Transaction Handling in SAP R/3 — What Every Programmer Needs to Know

Thomas G. Schuessler

Thomas G. Schuessler is the founder of ARAsoft, a company offering products, consulting, custom development, and training to customers worldwide, specializing in integration between SAP and non-SAP components and applications. Thomas is the author of SAP's CA925 "Programming with BAPIs in Visual Basic" class. Prior to founding ARAsoft in 1993, he worked with SAP AG and SAP America for seven years.

(complete bio appears on page 46)

How is it that R/3 is able to scale so well, handling even very large numbers of transactions with ease? Understanding the architecture of R/3 that makes this possible is required for developers who want to build performance-optimized applications as well as for administrators in charge of overall system performance. Even developers who create external BAPI-enabled applications need to understand the implications of SAP's architecture.

In this article, I will explain how transactions are handled in R/3. I will start with a brief discussion of the three-tier SAP client/server architecture. Then I will dive into the details of transaction processing. Because the asynchronous update scheme used in R/3 is such an integral part of this process, I will spend quite a bit of time on this topic. Lastly, I will cover the consequences of SAP's architecture for the ABAP programmers who write BAPIs and external application developers who use BAPIs in their software.

The SAP R/3 Three-Tier Client/Server Architecture

In order to provide the scalability required for any serious multiuser application, R/3 is built upon a three-tier architecture. **Figure 1** shows the three tiers.

These tiers can be run on one or more physical servers. It is common to see configurations, where:

Figure 1 The Three Tiers of an R/3 System

Presentation This is the "thin client" GUI layer.	• The standard presentation layer is SAPGUI[1], a graphical user frontend available for Windows and other platforms. • No application logic is executed in SAPGUI, it just displays a screen (called dynpro in the SAP vernacular) that enables the user to interact with the R/3 system. Alternatively, you can build your own presentation layer, one that calls application functionality using SAP's Remote Function Call (RFC) protocol.
Application Server[2] This is also known as an "R/3 Instance".	• All application logic is executed in the Work Processes (Services) of an Application Server. • While there can be only one Database Server in an R/3 system, multiple Application Servers can be used to support large numbers of clients. • The Application Server "speaks" SQL with the Database Server.
Database Server This is the relational database engine of an R/3 system.	• There can only be one Database Server in an R/3 system. • The Database Server can be any of the relational database products supported by R/3. • No SAP code runs in the Database Server.[3]

- The Database Server and one Application Server, supporting multiple clients, reside on the same physical server.[4]
- Multiple Application Servers, each supporting multiple clients, and each running on a separate physical server, use the same Database Server. (The Database Server can reside on the same physical server as one of the Application Servers.)

[1] The protocol used for the communication between SAPGUI and the Application Server is proprietary, but you can utilize the SAP Automation GUI Interface to support it in your own applications.

[2] Note that "Application Server" and "R/3 Instance" are synonymous. I will use the term "Application Server" throughout this article.

[3] Unless, of course, you are using SAP's own database, SAP DB (**http://www.sap.com/solutions/technology/sap_db.htm**).

[4] When I use the term "Server" in this article, I am referring to software. Otherwise, I will explicitly use the term "physical server".

Figure 2 *R/3 Work Process Types*

Type	Role	Total Number of Processes of This Type in an R/3 System	Number of Processes of This Type in One Application Server
Dialog	Executes dialog steps and remote function calls	2 or more	2 or more
Update	Performs asynchronous database updates	1 or more	0 or more
Enqueue	Manages enqueues (locks)	1	0 or 1
Spool	Controls printing	0 or more	0 or more
Background	Runs batch programs	1 or more	0 or more

Each Application Server consists of the following elements:

- A Dispatcher, which is responsible for accepting requests and assigning them to an available Work Process.
- Multiple Work Processes — **Figure 2** lists all the Work Process types you may find on an Application Server.
- Buffers, which are there to improve performance. They are not pertinent to the discussion at hand.

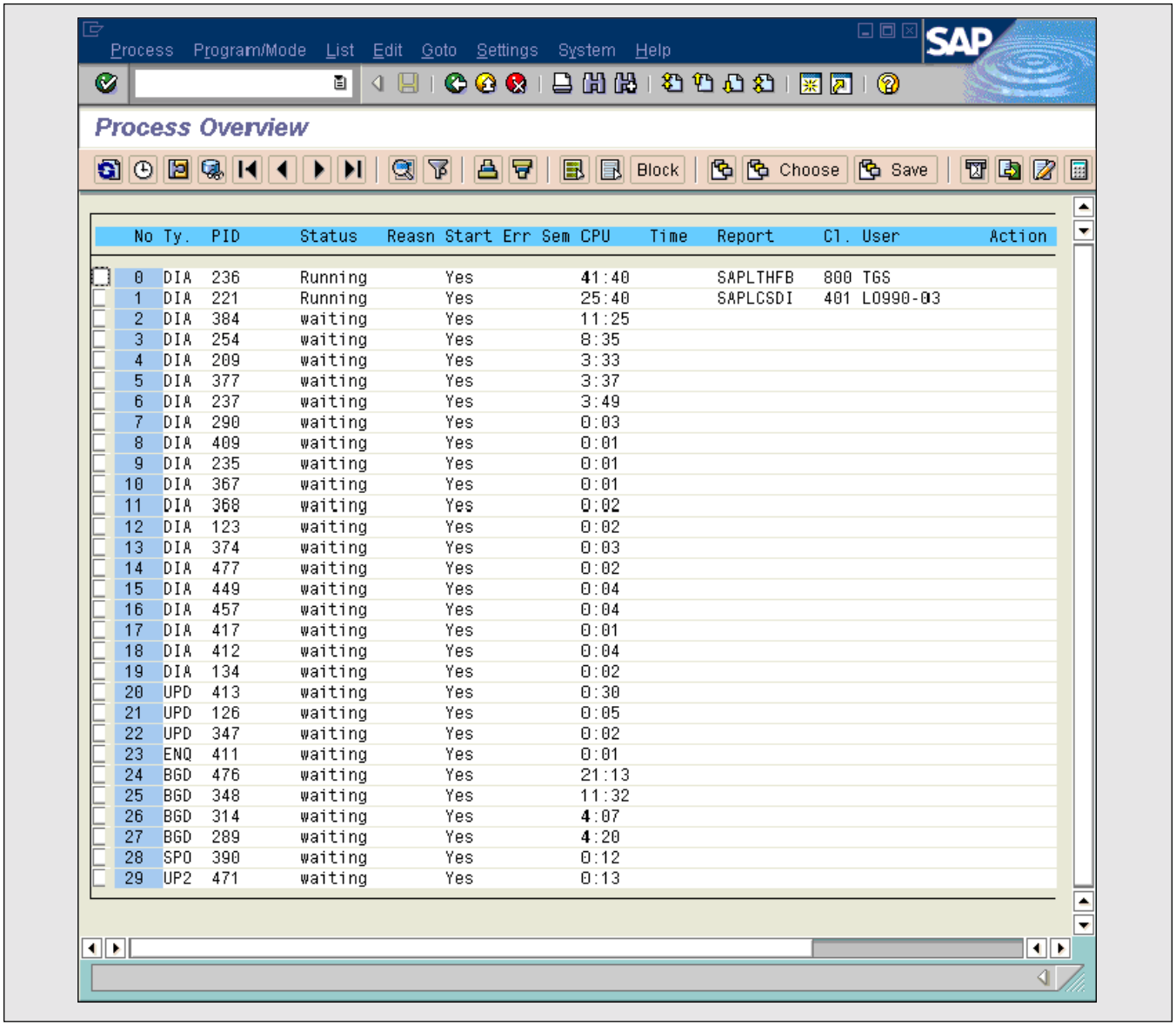
Each Application Server has two or more *Dialog Services*,[5] which is where most application activity takes place. There are two types of requests that are serviced in Dialog Services:

