Web services or RFCs — choosing the right technology for your SAP integration challenges

by Dr. Willi Nüßer

Dr. Willi Nüßer
Professor, University of Applied Sciences, Paderborn

Prof. Dr. Willi Nüßer holds the Heinz Nixdorf foundation chair for applied computer science at the University of Applied Sciences (FHDW) in Paderborn, Germany. His research and consulting interests include Web services in general and in the SAP environment. Before joining FHDW, he worked for six years at SAP headquarters in Walldorf, Germany, where he was responsible for several projects in the SAP LinuxLab. He is the co-author of "SAP Interface Programming" and "SAP on Linux – Architecture, Implementation, Configuration, Administration."

When I ask SAP customers their opinion of Web services technology and its impact on their IT landscapes (and on the business world as a whole), I usually receive a wide spectrum of replies. Their perspective ranges from "everything will eventually be a Web service" to "it's all just more useless hype." In fact, many people are unsure exactly how to think about Web services. Are they a delivered solution? A development approach? An infrastructure design concept?

I propose a more pragmatic view, which I think is now taking hold in the IT community: viewing Web services as a middleware technology and a viable alternative to established middleware technologies such as SAP remote function calls (RFCs), remote procedure calls (RPCs), Microsoft Component Object Model (COM), CORBA, and so on. But when should you select a Web service approach over the others — RFC in particular? This question is of special importance in an SAP environment because RFC is a well-established and trusted approach to integrating different applications, having played a central role in so many projects over the last 12 years. (Indeed, RFC remains a cornerstone of customer systems today.) In contrast, support for Web services in SAP is relatively new. It began with a very basic Simple Object Access Protocol (SOAP) implementation only five years ago in SAP R/3 Release 4.6D, and many customers still have little or no experience with it.

This article bridges the gap by providing a foundation for choosing the appropriate approach for your particular situation and SAP environment. First, I create an example that demonstrates the basic process of accessing an ABAP-based Web service from a Java client located somewhere on the Web. This example mirrors a typical Web services scenario of integrating different technology platforms. It serves as a tutorial for developing Java-based clients for SAP Web services, especially for readers who are less familiar with this approach. I then examine the differences between Web

Different approaches to implementing Web services

The Web services journey in this publication began with an article by Arthur Wirthensohn that appeared in the July/August 2005 issue ("Extend the Internal and External Reach of Your Applications with ABAP-Based Web Services"). That article showed you how to build an ABAP-based Web service in SAP Web Application Server (SAP Web AS) 6.20 and 6.40 using tools provided with SAP Web AS, such as the ABAP Workbench. Both the Web service provider (the server) and the Web service requester (the client, known as the "consumer" in official Web services terminology) were implemented in ABAP. This is a natural starting point, since every RFC-enabled function module (RFM) in SAP Web AS 6.20 and higher is exposed as a Web service. Thus, SAP Web AS out-of-the-box is a provider of many powerful Web services.

This article complements the previous article by presenting a typical outside-in approach. Here, SAP Web AS 6.40 provides the Web service and the consumer is implemented on the Java platform. It is exactly in these types of heterogeneous technology situations where Web services come into play and where the decision to use Web services or RFC is most important due to differences in dealing with various clients. In pure ABAP-to-ABAP communications, RFC is still the preferred choice in most situations.

services and RFC to help you assess the role that each technology should play in your SAP environment. Based on the presented example, I compare and contrast the two approaches in terms of performance and transactional behavior. I conclude with some deployment recommendations that I have found to be quite helpful for deciding for or against a particular approach in real-world situations.

However, let's begin with a quick review of key RFC and Web services concepts that are important for understanding the rest of the article.

Understanding key RFC and Web services concepts

Starting in SAP R/3 Release 4.6D, SAP began complementing the existing RFC infrastructure with Web services technology. Think of both the SAP-proprietary RFC middleware and the standardized Web services as different approaches to implementing distributed systems. In this section, I examine each of these technologies in more detail in order to establish a foundation for understanding and developing the example later in the article.

What are RFCs and RPCs?

SAP remote function calls (RFCs) are an adaptation of the well-known remote procedure calls (RPCs) to the SAP environment. The RPC concept is based on the idea of calling functions that reside on other computers in a similar way to calling local functions. Since this approach has a long history in computer science going back about 30 years, the benefits and drawbacks of RPC are well known. The following aspects are usually mentioned:

- RPC provides a simple model to extend an application into the network. Since RPC hides the complexity of network communications (that is, creating network messages, establishing network connections, and so on) from the application, it is still in widespread use.
- However, there is no generally accepted scientific or industrial standard for how RPCs should behave. Usually only remote functions within the same middleware communicate seamlessly.
- Therefore, integrating different RPC frameworks — for example, the classic Unix/Sun RPC and Microsoft COM+/Distributed COM (DCOM) — is a non-trivial task that requires

additional development efforts in order to achieve interoperability.

