction SMLG. The load balancing information of an RFC server group is updated much more quickly than the load balancing information of the logon groups and therefore much more suitable for RFC load balancing. From the maintenance point of view, RFC server group is very similar to the logon groups. Each group consists of one or more application servers of the system. A unit sent to an inbound destination corresponding to an RFC server group will be sent to the server in the maintained group that currently has the lowest load.

The second maintenance transaction (SQRFCUSTIDST) comes into play when bgRFC inbound units of type Q are to be created. This brings up an interesting point: In contrast to the outbound case, destination and queue names are not independent of each other; it is not possible to have the same queue name for two destinations. Each queue name corresponds to exactly one inbound destination. Therefore, in the maintenance transaction, each queue prefix is assigned to one destination. But one destination may have more than one queue prefix assigned to it.

In summary, we can say that an outbound destination is a real remote system, whereas an inbound destination is just a name for a group of queues. So why do such different concepts have identical nomenclature?

Destinations serve to group queues; this is the case for both outbound and inbound destinations. Each

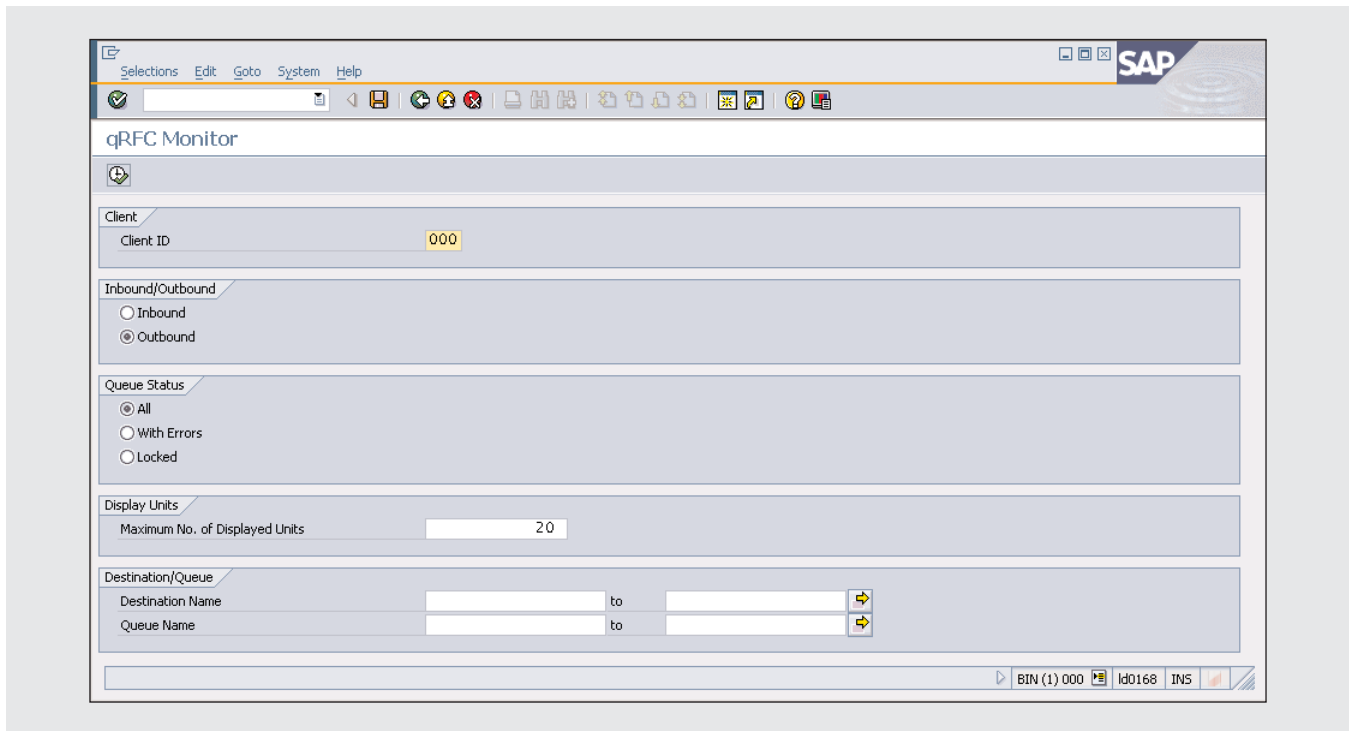
Logon Group	Instance	Status
LOTTRFC	1s3022_BIN_09	
MASOUD	binmain_BIN_53	
MASOUD	pldf0195_BIN_53	
MASOUD_2	ldp007_BIN_29	
MASOUD_BINMAIN	binmain_BIN_53	
MASOUD_NT	pldf0195_BIN_53	
MASOUD_NT	pldf2310_BIN_53	
MY_QRFC	binmain_BIN_53	
P127866	P127866_BIN_53	
P58055	P58055_BIN_53	
PBT	hs0311_BIN_53	
PBT	pcintel_BIN_53	
PBT	pn0202_BIN_53	
PBT	ss0011_BIN_53	
PRFC	hs0311_BIN_53	
PRFC	hwd1364_BIN_53	
PRFC	hwd1426_BIN_53	
PRFC	is0007_BIN_53	
PRFC	pn0202_BIN_53	
PRFC	ss0011_BIN_53	
PRFC	us0201_BIN_53	
QRFC_BIN	binmain_BIN_53	
QRFC_CARL	1s3022_BIN_17	
QRFC_I	binmain_BIN_53	
SCHMITTAN	P58055_BIN_53	
WEBMAIN	binmain_BIN_53	
WEBMASTER	binmain_BIN_53	
WERNER	1s3022_BIN_11	
WREHM	binmain_BIN_53	
WREHM	1s3022_BIN_12	
eltest	binmain_BIN_53	
eltest	1s3022_BIN_14	
huey	binmain_BIN_53	
huey	ds0304_BIN_53	
titanium	itanium06_BIN_53	
micky	hwd1393_BIN_53	
parallel_generators	P120100_BIN_53	
parallel_generators	WDFD00100250A_BIN_53	
parallel_generators	WDFD00114601A_BIN_53	
parallel_generators	WDFD00146517A_BIN_53	
parallel_generators	p67476_BIN_53	
parallel_generators	pldf2697_BIN_53	
qRFC	binmain_BIN_53	
sasan	is0206_BIN_53	
webmaster	us0301_BIN_53	