- **Execution of a request on behalf of an SAPGUI user:** Since SAPGUI is a thin client, all user interactions with SAPGUI (beyond simple activities like filling in data entry fields) require processing in the Application Server. When a user presses Enter, hits a function key, double-clicks with the mouse, and so on, SAPGUI sends an appropriate request to the Application Server. After the request has been processed, the next screen is sent back to SAPGUI and the user can continue. The execution of a request on behalf of an SAPGUI user is known as a *Dialog Step*.
- **Invocation of an ABAP function via RFC:** R/3 allows external (SAP and non-SAP) systems to invoke ABAP function modules via RFC. Each function call (for example, the invocation of a BAPI) is a request that is processed by a Dialog Service.

Update Services are responsible for asynchronous updates on behalf of activities in Dialog Services. This will be explained in more detail in the next section. The *Enqueue Service* allows an application to gain exclusive (or shared) control of an entity (e.g., a particular customer). Before updating something (e.g., a customer's address), an application usually needs to obtain exclusive control to prevent other users from interfering with the planned update. The *Spool* and *Background* Work Process types are of no

[5] Concrete instances of particular Work Process types are usually called "Servers" — e.g., "Enqueue Server". Since then an "Application Server" would contain other "Servers", which might be confusing, I will use the term "Service" instead. SAP's publications use both terms in this context, so I guess I can choose the one I find more appropriate for our discussion.

Figure 3 The "Process Overview" Transaction (SM50)



No	Ty.	PID	Status	Reasn	Start	Err	Sem	CPU	Time	Report	Cl.	User	Action
0	DIA	236	Running		Yes			41:40		SAPLTHFB	800	T6S	
1	DIA	221	Running		Yes			25:40		SAPLCSDI	401	LO990-03	
2	DIA	384	waiting		Yes			11:25					
3	DIA	254	waiting		Yes			8:35					
4	DIA	209	waiting		Yes			3:33					
5	DIA	377	waiting		Yes			3:37					
6	DIA	237	waiting		Yes			3:49					
7	DIA	290	waiting		Yes			0:03					
8	DIA	409	waiting		Yes			0:01					
9	DIA	235	waiting		Yes			0:01					
10	DIA	367	waiting		Yes			0:01					
11	DIA	368	waiting		Yes			0:02					
12	DIA	123	waiting		Yes			0:02					
13	DIA	374	waiting		Yes			0:03					
14	DIA	477	waiting		Yes			0:02					
15	DIA	449	waiting		Yes			0:04					
16	DIA	457	waiting		Yes			0:04					
17	DIA	417	waiting		Yes			0:01					
18	DIA	412	waiting		Yes			0:04					
19	DIA	134	waiting		Yes			0:02					
20	UPD	413	waiting		Yes			0:30					
21	UPD	126	waiting		Yes			0:05					
22	UPD	347	waiting		Yes			0:02					
23	ENQ	411	waiting		Yes			0:01					
24	BGD	476	waiting		Yes			21:13					
25	BGD	348	waiting		Yes			11:32					
26	BGD	314	waiting		Yes			4:07					
27	BGD	289	waiting		Yes			4:20					
28	SPO	390	waiting		Yes			0:12					
29	UP2	471	waiting		Yes			0:13					

relevance for our discussion, so I will not elaborate on either.

Figure 3 shows a screen shot displaying the various Work Processes (Services) in one Application Server. As you can see, there was not a lot of activity when I took the screen shot. (The difference between "UPD" and "UP2" processes will be explained later.)

What you do not see in the figure is the Message Service[6]. This service runs on just one Application Server. Its role is to facilitate communications between multiple Application Servers (and the Services running inside them). Due to the importance of the Message Service, the Application Server on which it runs is given a special name — the "Central R/3 Instance".

[6] Often called "Message Server" in the R/3 literature. See Footnote 5 for the usage of "Server" and "Service" in this article.

The Life and Adventures of an SAP Transaction

When the term "transaction" is used by programmers, it is often as a synonym for "Logical Unit of Work" (LUW), an indivisible grouping of activities that either succeed completely or are all summarily disregarded (rolled back).[7] When R/3 users call upon this term, they are generally referring to someone running a transaction (identified by a transaction code) in an application system. Since "LUW" nicely (and unambiguously) identifies the former concept, I will stick with this nomenclature when referring to a Logical Unit of Work. I will use the term "transaction" to denote the latter concept, the more colloquial one, the one that is more familiar to R/3 users. In this context, a transaction consists of multiple Dialog Steps, each processed by a Dialog Service.