- Many RPC frameworks only support synchronous calls in which both the caller and callee must be active simultaneously. This restriction introduces a tight coupling between applications, which is not suitable in some integration scenarios due to the need for detailed knowledge about the partner that may not be available. For example, consider a situation in which the types of clients that address an SAP system are not predetermined, which is the standard in Internet traffic. (See the "Inspecting Coupling" sidebar below for more information.)
- RPC is based on the procedural programming paradigm. Although well-known, this model is not suitable for problems where the natural way of thinking does not deal with functions but instead with objects or even services. These entities are more coarse-grained (they comprise more functions) and therefore better suited to analysis and design that starts from business processes rather than from functions in computer programs.

With RFC, SAP extended the established RPC model in several ways. First, starting with the 2.x releases, the SAP application server constitutes a powerful RFC middleware framework that includes the usual runtime support for executing RFC-based applications plus a fully integrated development environment. Second, this environment accommodates transactional behavior — a feature that is lacking in most traditional RPC implementations. Transactional support is embedded directly into the ABAP stack, thereby enabling the creation of safe transactional RFC applications. In addition, these applications can be coded in ABAP, C, Visual Basic, Java, and so on. Integrated support for many different programming languages is the third SAP extension of RPC. SAP supports many different languages out of the box, while other RPC frameworks usually concentrate on only a few languages. The fourth aspect addresses the communication mode between the caller and callee. Queued RFC[1] makes asynchronous RFC calls possible, which reduces coupling between applications and leads to more flexible solutions. For example, the SAP RFC middleware implemented in the 3.x and higher releases of the SAP application server permits the temporal decoupling of applications.

[1] Queued RFC is an extension of RFC that enables the SAP runtime to guarantee the retention and one-time execution of several RFMs.

Inspecting coupling

I introduced the term *coupling* when characterizing the RPC model. This notion is important to any discussion of the pros and cons of Web services and is covered in detail in many publications.* Simply put, coupling means that making changes to one communication partner (the application, let's say) also requires making appropriate changes to the other partners (in other words, the clients or peers).

Several forms of coupling exist. The type of coupling relies upon the types of changes to one partner that *really* require changes to the other partner. For example, consider a typical Internet application such as a Web shop. Here changes to the internal code of the Web shop application must not require changes at the client side. Internet solutions must minimize dependencies between the participating partners because the nature of the requesting clients is not known beforehand. In contrast, relocating the Web shop almost inevitably leads to changes at the client side (that is, at least changing the URL).

Going a bit further, we can distinguish between several types of changes:

* For more information about coupling and its relationship to current ways of implementing applications, refer to *Enterprise SOA: Service-Oriented Architecture Best Practices*, by Dirk Krafzig, Karl Banke, and Dirk Slama (Prentice Hall PTR, 2004), Chapter 3.5.

Continues on next page

Continued from previous page

- The first and most obvious type of change comprises changes to the implementation of an application, including simple code changes. Today almost all applications try to hide these types of changes completely. The object- or component-oriented approach is based on this concept of encapsulation and decoupling. However, switching the programming language that is used in the application is not hidden in all cases. In some middleware solutions (including RPC), this type of change also requires adaptations to the partner code.
- The next type of change applies to the interface that the application provides to its partners. These changes typically affect the name of a method or its number and types of parameters. Modifications to the interface of a software module obviously have consequences for the calling modules. In all current middleware solutions, this form of coupling still largely exists.
- A change in the address (that is, the URL) of one partner is the next type. Possible reactions range from doing nothing (which is possible when the partners find each other dynamically, such as via some broker mechanism), to simply changing the URL, to abandoning the middleware solution altogether. Relocating an in-house solution to an address outside the enterprise firewall is one example of the last, rather dramatic case. Here a simple CORBA solution may run into serious trouble when the firewall blocks the CORBA communication protocols GIOP/IIOP.**
- The last type of change deals with communication style. Consider what happens to a solution when one of the partners shuts down. If the other partners call this partner synchronously (in other words, by waiting for the response), they may block or even abort the method call or the entire application. Therefore, calling the partner asynchronously is considered to provide a looser coupling. This approach is beneficial when different institutions administer the partner applications, which may lead to different up and down times.

Creating loosely coupled applications (that is, applications that are insensitive to some or most of the types of changes mentioned above) is a major target when operating in a distributed environment such as the Internet. A loose coupling facilitates the reuse of software components and usually results in flexible software architectures. RFC and Web services offer different forms of coupling, as I explain in the "What are RFCs and RPCs?" and "What are (Web) services?" sections.

** General Inter-Orb Protocol/Internet Inter-Orb Protocol.

What are (Web) services?

Both RPC and RFC extend the procedural programming model in a simple way to distributed applications. This model, however, became increasingly problematic when dealing with big and complex software systems such as those that realize business processes. Object-oriented (OO) or component-oriented approaches are superior to the procedural model in terms of modularization, maintenance, and flexibility. Therefore, the OO paradigm began to dominate software development. In a certain respect, services are simply the latest incarnation of objects and components. A service is often defined as "a self-contained software unit that is described, provided, located, and used in a network-transparent manner." You could think of a service as a large object that lives somewhere on the network and holds everything needed to fulfill a task. If you emphasize the network aspect, the service is referred to as a *Web* service to reflect that the service is accessed via a network connection.