**Figure 27** Maintain server groups (transaction RZ12)

application can maintain its own set of destinations and therefore find its own queues. The second reason is symmetry — although inbound destinations have fewer features than outbound destinations, they are both used to create instances for units. The API has become easier to understand, because the handling in outbound and inbound scenarios is very similar.

## Monitoring bgRFC units

Because the processing of units takes place asynchronously, there is a need to monitor if queues are



**Figure 28** Selection screen of the bgRFC monitor transaction SBGRFCMON1

stopped, if a certain unit has been processed, which function modules have been registered, and which queues a unit is in. For debugging purposes, a queue may have to be stopped, a unit may need to be debugged, or a forgotten lock may have to be released. All of these tasks can be executed by the bgRFC monitor.

In fact, there are two such monitors: SBGRFCMON1 for type Q, and SBGRFCMON2 for type T bgRFC units. They differ mainly in the select options available at the time the monitor is started, but their main functionality is similar. Therefore we will describe monitoring the outbound scenario for bgRFC units of type Q with the help of transaction SBGRFCMON1.

## Monitoring queued bgRFC — transaction SBGRFCMON1

The first selection screen, shown in **Figure 28**, lets you choose the client (inbound or outbound) and

restrict destination and queue names. Another option lets you look at locked or erroneous queues.