Before I move on with this discussion, I want to direct a few words to those of you who are now wondering, "This is an article about *update* transactions, but by your own definition a 'transaction' does not have to access a database at all; it is just an activity in a system that can be called up by entering a transaction code." True, there are lots of transactions in R/3 that will never update the database and hence are not central to this discussion, but we are only concerned with those that do perform database updates.

Processing an Update Transaction Request

A user who has logged on to R/3 is said to have a *session* with R/3. The session exists within a particular Application Server, but is not limited to a specific Dialog Service. The session starts with the logon and ends with the logoff. A user normally runs sequential, multiple transactions in a session. A user can also have multiple independent sessions with the same R/3 system.

[7] The concept of an LUW is usually associated with databases (either all related database updates are successful or any database updates done so far have to be taken back).

Incoming requests from a client are directed to the first available Dialog Service by the Dispatcher. This architecture allows a better utilization of physical resources than if we started a different process or thread for each session, requiring memory to be allocated to each of them, rather than sharing it. The consequences of this approach are as follows:

- Different requests emanating from the same session are not necessarily processed by the same Dialog Service.

- One Dialog Service usually services requests from different sessions in an interlaced fashion.

- Each Dialog Service is an active database (DB) user executing SQL statements on behalf of the application code. The database does not know anything about the different SAP sessions. As far as it is concerned, there is one user (the Dialog Service) that keeps executing SQL statements and needs to manage its database LUWs via Commit and Rollback calls:

  - **Commit** tells the database that all updates since the beginning of the LUW can now be made on the actual database tables.

  - **Rollback** tells the database that the application ran into a problem and wants all updates made since the beginning of the LUW to be discarded. After the rollback, the application starts with a clean slate.

  Both Commit and Rollback mark the beginning of a new LUW.

Is it clear to everyone that Dialog Steps from different sessions cannot share the same database LUW? To do so would mean that they would end up committing or rolling back updates made by a different session, which would destroy the integrity of the database. Hence, the Dialog Service automatically issues a DB Commit at the end of each Dialog Step. This guarantees that changes made by

different sessions can never end up in the same LUW, but adds a new and significant complication:

> One transaction in one session consists of multiple database LUWs (one for each of its Dialog Steps).

This is a problem because the Commit/Rollback process needs to be under the control of the transaction in order to keep the database consistent. If multiple updates must be executed as part of the same LUW (and this is usually the case in business transactions), we cannot make any changes unless we are in the last Dialog Step. Otherwise, our changes in an earlier Dialog Step would be committed to the database due to the automatic DB Commit issued by the Dialog Service, while we do not know yet that we will ever reach the end of our transaction successfully.

Requiring an application to make all DB changes in its last dialog step would solve the problem (and indeed you can build SAP transactions that do precisely that). On the other hand, it would be a severe restriction on the possible design of individual applications. SAP came up with a more flexible remedy: using an asynchronous update process for the actual "application" updates.

The Asynchronous Update Process

The asynchronous update process works as follows:

1. The application never updates the application tables directly, but instead writes information about required updates to a special Update Log table.[8] An entry in that table is called an Update Record. Multiple Update Records can be written in multiple Dialog Steps. It is these Update Records that are committed to the database by the automatic DB Commit at the end of each Dialog Step.

2. R/3 will not act on these Update Records until the application signals that it has successfully completed all its activities and is ready to commit all the changes to the actual application tables. To do this, it executes the COMMIT WORK statement in ABAP.

3. COMMIT WORK causes the Message Service to locate an available Update Service and tell it to start processing all the Update Records for this session that have been written between the start of the SAP LUW and the COMMIT WORK statement. (An SAP LUW starts at the beginning of a session, and after each COMMIT WORK and ROLLBACK WORK.)

4. The Update Service will make all application database changes prescribed in the Update Records for this session in one database LUW. This ensures that all related updates are either completed successfully or not at all (rollback).

5. If the application never issues a COMMIT WORK, because, for example, it fails in a later Dialog Step or the user decides to abort the transaction, the Update Records written by this session since the beginning of the SAP LUW are simply ignored and will be garbage-collected later.

6. If the application wants to discard all previous changes and start over, it issues the ROLLBACK WORK statement.

Figure 4 shows all Update Records in my test system at a particular time. Normally, Update Records are processed as quickly as possible and then deleted, so in a low-activity system most of the time the list contains only Update Records that failed.

Figure 5 shows the various ABAP function modules that will be called by the Update Service to

[8] For information about how to actually do this in an ABAP program (CALL FUNCTION xxx IN UPDATE TASK ...), you should consult the ABAP online documentation.

Figure 4 Displaying All Update Records (Transaction SM13)