Combining different services into a complete solution is the mission of a Service-Oriented Architecture (SOA). For example, the SAP Enterprise Services Architecture (ESA) is an implementation of a generic SOA with some SAP-specific features added. All SOAs aim to realize the following objectives:

- A service provides a reusable and complete unit of business logic that can be incorporated into a complete application. The last requirement suggests that a service combines more application logic than a common object. A service has much more coarse granularity, meaning that it usually consists of multiple objects.
- The descriptions of the service interfaces should be independent of the service implementation in order to accommodate different types of clients. To be more precise, the language of the interface description (usually Web Services Description Language, or WSDL) does not relate directly to the language of the service implementation.
- The communication protocols should emphasize interoperability and location transparency in order to enable direct communication between partner applications. For example, when using SOAP, whether the accessed service is written in ABAP, Java, or C# becomes irrelevant to a user. When using a Universal Description, Discovery, and Integration (UDDI)[2] directory, this transparency may extend to an independence of the actual location (that is, the URL) of the service. (See the "Inspecting heterogeneity" sidebar on the next page for more information.)

To summarize the implications of these features, an SOA centers on three main concepts: a *coarse granularity* to enable reasonably sized entities, a *loose coupling* to remove unnecessary dependencies between services, and a strong emphasis on the integration and interoperability of *heterogeneous* systems.

Assuming that the chosen SOA implementation meets these requirements, the following guidelines for using an SOA are generally accepted as best practices:

- Use a service-oriented approach to work with entities that are better suited for modeling business objects and processes, such as invoice processing.
- Use a service-oriented approach when dealing with heterogeneous technology situations, such as applications from different vendors.
- Use a service-oriented approach when loose coupling between partner applications is necessary, such as when integrating different programming languages or runtime environments.

At least from a conceptual point of view, SOA and Web services — both in general and in SAP — appear to offer significant advantages over RPC/RFC in some areas. But can we be certain that these recommendations are really adequate or even true in an SAP environment? In reality, Web services implementations may deviate from these theoretical characteristics for better or worse. Therefore, in the rest of this article, I concentrate on the behavior of current Web services implementations in SAP Web AS 6.40 and compare these solutions to the above recommendations. This approach will lead to more concrete guidelines for using Web services in the SAP world.

Overview of the example application

Now that you understand the origin and basic nature of RFC and Web services, let's create a simple example to help us evaluate and compare their capabilities in the SAP environment. I chose to use an outside-in situation for this test case. The Web services provider is SAP Web AS 6.40 and the Web services consumer is a Java-based client that resides somewhere on the Web. This configuration models a typical heterogeneous integration scenario. The business logic resides in ABAP code in SAP Web AS and needs to be accessed from other applications, perhaps a different enterprise resource planning (ERP) or customer relationship management (CRM) solution.

In addition to creating a Web service, we also create an RFC client using SAP Java Connector (JCo) that accesses the same business logic in order to

[2] For more information on UDDI, check the UDDI area at www.oasis-open.org.

Inspecting heterogeneity

Heterogeneity exists in IT systems on several different levels. The first level is certainly hardware. From an application perspective, this aspect is relatively trivial today since most modern operating systems hide differences in hardware architecture and configuration. In contrast, integrating operating systems from different vendors is definitely more complicated and not completely hidden from users. However, quite a few software approaches (for example, Samba*) allow for the coexistence and cooperation of, say, Windows-based and Unix-based systems.

The problem with heterogeneity builds up further at the next level, which is the middleware. Bridging two different middleware solutions usually requires writing some adapter code in both partner applications or even completely replacing one of the solutions. Here is specifically where Web services offer the promise to combine and integrate different technologies.

On still higher levels such as applications and data, heterogeneity is commonly considered inevitable. Web services make no attempt to harmonize applications or the data that the applications share. Instead, Web services simply provide the means to allow communication between different applications. The data format can and should be negotiated between the communicating partners. SOAP simply transports them all. In this sense, Web services are in fact neutral to the data format.

* Samba is an open source software suite that runs on various Unix platforms and is installed on a host server. Microsoft Windows clients and servers can then interact with the host server, which acts as a Windows file and print server. For more information, refer to www.samba.org.

illustrate the different programming models and lay the foundation for comparing Web services and RFC.

Figure 1 shows the system landscape for development and testing. It consists of three ABAP-based function modules running on SAP Web AS that are accessible using either a Web service or a native RFC Java client, as shown on the left side in Figure 1. Notice that Web service and RFC clients use different communication protocols for this purpose (SOAP and RFC, respectively).

System prerequisites

If you want to reproduce this example, you need access to a server that is running SAP Web AS 6.40 in order to create the Web service provider. You could code the Java client that accesses the Web service in an integrated environment such as SAP NetWeaver Developer Studio. However, in this article, I chose to use a toolchain[3] that is used in a wide range of Web service projects because of its power, stability, and widespread usage. This framework for implementing Web services is based on the open source Apache Axis project,[4] which is one of the most stable and accepted Web services implementations currently available. I use this approach to outline the basic steps in creating Java-based Web service clients.