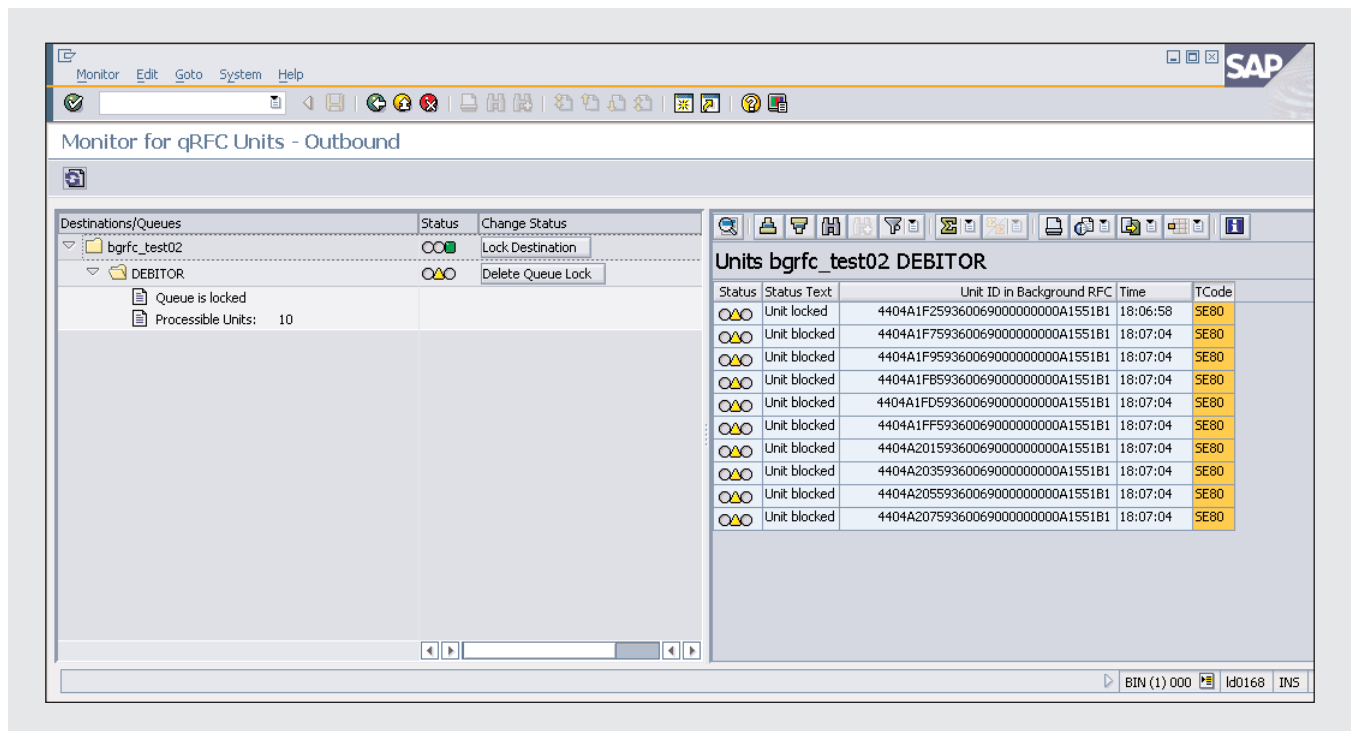
### *Note!*

Although all erroneous queues are locked, not all locked queues are erroneous.

The left column of the screen shown in **Figure 29** on the next page has a hierarchical view of the bgRFC with the names of the destinations and queues. The top level is built up by the destinations, and below the destinations are the queues. Both the destination and queues are shown with their current state:

- The state of a destination can be green (units of this destination will be processed by the scheduler), or it can be yellow (the destination is locked,





**Figure 29** Screen of bgRFC monitor SBGRFCMON1

meaning the scheduler will not execute any unit of this destination).

- The state of a queue can be green (units of this queue will be processed), yellow (the queue is locked or blocked, meaning the units will not be executed because of an existing lock or an unaccomplished dependency on units of other queues), or red (an error has occurred).

Double-clicking on a queue name fills the grid control on the right side with the first units registered to the queue. The first units are those that the system executes next.

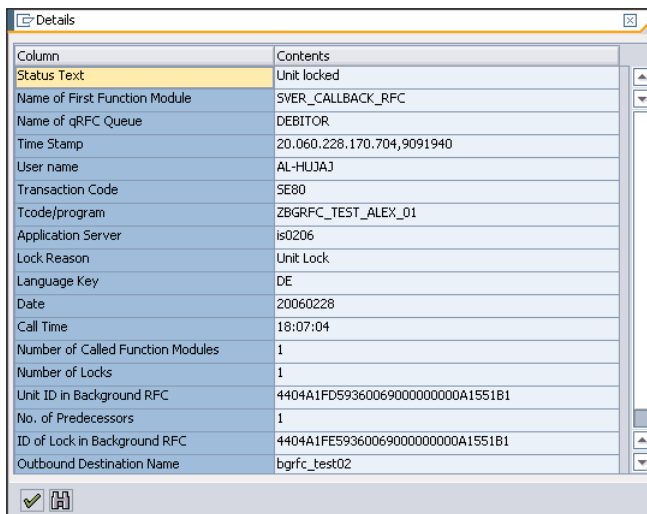
Below the queue, you find some more details: the state of the queue and the number of units that are processable (those units that the system has not yet successfully processed).

For every unit, a variety of information is available. The view of the information is customizable. In this example, you see that the unit is locked, the unit ID, the creation time of the unit, and the transaction code of the creating program (SE80).

You get a more extensive view of the unit information if you double-click on its corresponding entry, as shown in **Figure 30**. Here you find more information, such as the status text (the textual description of the status of the unit), the name of the first function module (SVER\_CALLBACK RFC), the name of the queue (DEBITOR), and many others.

In **Figure 29**, note the two buttons in the tree: Lock Destination and Delete Queue Lock. With the first one, you lock the destination, which keeps the scheduler from further processing units for this destination. This affects all units sent to this destination, regardless of whether they are type T or type Q units, but not those units that are already prepared for execution by the schedulers.

The second button lets you unlock a queue. A queue lock is just a lock on a unit. Any unit lock will become a queue lock if the unit has reached the first position of a queue. Therefore, you will find in the Details window (similar to the one shown in **Figure 30**) a lock reason “Unit locked,” although it is shown in the tree as a queue lock. However, there are also



Column	Contents
Status Text	Unit locked
Name of First Function Module	SVER_CALLBACK_RFC
Name of qRFC Queue	DEBITOR
Time Stamp	20.060.228.170.704,9091940
User name	AL-HUJAJ
Transaction Code	SE80
Tcode/program	ZBGRFC_TEST_ALEX_01
Application Server	is0206
Lock Reason	Unit Lock
Language Key	DE
Date	20060228
Call Time	18:07:04
Number of Called Function Modules	1
Number of Locks	1
Unit ID in Background RFC	4404A1F95936006900000000A1551B1
No. of Predecessors	1
ID of Lock in Background RFC	4404A1FE5936006900000000A1551B1
Outbound Destination Name	bgrfc_test02

**Figure 30** Detail information for a unit (partial)

queue locks that can be set via the monitor as well. They will result in locks on units, too, but the monitor display will show “Queue lock at start of Queue” or “Queue lock at end of Queue” as the lock reason.