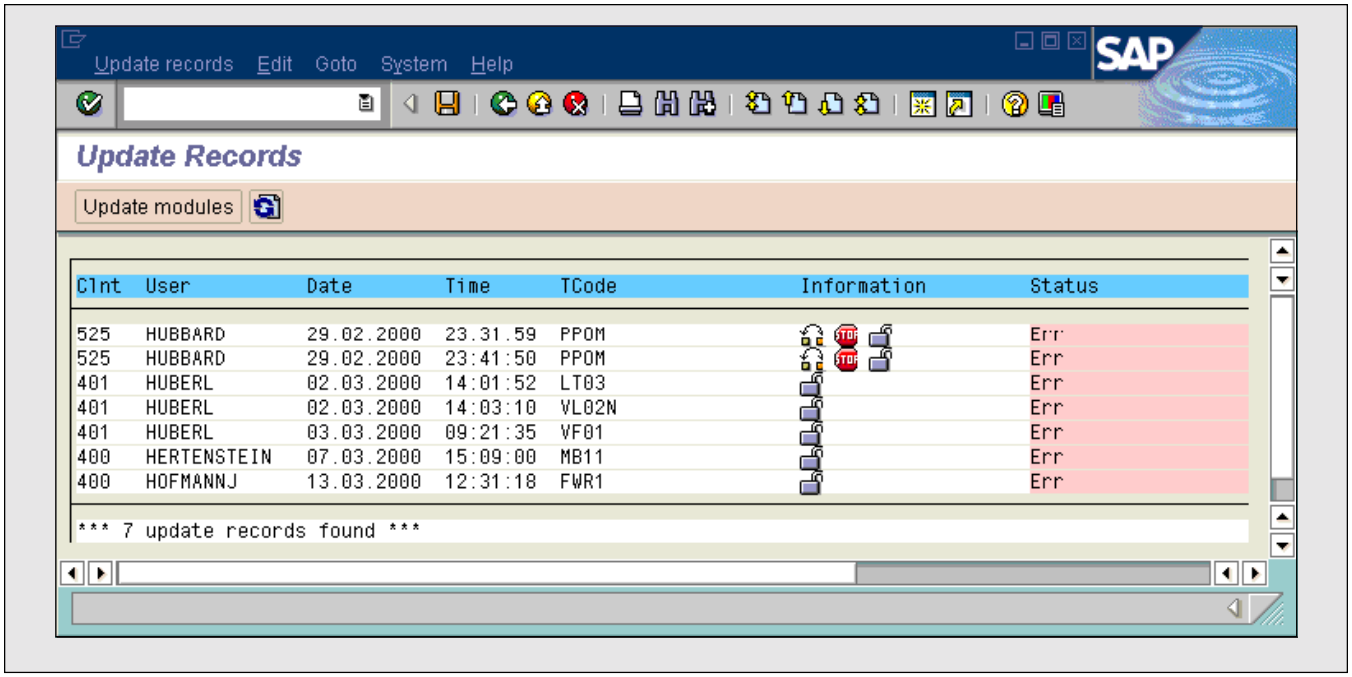
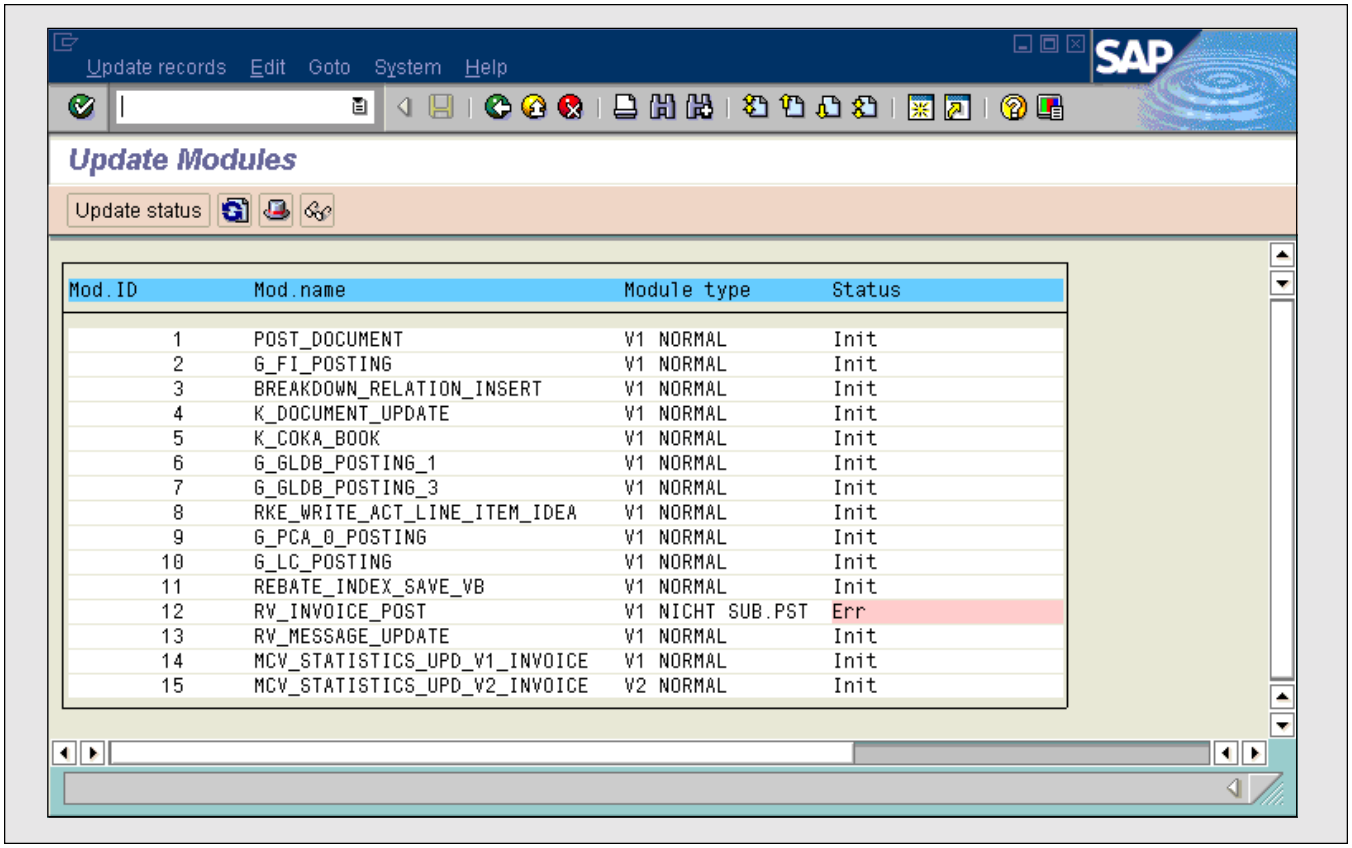


Figure 5 Displaying the Update Modules for One Update Record



make the application changes. Notice the "V1" and "V2" assignments that appear under the "Module type" column. In order to speed up the update process for an individual transaction, SAP differentiates between two types of updates:

- "V1" are all updates that constitute "the meat" of the transaction. All master table and transaction table updates belong to this category, as well as any other updates that are necessary for a consistent database from an operational standpoint. All "V1" updates are executed as part of one database LUW as described above.