[3] A toolchain is a collection of tools such as development workbenches, compilers, debuggers, and libraries that aid the development of applications. For more information, see http://en.wikipedia.org/wiki/toolchain.

[4] Downloads and documentation are available at http://ws.apache.org/axis.

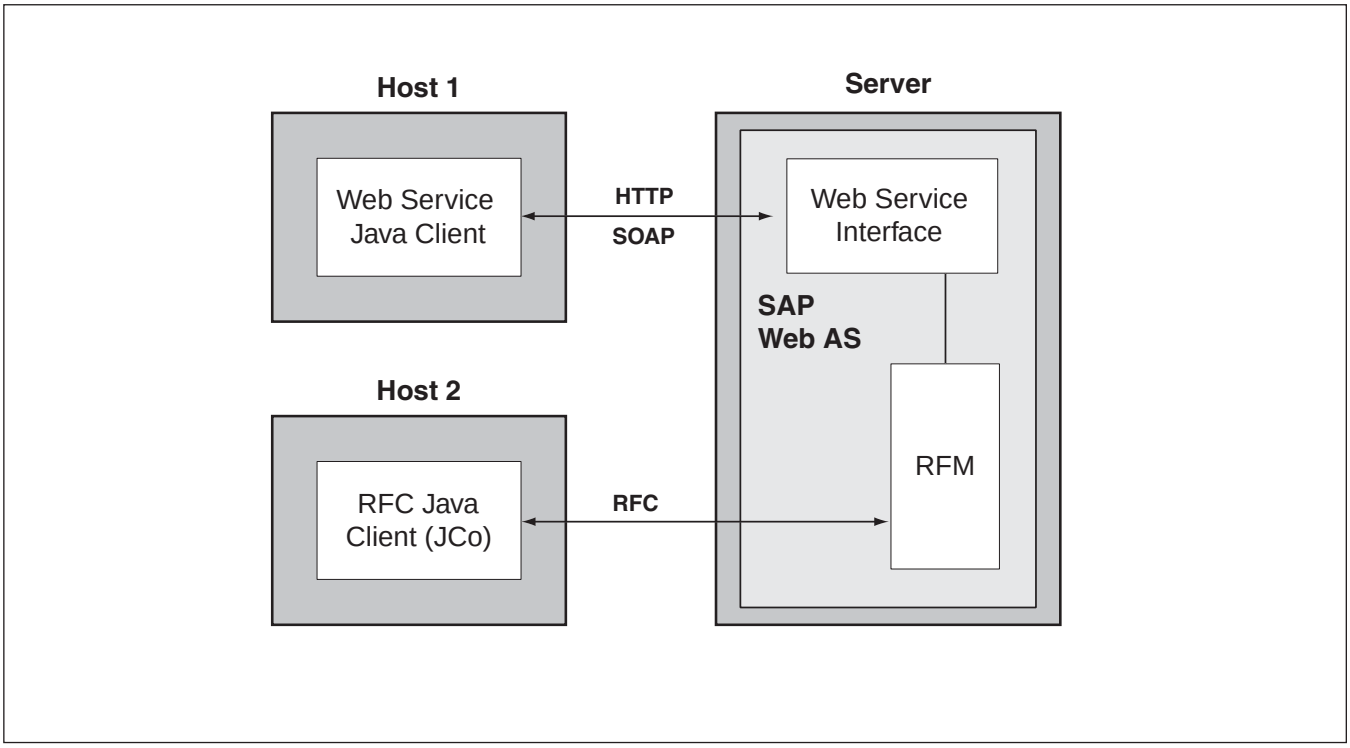

Figure 1 Architecture of the example application

Creating the ABAP-based Web service

Almost every non-trivial RFM and consequently every Web service that is used in a production application accesses at least one database table in the SAP system. Therefore, this behavior is a logical basis for our example Web service. To implement the example, we create a clone of the familiar customer table KNA1 that comes with all SAP systems. Our cloned table is named ZKNA1DEMO and is filled with several test records. This article assumes that you are familiar with the basic ABAP tasks of developing tables and their associated function modules. For simplicity, this section focuses on the tasks that are specific to implementing a Web service — in other words, the ABAP code for implementing the RFMs.

In addition to a database table, our example uses the following ABAP function modules to implement the business logic:

- **Z_GET_CUSTOMER** selects all records from the table ZKNA1DEMO. We also use this module later in the article when comparing the performance of Web services and RFC.
- **Z_INSERT_CUSTOMER** inserts a new customer record directly into the table ZKNA1DEMO, while **Z_INSERT_UPD** uses the familiar SAP table update mechanism. The purpose of these modules is to demonstrate the transactional capabilities of SAP Web AS 6.40.

After creating these RFMs, we will create a single Web service interface for all three modules. Both the Web service and SAP JCo clients that we create next are able to access all of these function modules.