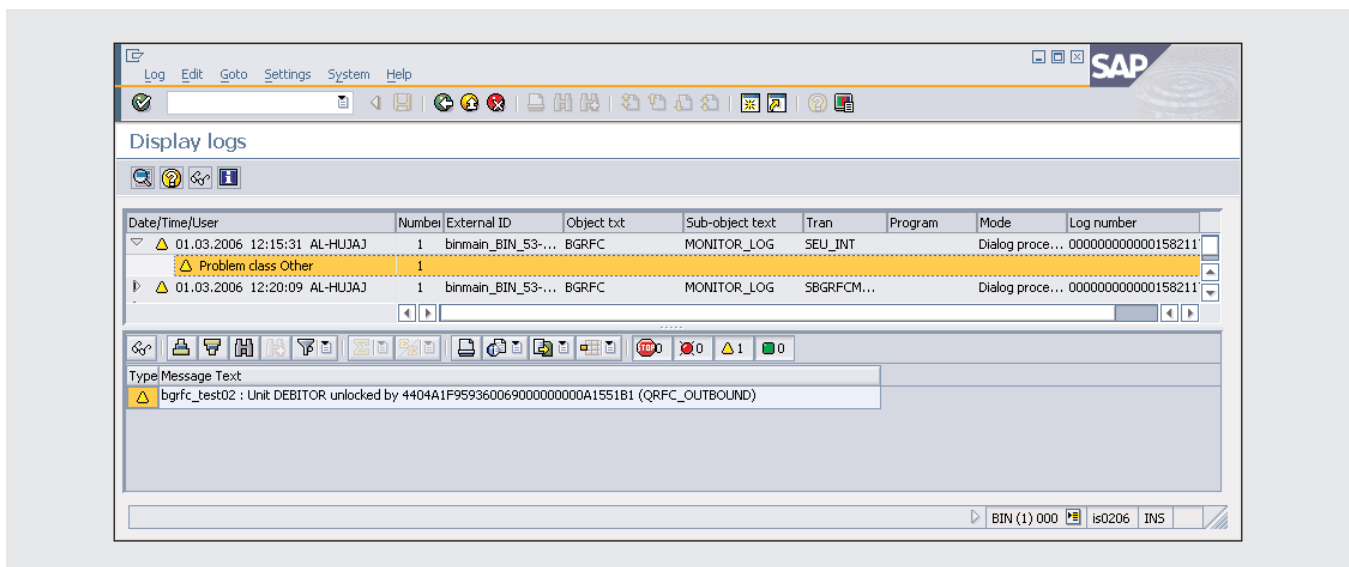
The main difference between locks set on queues and locks set on units is the point in time at which they can be set. Unit locks (set by a method of the `IF_BGRFC_UNIT` interface) can only be set before `COMMIT WORK`; therefore no unit locks can be set when

the unit becomes visible in the monitor, whereas queue locks can be set from the monitor. Normally, the user has no precise control over which unit will be locked (because the schedulers execute units in the background) and therefore the unit the user wants to lock may have already been executed.

Interfering with scheduler processing — e.g., stopping the processing by setting a lock or starting it by removing it — is a delicate task. It may lead to unexpected application behavior and may raise problems or questions. On the one hand, authorization for any intervention can be finely controlled by the authority object `S_BGRFC`. On the other hand, every action that changes the processing is written to the application log under the object `BGRFC`. In the example shown in **Figure 31**, you find the message that the queue has been unlocked in the monitor.