- "V2" updates are deemed less critical. SAP maintains redundant rolled-up (accumulated) statistical information that is the basis for many reports in the system. When a user wants a report about product sales per product group per month for a specific country, for example, it could take a long time to read all sales orders to find the required information. Instead, SAP allows for the definition of Information Structures containing accumulated data based on criteria defined by the customer.[9] These Information Structures are automatically maintained by the update modules of the applications.

The application developer can decide which Information Structures must be updated as part of the "V1" processing, and which do not. Those that are not important enough to be constantly updated would be updated later, in a separate step ("V2"). The system administrator can define special "UP2" processes to handle just "V2" updates, otherwise they are also processed by normal Update Services, but only after the process has no more "V1" records to work on. As a consequence, "V2" updates are not part of the same database LUW as the associated "V1" updates, and there is no guarantee that they will ever be executed successfully. ("V2" updates can be used for other non-critical updates than just the Information Structures.)

It is up to the application designer to decide which updates of a transaction need to be part of the same database LUW ("V1"), and where a rare failure does not pose a major threat to the system. An Information Structure that does not reflect one out of a million sales orders, for example, can only lead to marginally different statistics. Also, you can run batch programs to rebuild an Information Structure from scratch if you need absolutely accurate statistics. Usually, all "V2" updates will be successful, of course, but there is no formal guarantee as for "V1" updates.

Asynchronous Update: Friend or Foe?

Ascertaining whether or not the asynchronous update process is a help or hindrance requires us to look at two important consequences of the asynchronous update service:

- Performance
- Update failure

From a performance standpoint, asynchronous updating is definitely a good thing. In a comprehensive business application with a high degree of integration, one relatively harmless-looking update (e.g., a new sales order) may cause updates in

[9] Of course, SAP delivers some predefined Information Structures as well.

many different application areas that all have to be part of one database LUW. If the user had to wait for the completion of all these updates in the last Dialog Step, that would create very uneven response times and thus very unhappy users. (Research has shown that — within limits — it is more important to have even response times than to reduce the average response time.)

Using asynchronous updating, the user can continue to work as soon as the COMMIT WORK has been issued. The only side-effect here is that if the user wants to display the status of, for example, a newly created sales order, immediately after he created it, and the Update Service is still busy making all the required changes, the user would get a message stating that this sales order was still being processed and would he please retry soon. In my estimation, this is not really a problem.

What about failed update processing, though? Since during the updating process code is being executed, there is always the possibility of failure — the ABAP code could be wrong, a table space may be full. In a properly maintained production system, failure is rare (my screen shot in Figure 4 was taken on an SAP training system). But still, we have to understand what happens in the case of a problem. Well, here we go:

- An entry is made into the SAP System Log. This entry documents exactly what went wrong and where. Any system administrator worth his salary will review the System Log at least once a day.

- A warning will be displayed on the Alert Monitor (part of SAP's Computing Center Management System, or CCMS). Any good data center will have someone watching the Alert Monitor at very short intervals for production systems.

- A message is sent to the user who entered the transaction.

The ABAP language also offers a variation of the COMMIT WORK statement, called COMMIT WORK AND WAIT. (ABAP syntax is sometimes funny, isn't it?) If you use that statement, the Dialog Service will wait until the Update Service is finished with all "V1" updating activities and then report back (in a system variable called SY-SUBRC, if you must know) whether the update was successful or not.

> ✔ ***Tip***
> *Before you rush off to liberally apply the COMMIT WORK AND WAIT statement all over your ABAP code, you should consider the performance implications: The Dialog Service will be blocked until the Update Service is done. This can literally take seconds for big transactions. Making lots of calls to COMMIT WORK AND WAIT can single-handedly destroy the performance of an R/3 system. Before using this anywhere, discuss it with your system administrator and consider setting up dedicated Application Servers for these transactions!*

Enqueue Handling in R/3

There is one more facet of SAP's transaction processing architecture that we need to discuss. How can an application actually enqueue or lock something? When we start to change something, it is usually a good idea to have exclusive control over that entity to avoid an inconsistent database or, at the minimum, very unhappy users. SAP uses its own Enqueue Service to control access to entities. An application requests a lock on a certain entity (encapsulated in a Lock Object defined in the SAP Data Dictionary). This request is either granted (the application now holds a lock) or denied (the application has to try later or inform its user that the activity is currently not possible).

Figure 6 Lock Entry List (Transaction SM12)