Implementing the function modules

Let's begin our example by creating the RFMs that implement the business logic. The first function

```
FUNCTION Z_GET_CUSTOMER.
*"----------------------------------------------------------------------
*" Local interface:
*"  TABLES
*"      ET_CUSTOMER_DATA STRUCTURE  ZKNA1DEMO
*"      ET_RETURN STRUCTURE  BAPIRETURN1 OPTIONAL
*"----------------------------------------------------------------------

select *
  from zkna1demo
   into corresponding fields of table et_customer_data.

ENDFUNCTION.
```

Figure 2 The Z_GET_CUSTOMER remote function module

module Z_GET_CUSTOMER (**Figure 2**) performs a database query and has a very simple structure.

Both of the other two function modules perform a database insert, although they differ in how they access the table. Z_INSERT_CUSTOMER (**Figure 3**) inserts the transferred customer data directly into the table ZKNA1DEMO. The if clause (starting at the highlighted line in the middle of Figure 3) handles any possible errors. It calls the BAPI return function module BALW_BAPIRETURN_GET2 to create an appropriate error message Z_WN_MSG.

The immediate insert that is implemented in Z_INSERT_CUSTOMER is well suited for small or standalone changes. However, you often need to combine several changes into what is referred to as a logical unit of work (LUW) in order to guarantee the overall consistency of several related database changes. The well-known LUW concept in SAP buffers the associated changes and writes them to the database in a single atomic transaction. The function module Z_INSERT_CUSTOMER does not reflect this approach, instead writing each change to the database immediately without performing any buffering.

In order to take advantage of the SAP update mechanism and demonstrate the LUW concept, we create another function module Z_INSERT_UPD (see **Figure 4** on page 78). Instead of performing the update directly, Z_INSERT_UPD simply calls an update function module named Z_UPDATE that is not RFC-enabled.[5] Z_UPDATE does exactly the same thing as Z_INSERT_CUSTOMER (refer back to Figure 3), performing a direct insert operation into the table ZKNA1DEMO. By implementing the table insert with this approach, calling Z_INSERT_UPD does not cause an immediate update to the database. Instead, the SAP system buffers all of the changes that are made in the Z_UPDATE module so they can be written to the database using the update work process that is provided by SAP Web AS.

Creating the Web service

Once the remote function modules have been developed, the next step is to create the Web service that wraps the RFMs. Here I assume that you are familiar with the basic process and focus on the aspects that pertain specifically to this example. (See the "Recap — implementing a Web service in SAP Web AS 6.40" sidebar on page 79 for more information.)

[5] For more information about update function modules, refer to the online documentation on your SAP system for the function module facility (transaction SE37).


```
FUNCTION Z_INSERT_CUSTOMER.
*"----------------------------------------------------------------------
*" Local interface:
*"  IMPORTING
*"     VALUE(IS_ZKNA1DEMO) LIKE  ZKNA1DEMO STRUCTURE  ZKNA1DEMO
*"  EXPORTING
*"     VALUE(ES_RETURN) LIKE  BAPIRET2 STRUCTURE  BAPIRET2
*"----------------------------------------------------------------------

data: par1 like sy-msgv1.
data: par2 like sy-msgv2.

insert into zkna1demo values is_zkna1demo.

if sy-subrc <> 0.
  write is_zkna1demo-kunnr to par1.
  write is_zkna1demo-name1 to par2.

  call function 'BALW_BAPIRETURN_GET2'
    exporting
      type = 'E'
      cl = 'Z_WN_MSG'
      number = '001'
      par1 = par1
      par2 = par2
    importing
      return = es_return
    exceptions
      others = 1.
endif.

ENDFUNCTION
```

Figure 3 The Z_INSERT_CUSTOMER remote function module

The first step in creating the Web service is to define a virtual interface in the Object Navigator (transaction SE80) by selecting Enterprise Services and then Virtual Interfaces. Our virtual interface is named Z_VI_SPJ to indicate its type, where Z_ is the usual prefix for customer objects, VI is our notation for a virtual interface, and SPJ is the unique name of the interface. In our example, the main purpose of this task is to aggregate all of the RFMs into one virtual interface. In more complex scenarios, creating a virtual interface also helps with defining views to the underlying RFMs such as by renaming parameters.

In addition to the three RFMs created earlier (`Z_GET_CUSTOMER`, `Z_INSERT_CUSTOMER`, and `Z_INSERT_UPD`), we include two SAP-provided RFMs (`BAPI_TRANSACTION_COMMIT` and `BAPI_TRANSACTION_ROLLBACK`)[6] in order to support

[6] The well-known `BAPIService.TransactionCommit()` and `BAPIService.TransactionRollback()` rely on these RFMs.

```
FUNCTION Z_INSERT_UPD.
*"----------------------------------------------------------------------
*" Local interface:
*"  IMPORTING
*"     VALUE(IS_ZKNA1DEMO) LIKE  ZKNA1DEMO STRUCTURE  ZKNA1DEMO
*"  EXPORTING
*"     VALUE(ES_RETURN) LIKE  BAPIRET2 STRUCTURE  BAPIRET2
*"----------------------------------------------------------------------

data: par1 like sy-msgv1.
data: par2 like sy-msgv2.

call function 'Z_UPDATE' in update task
  exporting
    IS_ZKNA1DEMO = is_zkna1demo.

...

ENDFUNCTION.
```