## Basic bgRFC scheduler configuration

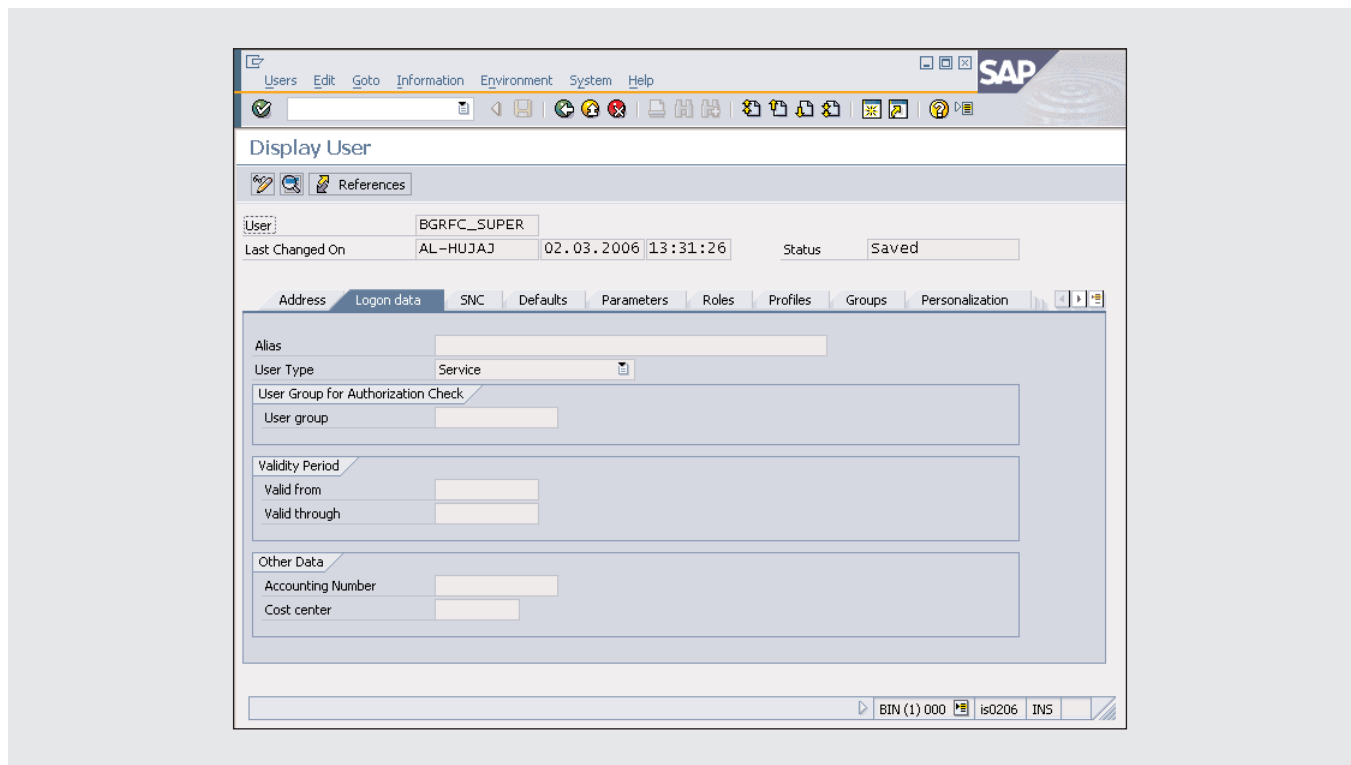
Before bgRFC schedulers can work, some customizing is required. With the default configuration, no scheduler will run and therefore no bgRFC units will be processed. If you see, perhaps, that the units that



Date/Time/User	Number	External ID	Object txt	Sub-object text	Tran	Program	Mode	Log number
01.03.2006 12:15:31 AL-HUJAJ	1	binmain_BIN_53-...	BGRFC	MONITOR_LOG	SEU_INT			Dialog proce... 000000000000158211
<div> <div>Problem class Other</div> <div>1</div> </div>								
01.03.2006 12:20:09 AL-HUJAJ	1	binmain_BIN_53-...	BGRFC	MONITOR_LOG	SBGRFCM...			Dialog proce... 000000000000158211

Type Message Text  
 bgrfc\_test02 : Unit DEBITOR unlocked by 4404A1F95936006900000000A1551B1 (QRFQ\_OUTBOUND)

**Figure 31** Application log (transaction SMLG1) for object BGRFC, showing two log entries



**Figure 32** Transaction SU01 for the scheduler user

you created with the example code are not being processed, then the system has not been customized for running a bgRFC scheduler.

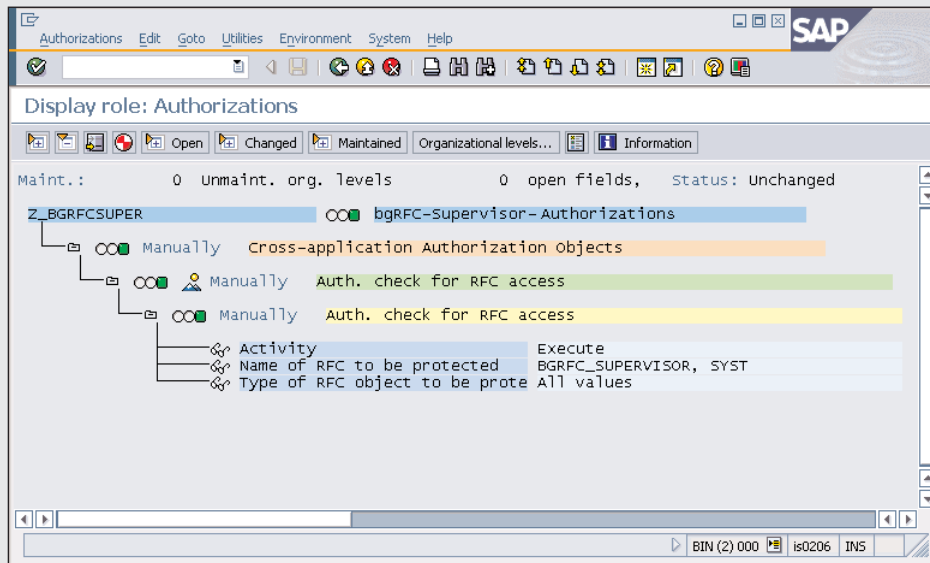
The most important element of scheduler configuration with which to be concerned is the scheduler user. This user and password are attached to an RFC destination. You must create the scheduler user in transaction SU01, as shown in **Figure 32**. Normally, you would create a service user to prevent security issues (service users are not allowed to log in).