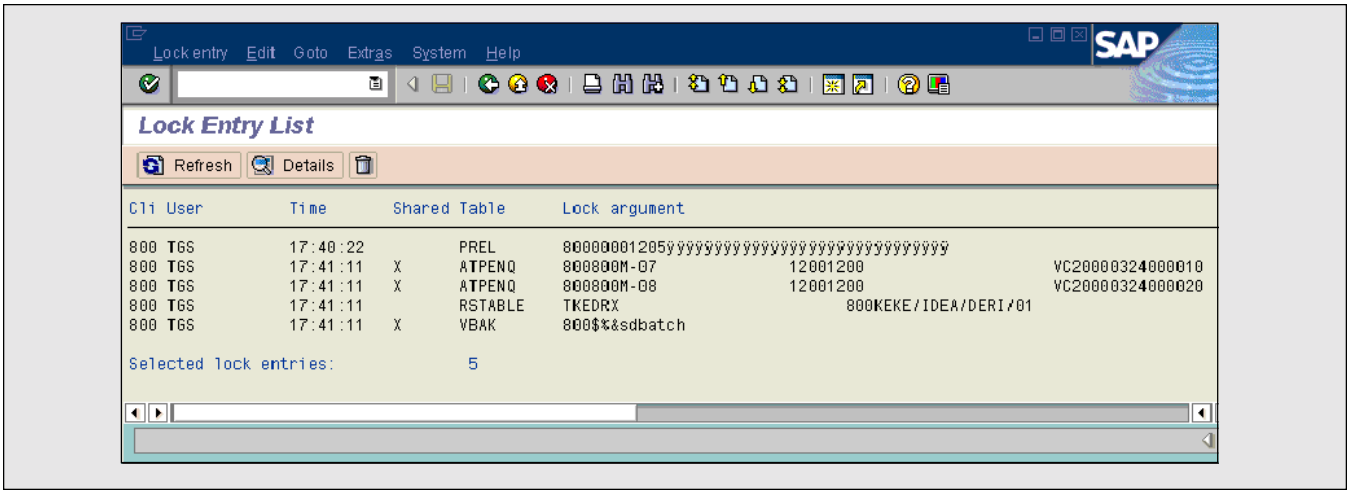


Figure 7 Details of One Lock Entry

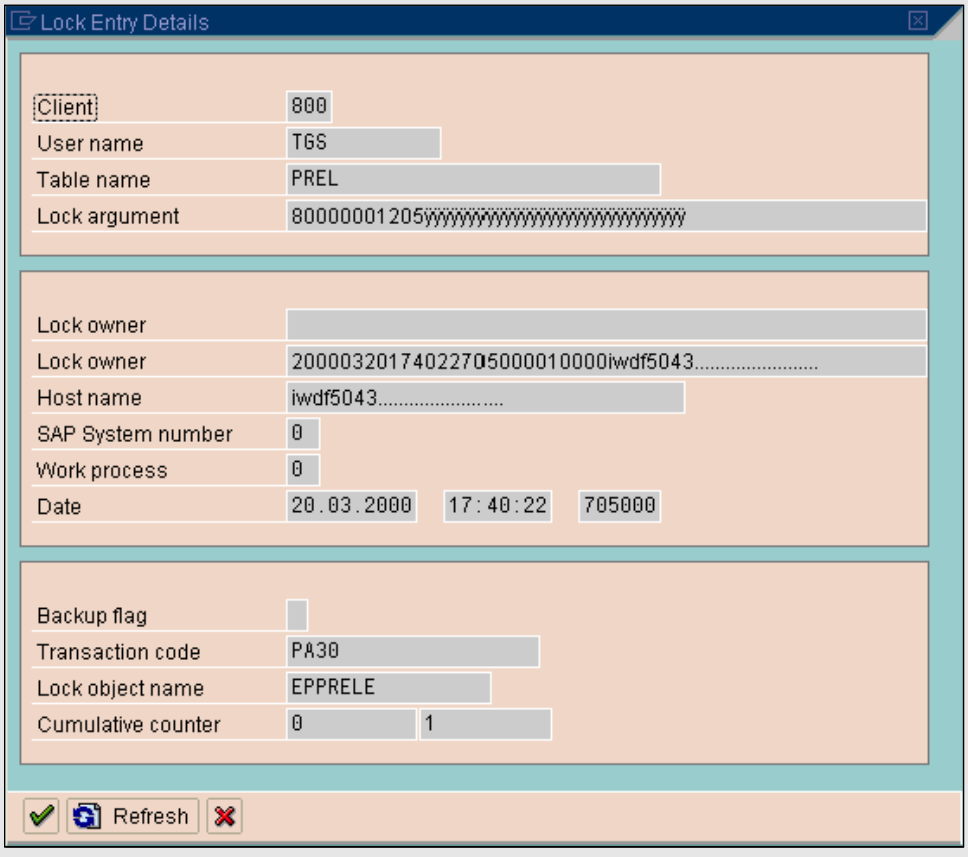


Figure 6 shows a list of all lock entries at a certain point in time, and **Figure 7** offers detailed information about the first lock entry in Figure 6.

Locks can be requested in any Dialog Step. They remain valid until they are released. This usually happens in the asynchronous update portion of a

transaction, after all database updates have been applied successfully or the database updates were rolled back due to a problem. This begs the following question: "What if a user comes into the office in the morning, starts to, let's say, update an employee's address, gets distracted (checking the SAP stock price), goes to lunch, and finally, hours after obtaining the lock, saves the address change, thus unlocking the employee? How can anyone else update this employee during that time?" Here is how this kind of situation is handled: Usually the R/3 system administrator has defined a time-out value that determines how many minutes of inactivity a user is granted before he is thrown off the system (which will release all locks held by that user).

An administrator can also manually intervene. Using transaction SM12, administrators can check who is holding a particular lock, and tell the user to finalize his transaction or leave it. Worst case (for example, if the user holding the lock cannot be found and someone else needs to make a rather important change), the administrator can terminate the user's session and/or manually delete the Lock Entry.[10]

BAPIs and Transactions

So how does all of what you just read apply to BAPIs? Well, BAPI calls are handled by the Dialog Services exactly like Dialog Steps. After each BAPI execution there is an automatic DB Commit. BAPIs can request locks and write to the Update Log, just like any online transaction. This allows us to build applications where we combine multiple updates made by different BAPIs into one LUW, provided the BAPIs do not contain their own COMMIT WORK statement.

[10] Proper authorization is required, of course. Also, read the warning message R/3 displays when you attempt to delete a Lock Entry and make sure that you follow the advice given, otherwise you may destroy the integrity of the database.

Update BAPIs in 3.1

All update BAPIs in Release 3.1 contain their own COMMIT WORK statement. This implies that each update takes place independently of any other update done through any other BAPI call. While this approach is sufficient for some application scenarios, there are many scenarios for which it is not appropriate. Sometimes we have to combine multiple activities into one LUW that should succeed completely or not at all.

Waiting for the Update Service in the Client Program

The fact that a Release 3.1 BAPI issues its own COMMIT WORK (but not COMMIT WORK AND WAIT) has an interesting effect for the client programmer. Let us assume that we have just created a sales order by successfully calling the SalesOrder.CreateFromData BAPI. Immediately after the BAPI returns control to us, we now call SalesOrder.GetStatus for this new sales order. If the asynchronous update process has not finished processing this order yet, our BAPI call will fail. In this kind of situation, you want to code a loop where you

- wait for a specific interval (100 ms, for example), and
- then retry the GetStatus BAPI call until it succeeds or you have reached a certain number of retries (remember: in theory the update process might fail altogether, which would cause an indefinite loop).