Figure 4 The Z_INSERT_UPD remote function module

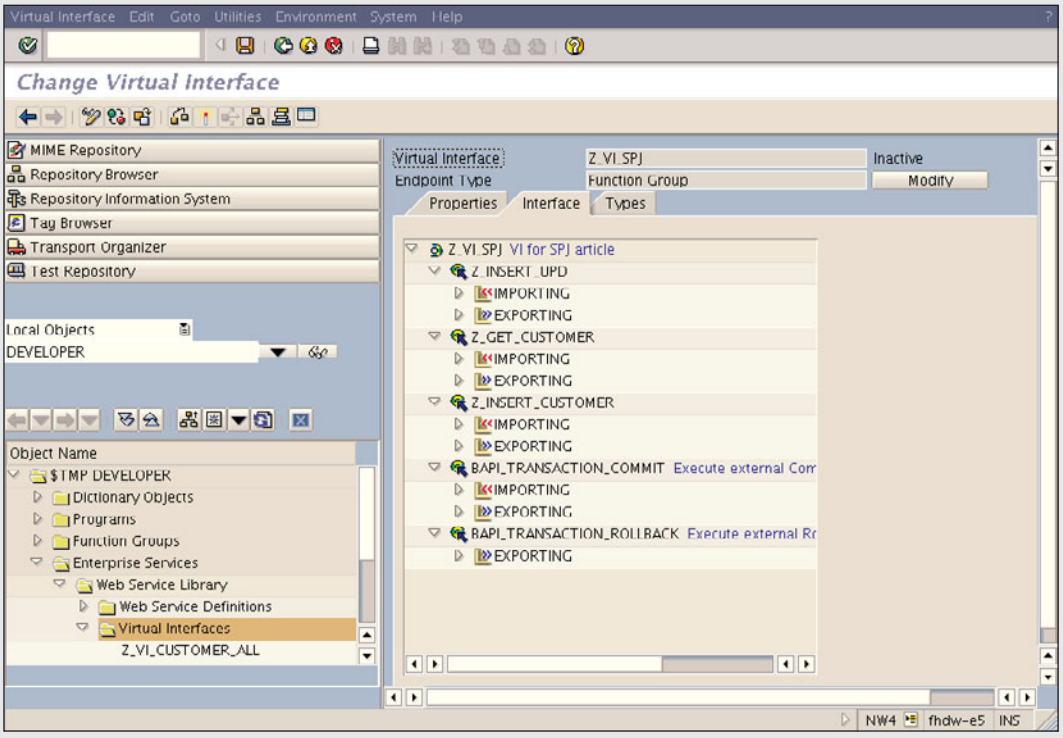

Figure 5 Defining the virtual interface Z_VI_SPJ

Recap — implementing a Web service in SAP Web AS 6.40

The article "Extend the Internal and External Reach of Your Applications with ABAP-Based Web Services" that appeared in the July/August 2005 issue of this publication described the process of creating a Web service in SAP Web AS 6.20 and 6.40 in detail. This article assumes that you have read the previous article and are already familiar with the process and its concepts. The following checklist is a reminder of the key tasks for SAP Web AS 6.40, which is the focus of this article:

1. Run the wizard provided in SAP Web AS 6.40 to create the components of the Web service, including the virtual interface, its associated RFM, and the Web service definition. Either run the WS_WZD_START transaction (Wizard for Creating and Deploying WS) or navigate to the wizard in the Utilities menu path in the Function Builder (transaction SE37).
2. In the ABAP Editor, review and if necessary change any of the automatically assigned parameters for the generated virtual interface.
3. Run the transaction WSADMIN (Web Service Administration for SOAP Runtime) to create the Web Service Description as a WSDL document based on the Web service definition that was automatically generated in Step 1.
4. Save the generated Web Service Description to a file on your local computer for use in implementing the Web service client.
5. Optionally, publish the Web service to a UDDI registry of your choice, either via Web service definition maintenance in the Object Navigator (transaction SE80) or via the WSADMIN transaction using a UDDI client.

Dual meanings for "WSD"

Remember that the WSD acronym refers to both Web Service Description (which refers to the WSD standard in the W3C Web Services Architecture Working Note) and Web service definition (which refers to the WSD that you create in SAP Web AS).

the LUW concept. I explain the purpose of these specialized RFMs for controlling transactions in more detail shortly. **Figure 5** shows the configured interface Z_VI_SPJ before it is initially saved.

The next step in creating the Web service is to build a Web service definition, which in this example is named Z_WS_SPJ. The starting point for this task is selecting Enterprise Services and then Web Service Definition in the Object Navigator. The Web service definition defines the central features of the Web service such as session handling and authentication mechanism. To define the methods that constitute and are exposed by the Web service, we refer to the methods named in the virtual interface Z_VI_SPJ that we just created. The Web service Z_WS_SPJ is composed of five RFMs and therefore offers five methods to Web service consumers such as the example clients that we create in the following sections.