Because the scheduler triggers RFC calls, and these are authority checked, the scheduler user needs special authorities. Two function modules have to be called: RFC\_PING from the function group SYST and BGRFC\_CHECK\_SCHEDULERS from the function group BGRFC\_SUPERVISOR. The first one is called during the processing and checks to see if a destination is available. The second one is called during the recurring check to see if the configured number of schedulers is still running.

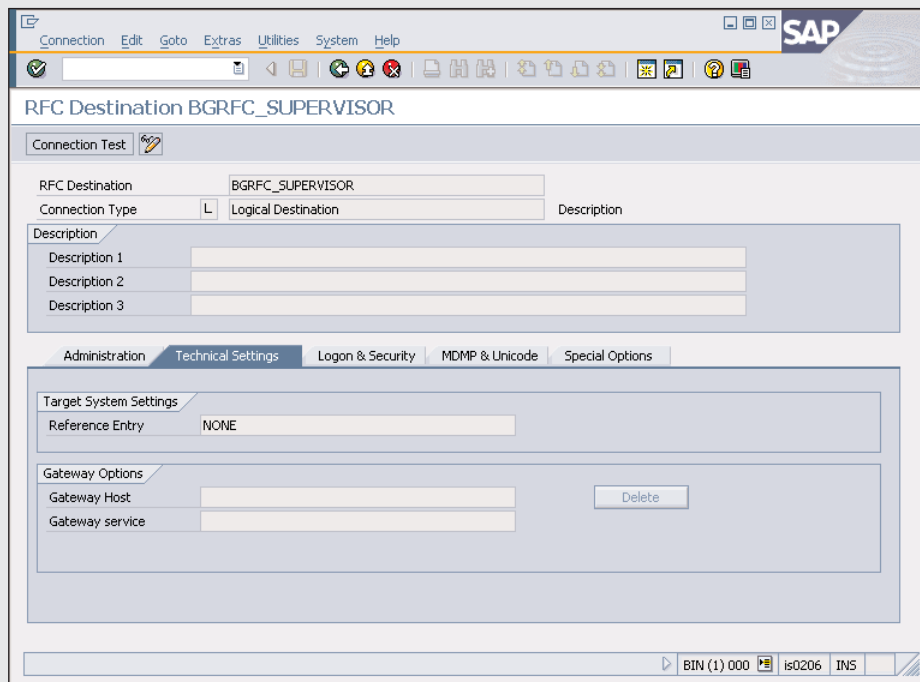
The scheduler user configuration must contain the S\_RFC (authority check for RFC access) authority object with activity “Execute,” and the name of the RFC to be protected (i.e., BGRFC\_SUPERVISOR and SYST, which are the function groups just mentioned). **Figure 33** shows what this minimum required authority should look like.

After you have configured the scheduler user and given it the correct authorities, you can call transaction SM59 to create the destination. This destination does not serve for a remote connection, but for local processing by a definite user (including his/her password). Therefore, you have to ensure, by the configuration, that the processing takes place on the very same application server. This can be done in two ways: as a logical destination or as an ABAP connection.