Update BAPIs in 4.0 and Later

In Release 4.0, SAP changed the concept for committing BAPI updates. Now BAPIs are *not* supposed to commit their own changes anymore, instead this task is left to the client program. Due to the upward compatibility requirement for the BAPIs, this new concept does not affect the BAPIs introduced in Release 3.1. Unfortunately, it also did not affect some of the new BAPIs introduced in 4.0!

How is this possible? SAP is a nice company to work for. You can break the rules without (in most cases) having to face severe penalties. Therefore, some developers interpreted the new rule to be just a suggestion and ignored it.

So how do we find out whether any given update BAPI does or does not execute its own COMMIT WORK? According to the rules laid down in the BAPI Programming Guide (available on the Open BAPI Network at **http://www.sap.com/products/techno/bapis/edu/docu/45a/prog45ae.doc**), any BAPI added in 4.0 or later that does *not* follow the rule about *not* executing its own COMMIT WORK should say so in its documentation. (After what I just told you about people following rules, you can guess how successful this idea has been.)

In order to help with this situation, SAP has created an OSS (Online Software Services) note (131838) that, in theory, lists all the BAPIs that do their own COMMIT WORK. But, as the note itself says, "Please note that we cannot guarantee the completeness of this note." If you look at **Figure 8**, you will see that that is an understatement: Many BAPIs that are doing their own COMMIT WORK calls are missing.

If a BAPI (added in 4.0 or later) does not explicitly document its approach, the only ways to really know which commit paradigm it adheres to are as follows:

- Read the source code. This may not be a very practical suggestion, because you have to read not only the source code of the function module implementing the BAPI itself, but also the source code of any function called in that function module until you find a COMMIT WORK statement or reach the end of the source code. Given the layered architecture of the application — which is a good idea in principle — this could mean trudging through a lot of source code, indeed.
- Test the individual BAPI without issuing an explicit COMMIT WORK in your client application. If the application tables are changed, then the BAPI obviously commits its own work.
- Finally, some BAPIs let the client program choose whether they should commit or not by offering a parameter (most often called "Nocommit"). **Figure 9** is a list of the BAPIs that I could find in Release 4.6 with that behavior. I cannot guarantee completeness of this list![13]

Figure 8 BAPIs That Are Doing Their Own COMMIT WORK Calls

CTRequest.CreateTasks
CTRequest.Release
InvestmentProgram.SaveValueReplicas
Kanban.SetInProcess
SalesOrder.CreateFromData
SalesOrder.CreateFromDat1
SiteLayoutModule.Change
BapiService.TransactionCommit[11]
RetailMaterial.Clone[12]

Commit and Rollback in a Client Program

Assuming that we are only using well-behaved BAPIs, we now need to discuss how to commit

[11] This is the BAPI used to actually issue a COMMIT WORK statement in a program using BAPIs that do not. See below for details.

[12] According to the note, "This BAPI does not contain a Commit Work, however, it requires a Commit Work command after every call since it writes directly to the database without update." Please read the documentation of this BAPI extremely carefully to understand how it handles commit and why it requires an explicit COMMIT WORK call by the client program.

[13] Actually, two types of BAPIs were omitted on purpose. First, all Human Resources (HR) BAPIs called "Simulatecreation," because they never write anything to the Update Log in the first place. Why they should have a "Nocommit" parameter is beyond my comprehension. Second, all country-specific subclasses (e.g., EmployeePrivatAdrCH) of non-country-specific HR classes (e.g., EmployeePrivAddress) listed. The latter was done to save some space.

Figure 9 BAPIs with Commit Parameters in Release 4.6

CustomerInquiry.CreateFromData
CustomerQuotation.CreateFromData
EmplCommunication.Change
EmplCommunication.Create
EmplCommunication.Createsuccessor
EmplCommunication.Delete
EmplCommunication.Delimit
EmployeeAbsence.Approve
EmployeeAbsence.Change
EmployeeAbsence.Create
EmployeeAbsence.Delete
EmployeeAbsence.Request
EmployeeBankDetail.Approve
EmployeeBankDetail.Change
EmployeeBankDetail.Create
EmployeeBankDetail.Createsuccessor
EmployeeBankDetail.Delete
EmployeeBankDetail.Delimit
EmployeeBankDetail.Request
EmployeeBasicpay.Approve
EmployeeBasicpay.Change
EmployeeBasicpay.Create
EmployeeBasicpay.CreateSuccessor
EmployeeBasicpay.Delete
EmployeeBasicpay.Request
EmployeeFamilyMember.Approve
EmployeeFamilyMember.Change
EmployeeFamilyMember.Create
EmployeeFamilyMember.Createsuccessor
EmployeeFamilyMember.Delete
EmployeeFamilyMember.Delimit
EmployeeFamilyMember.Request
EmployeeIntControl.Approve
EmployeeIntControl.Change
EmployeeIntControl.Create
EmployeeIntControl.Createsuccessor
EmployeeIntControl.Delete
EmployeeIntControl.Delimit
EmployeeIntControl.Request
EmployeePersonalData.Change
EmployeePersonalData.Create
EmployeePersonalData.Createsuccessor
EmployeePersonalData.Delete
EmployeePersonalData.Delimit
EmployeePrivAddress.Approve
EmployeePrivAddress.Change
EmployeePrivAddress.Create
EmployeePrivAddress.Createsuccessor
EmployeePrivAddress.Delete
EmployeePrivAddress.Delimit
EmployeePrivAddress.Request
SalesOrder.CreateFromDat1

Figure 10 Commit and Rollback BAPIs

BAPI Name	Function Module Name
BapiService.TransactionCommit	BAPI_TRANSACTION_COMMIT
BapiService.TransactionRollback	BAPI_TRANSACTION_ROLLBACK

(or rollback) the Update Records written by a client program. Since Release 4.5A, the BapiService object has two BAPIs for this purpose. They are listed in **Figure 10**.