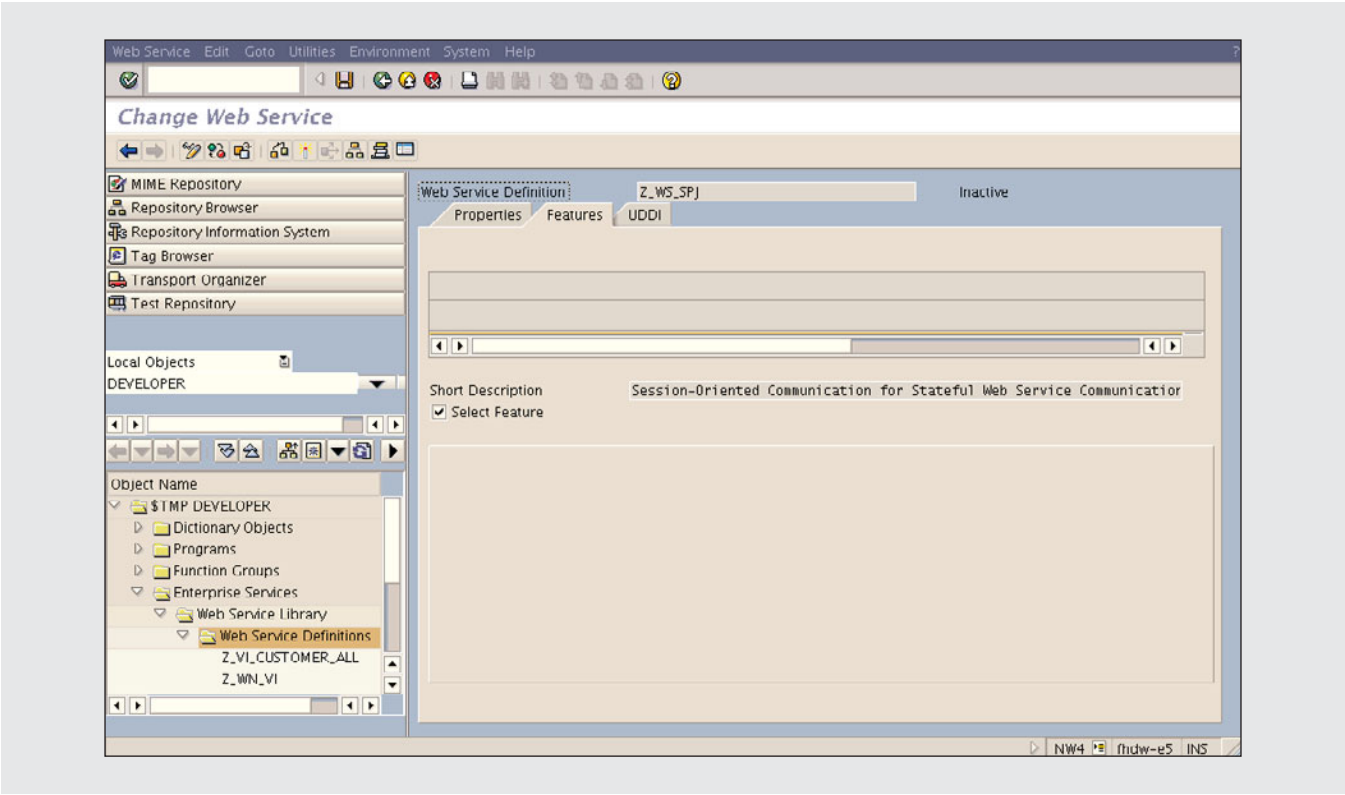

Figure 6 Defining stateful communication for the Web service definition Z_WS_SPJ

You also configure additional features such as security and transactional support for the Web service definition. **Figure 6** shows an important aspect of this process in SAP Web AS 6.40. Checking the Select Feature option enables stateful communication, which means in particular that several calls to the Web service can work on the same data and therefore act as a logical unit of work. This setting is required in order to support the transactional behavior that I discuss later in the article.

Now let's turn our attention to implementing the Java clients that access this Web service.

Note!

Support for transactional behavior is only available in SAP Web AS 6.40 and later. You must explicitly activate this setting for the Web service definition as shown in Figure 6. Because Web services in SAP Web AS 6.20 cannot hold state information, transactional support cannot be modeled.

Building the Java clients

You can build Web service clients using many different languages. In an SAP environment, the most important languages are Java and Microsoft .NET. This article focuses on the Java platform because most customer SAP projects today are based on the Java language. However, the general process of developing a client for a given Web service is independent of the chosen language. The first step in all language environments is obtaining access to the external Web service definition for the Web service.

This definition is based on the XML-based Web Services Description Language (WSDL).[7]

Poking around in the WSDL

Starting with SAP Web AS 6.40, obtaining the WSDL of a Web service is quite easy. As described in the earlier article on creating a Web service, the transaction WSADMIN provides a quick and easy path to the WSDL. At this point, let's assume that you have created a WSDL for the Web service — say, by pressing Ctrl-F1 in the transaction WSADMIN — and that you correctly invoked the browser when requesting the WSDL in this transaction.[8] Before continuing, save this document to your local file system so we can examine the WSDL more closely.