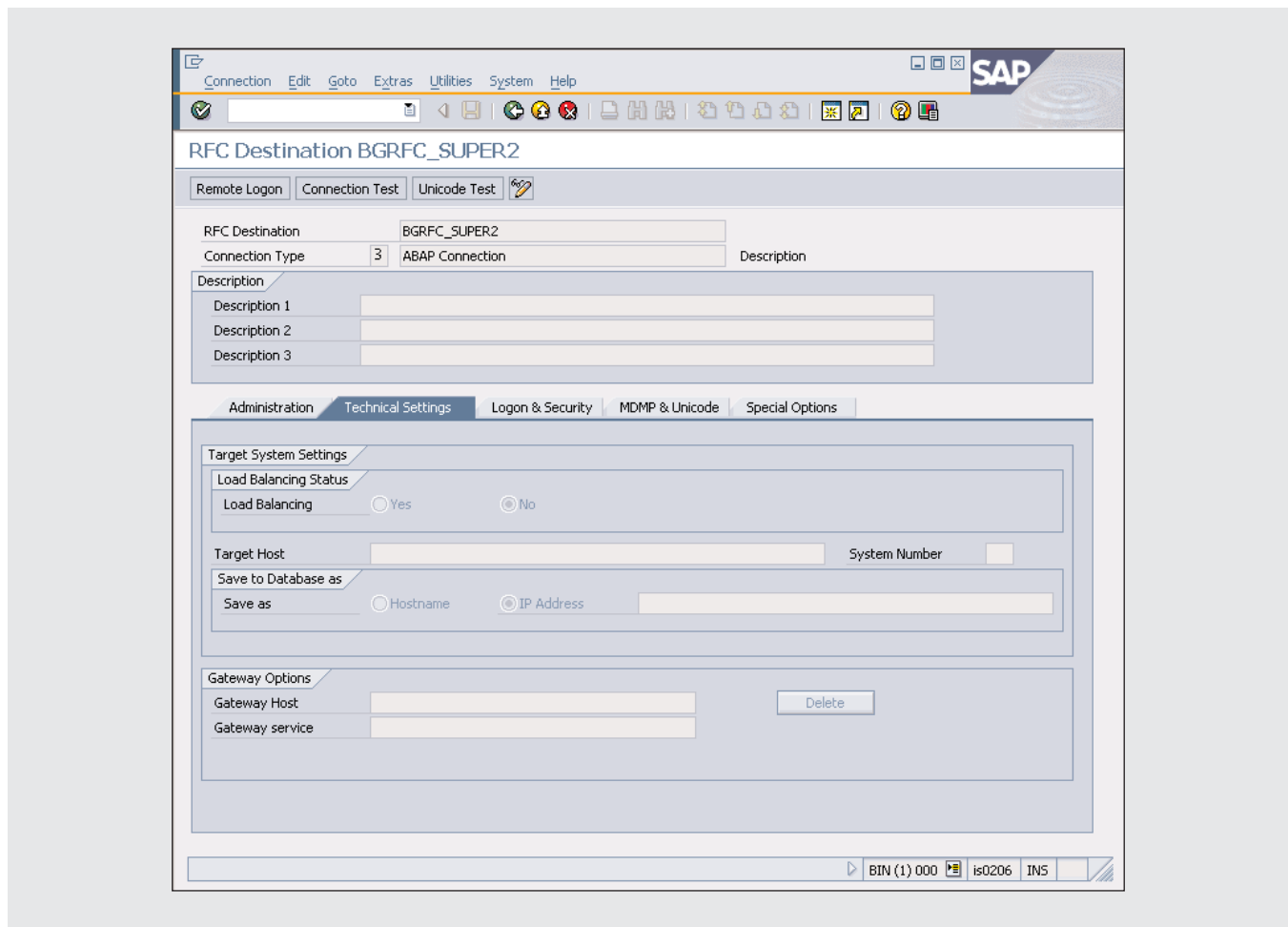
In the example shown in **Figure 34**, we have chosen a logical destination with the reference entry “NONE” and no gateway host or service.



**Figure 33** Minimum authorities needed for the scheduler user in transaction PFCG



**Figure 34** SM59: Technical settings for a logical destination



**Figure 35** Destination for the scheduler user: Technical settings for an ABAP connection

### **Note!**

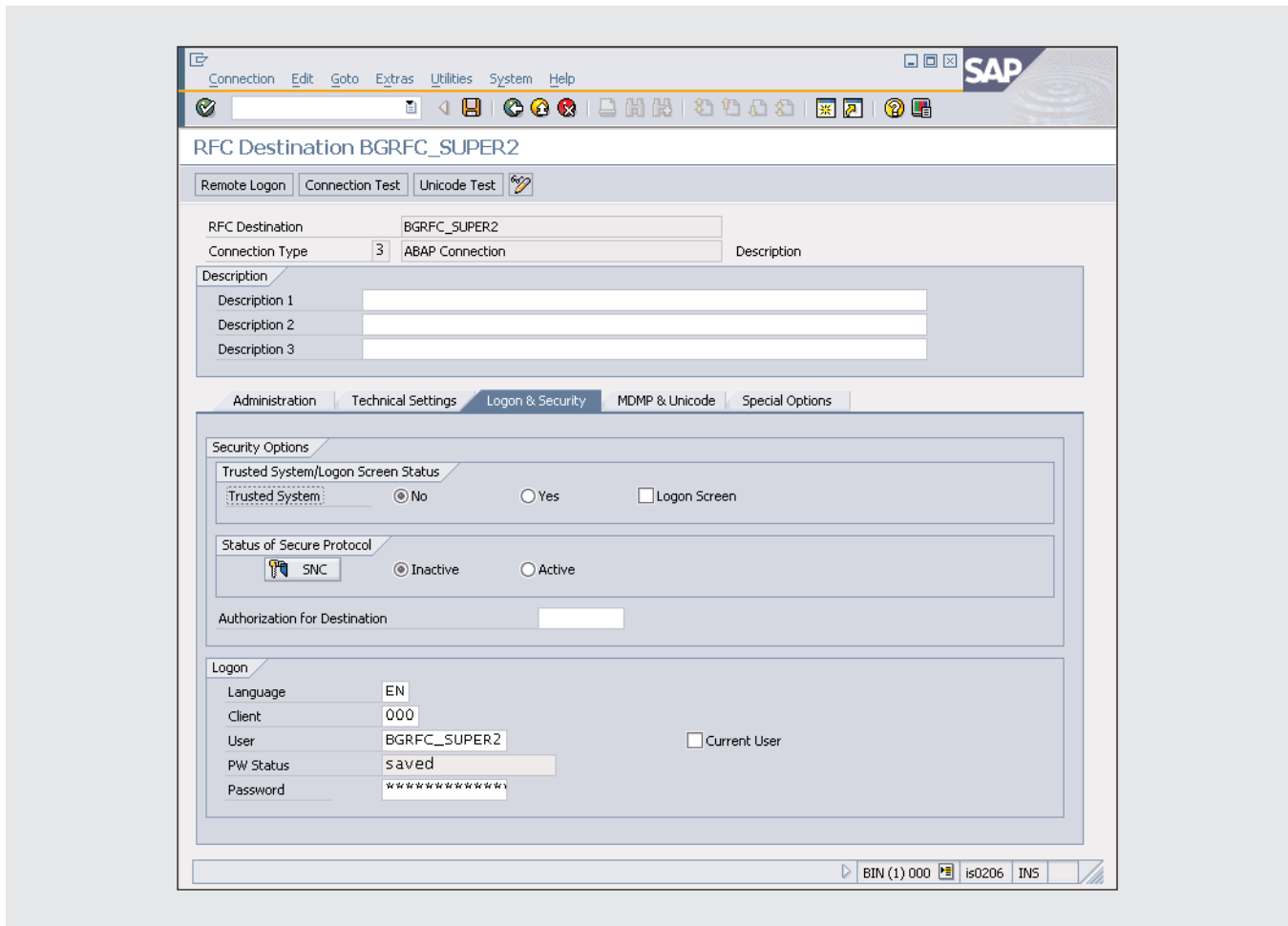
The destination shown in **Figure 34** is not an arbitrary one, but one that contains the user for the scheduler, i.e., the supervisor destination. This figure also lists some of the technical settings.