When you call TransactionCommit, all records written to the Update Log by any BAPIs since the last call to TransactionCommit (or TransactionRollback, or the beginning of your program, if you have not previously called TransactionCommit and TransactionRollback) will now be processed by an Update Service. When you call TransactionRollback, all records written to the Update Log by any BAPIs since the last call to TransactionCommit

Listing 1: Calling BAPI_TRANSACTION_COMMIT in Release 4.0

```
Dim oSapFunctions As SapFunctions
  Set oSapFunctions = _
    oSession.CreateInstance("SAP.SapFunctions")
  oSapFunctions.Bapi_Transaction_Commit
```

Figure 11 Parameters of BapiService.TransactionCommit

Parameter Name	Data Type	Optional	Import/Export
Wait	Character 1	Yes	Import
Return	BAPIRET2	Yes	Export

(or TransactionRollback or the beginning of your program, if you have not previously called TransactionCommit and TransactionRollback) will be discarded, and you start with a clean slate.

> ***✔ Tip***
> *If any BAPI call returns an error type of "A" (Abort) in its Return parameter, you should always call TransactionRollback, because something horrible has happened in R/3 and you want to be on the safe side and back out any changes just in case.*

Commit and Rollback in 4.0

In Release 4.0, the BapiService object type does not have the TransactionCommit and TransactionRollback BAPIs. But do not despair, the underlying function modules (BAPI_TRANSACTION_COMMIT and BAPI_TRANSACTION_ROLLBACK) are there.

The source code required to call these function modules will differ depending on the middleware and programming language being used in your project. **Listing 1** shows an example of a call to BAPI_TRANSACTION_COMMIT using Visual Basic and the SAP DCOM Connector.

Wait and See

Release 4.5 is the release in which the commit and rollback functions became "BAPI-fied". There was also new functionality added to the BAPI_TRANSACTION_COMMIT module, and as a consequence also to the BapiService.TransactionCommit BAPI. You can now elect to use the COMMIT WORK AND WAIT command instead of the simple COMMIT WORK. **Figure 11** shows the new parameters of the BAPI.

Setting the Wait parameter to "X" will cause two things to happen:

- The BAPI will execute COMMIT WORK AND WAIT. It will wait for the Update Service to complete before returning control to the client program.

- The Return structure will be filled in, so that we can find out whether the Update Service succeeded or not.

This is clearly an improvement, because now we can be absolutely sure that the asynchronous update process has succeeded. Remember my warning about the dire performance consequences, though. Do not use this without approval from your system administrator.

Do Not Mix and Match

Once you have figured out which update paradigm is implemented by the BAPIs you want to use in a particular application, you should be very careful not to mix update BAPIs of different types in the same SAP LUW. The COMMIT WORK statement commits all Update Log records for a session. Let us assume that you want to do three updates within one SAP LUW. Let us further assume that the first and third BAPIs called follow the new rules (i.e., do not contain their own COMMIT WORK), but that the second BAPI was developed for Release 3.1 and therefore does call COMMIT WORK. Once you call this second BAPI, any changes made by the first and second BAPIs will be committed. The third BAPI call will be executed in a separate LUW. There is no way to prevent an old-style BAPI from calling COMMIT WORK, so the only solution is not to combine new-style and old-style update BAPIs.

Transactions in the SAP DCOM Connector

Anything said so far applies to BAPI programming in general, regardless of the programming language and middleware used. The developers of the SAP DCOM Connector (SDC) did a nice job to support transaction handling directly in the proxy classes generated by SDC for the business objects in R/3. Each proxy class has the following methods:

- Sub CommitWork()
- Sub CommitWorkAndWait()
- Sub RollbackWork()

CommitWork and RollbackWork call function modules containing COMMIT WORK and ROLLBACK WORK statements, respectively.

CommitWorkAndWait is not implemented yet in the current release of SDC (4.6B). When it is implemented, SAP will need to extend its signature to include the Return parameter discussed earlier.

If you are using Release 4.0, you can choose between using either the function modules BAPI_TRANSACTION_COMMIT and BAPI_TRANSACTION_ROLLBACK or the SDC methods CommitWork and RollbackWork.

If you are on Release 4.5 or later, you should use the BapiService BAPIs TransactionCommit and TransactionRollback so that you can benefit from the new Wait parameter, where that is appropriate.

Conclusion

The architecture of R/3 is the basis for the scalability and performance of the system. Using an asynchronous update mechanism guarantees more even response times, but requires a certain programming style for updating transactions.

Release 4.0 has considerably extended the usefulness of the BAPI concept by allowing us to combine multiple update BAPIs into one LUW. You can take advantage of this in your BAPI-enabled applications. Should you write BAPIs yourself, you should definitely use the normal asynchronous update paradigm for your database changes and not make any COMMIT WORK calls, in order to let your BAPIs participate in extended LUWs.

Addendum

In the November/December 1999 issue of this journal, I published an article called "Simplifying BAPI Programming with Components". This article contained generic information, but also introduced the beta version of a product called the "ARAsoft Java BAPI Object Factory". Based on feedback from readers and from students of my Java BAPI programming classes, I was since able to add many interesting capabilities to this product and simplify the use of it. The first official release is now available. To receive the evaluation copy of the Object Factory, just send me an e-mail. If you are using Java for your BAPI projects, this product will save you significant amounts of time.

Thomas G. Schuessler is the founder of ARAsoft (www.arasoft.de), a company offering products, consulting, custom development, and training to a worldwide base of customers. The company specializes in integration between SAP and non-SAP components and applications. ARAsoft offers various products for BAPI-enabled programs on the Windows and Java platforms. These products facilitate the development of desktop and Internet applications that communicate with R/3. Thomas is the author of SAP's CA925 "Programming with BAPIs in Visual Basic" class, which he teaches in Germany and in English-speaking countries. His book on the same subject, "The BAPI Bible for SAP Programmers: The Comprehensive Guide to Integrating SAP Products with Web, Desktop, and Mobile Applications Using Java, Visual Basic, and ABAP", will be published soon by the SAP Professional Journal. Thomas is a regularly featured speaker at SAP TechEd and SAPPHIRE conferences. Prior to founding ARAsoft in 1993, he worked with SAP AG and SAP America for seven years. Thomas can be contacted at thomas.schuessler@sap.com or at tgs@arasoft.de.