Another option is to set up the destination as an ABAP connection without load balancing, target host, gateway host, and gateway service, as in the example shown in **Figure 35**.

For both solutions, the main ingredient is to specify the user, language, and password. This ensures that the scheduler will be executed on the very same application server as the specified user. The required logon and security settings are shown in **Figure 36**.

Having set up the user and the destination, we are now ready to enter the configuration transaction that writes the destination to the configuration tables. We call this destination a *supervisor destination*, because the schedulers are started by a “supervisor” that is called during the AUTO ABAP process. Therefore, the transaction to maintain the supervisor destination is named SBGRFCSUPER. It not only checks the destination before it writes it to the corresponding database tables, it also locks it against changes in SM59. If it





**Figure 36** Destination for the scheduler user: Logon and security settings

were easy to change the destination (e.g., by deleting it or changing the password), then no scheduler would run, and a large amount of short dumps would be found on the system.

We have described enough basic configurations to enable bgRFC processing. The default configuration entries that are shipped within the system will normally make sure that after a run of the AUTO ABAP process, one inbound scheduler and one outbound scheduler are started on every application server.

The only precondition for this is a sufficient amount of dialog work processes. At least two free dialog work processes plus one work process for each scheduler are needed to run schedulers on an applica-

tion server. That is, if you want to run one inbound and one outbound scheduler, you need at least six dialog work processes on the respective application server.

A more elaborate configuration of the system is possible to maximize the throughput, but a discussion of advanced configuration is beyond the scope of this article.

## Conclusion

This article introduced you to the new bgRFC framework for processing a large number of asynchronous business transactions. It presented the most important

concepts of bgRFC: destinations, queues, units, and dependencies. Furthermore, it demonstrated the classes and interfaces of this framework and provided you with some simple programming examples for all kinds of bgRFC units. The essential manner of operation of the bgRFC scheduler was described as well as basic monitoring and configuration. This fundamental article on the bgRFC framework will enable you to get the bgRFC schedulers working in your own SAP NetWeaver 2004s system and to start coding your first bgRFC programs. While bgRFC can work alongside your existing implementations, we hope we convinced you to use bgRFC in all new implementations for highly scalable integration solutions.

---

*Wolfgang Baur has more than 14 years of experience in software development and support. He joined SAP in 1998 and currently works as a developer in the ABAP Connectivity group, where he is responsible for tRFC and qRFC. During the last two years, he has directed the bgRFC project as project lead and has worked on the design, implementation, and tools of the ABAP communication infrastructure. You may reach him at [wolfgang.baur@sap.com](mailto:wolfgang.baur@sap.com).*

*Omar-Alexander Al-Hujaj studied physics and received his doctorate in theoretical atomic physics at the University of Heidelberg. He joined SAP in 2004 and became a member of the ABAP Connectivity group, where he works on the design, implementation, and tools of the ABAP communication infrastructure. Omar-Alexander is responsible for monitoring and configuration interfaces and tools of the bgRFC framework. You may reach him at [omar-alexander.al-hujaj@sap.com](mailto:omar-alexander.al-hujaj@sap.com).*

*Wolfgang Röder joined SAP in 1990. Earlier, he worked for SAP customers. He is currently a development architect in the ABAP language support group. In his long history with SAP, he has worked on ABAP test tools, customizing tools, data archiving (ADK), and Enterprise Application Integration between SAP applications and standard applications from SAP competitors. In addition, he has been an SAP technical consultant, SAP system administrator, and development manager for some of the aforementioned areas. You may reach him at [wolfgang.roeder@sap.com](mailto:wolfgang.roeder@sap.com).*