Inspecting the WSDL is normally not necessary. You would usually skip this step and simply leave the task of obtaining the WSDL to the development tools. Because these tools automatically handle all of the relevant actions, you do not need to do anything with the WSDL. However, let's take a brief look at the WSDL in order to gain a better understanding of the information upon which the Web service clients rely. **Figure 7** and **Figure 8** (on page 82) show some

[7] For more information about WSDL, refer to www.w3.org/TR/wsdl.

[8] On Linux-based systems, invoking the browser correctly may require changing the settings in the variables NETSCAPE and NETSCAPE_PATH. Refer to the documentation for the transaction FILE for more information.

```
<?xml version="1.0" encoding="utf-8"?>
   <wsdl:definitions
         targetNamespace="urn:sap com:document:sap:rfc:functions"
         xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
         xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
         xmlns:tns="urn:sap-com:document:sap:rfc:functions"
         xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
         xmlns:xsd="http://www/w3/org/2001/XMLSchema">

   ...

   <wsdl:portType name="Z_WS_SPJ">

      <wsdl:operation name="Z_INSERT_UPD">
         <wsdl:input message="tns:Z_INSERT_UPD" />
         <wsdl:output message="tns:Z_INSERT_UPDResponse" />
      </wsdl:operation>

      <wsdl:operation name="Z_GET_CUSTOMER">
         <wsdl:input message="tns:Z_GET_CUSTOMER" />
         <wsdl:output message="tns:Z_GET_CUSTOMERResponse" />
      </wsdl:operation>

      ...
```

Figure 7 WSDL excerpt for the Web service Z_WS_SPJ, part one *Continues on next page*

***Figure 7** continued*

```
  </wsdl:portType>

   <wsdl:binding name="Z_WS_SPJSoapBinding" type="tns:Z_WS_SPJ">
      <soap:binding
         style="document"
         transport="http://schemas.xmlsoap.org/soap/http" />

      <wsdl:operation name="Z_INSERT_UPD">
         <soap:operation soapAction="" />
         <wsdl:input><soap:body use="literal" /></wsdl:input>
         <wsdl:output><soap:body use="literal" /></wsdl:output>
      </wsdl:operation>

      <wsdl:operation name="Z_GET_CUSTOMER">
         <soap:operation soapAction="" />
         <wsdl:input><soap:body use="literal" /></wsdl:input>
         <wsdl:output><soap:body use="literal" /></wsdl:output>
      </wsdl:operation>

      ...

   </wsdl:binding>

...
```

```
...

   <wsdl:service name="Z_WS_SPJService">
      <wsdl:port
         name="Z_WS_SPJSoapBinding"
         binding="tns:Z_WS_SPJSoapBinding">
      <soap:address
         location="http://host:port/sap/bc/srt/rfc/sap/Z_WS_SPJ?sap-
                   client=000" />
      </wsdl:port>
  </wsdl:service>


</wsdl:definitions>
```

Figure 8 WSDL excerpt for the Web service Z_WS_SPJ, part two

important snippets of the XML code of the WSDL for our example Web service Z_WS_SPJ. These portions of the code clarify the relationship between the Web service components that we created in the previous section and the Java clients that we create in this section. They illustrate key Web service properties such as location, independence, and so on.

The first snippet in Figure 7 identifies the Web service (`<wsdl:portType name="Z_WS_SPJ">`) and shows two of the five methods that it exposes (`Z_INSERT_UPD` and `Z_GET_CUSTOMER`). These methods work by exchanging XML-coded parameters, which are defined in the second snippet in Figure 7. The structure of these XML messages is partly defined by the values for the attribute `style` in the element `<soap:binding>` and the attribute `use` in the element `<soap:body>`. In current Web service implementations, these attributes can have several possible values.[9] Future developments of the SOAP and WSDL standards will focus on the `style="document"` and `use="literal"` settings because this combination eliminates ambiguities that exist with other settings and therefore offers the best interoperability overall. In my experience, using this combination of settings creates Web services that are capable of integrating very heterogeneous systems and takes advantages of the greatest strength of Web services. I recommend that you always choose these settings when creating the WSDL from the Web service definition option in the transaction WSADMIN. Other combinations can result in severe problems such as inaccessible data fields when, for example, connecting C# and Java-based implementations.

Figure 7 also illustrates the aspect of coupling between systems. Since most platforms support XML, defining the Web service interface with XML messages (for example, in tags such as `<wsdl:input message="tns:Z_INSERT_UPD" />`) makes the interaction of the partners largely independent of the implementation and the programming platform. However, changing the Web service interface still leads to changes in the WSDL and consequently to the partner code. In this respect, Web services and RFC seem to be on an equal footing. We'll revisit this topic at the end of the article.

In our simple example, which uses the same RFMs for both Web services and native RFC calls, there is no significant difference in communication style. Both approaches support the synchronous mode very well. The WSDL in Figure 7 only implicitly shows this synchronous support by defining both an input message (see the tags that begin with `wsdl:input message`) and an output message (see the tags that begin with `wsdl:output message`). In many Web service implementations (including SAP at the moment), support for asynchronous calls is still in its infancy. Therefore, current Web service implementations may still experience problems with partner responses that do not immediately follow the request. In this respect, native RFC is still in a better position because of the excellent support of transactional and queued RFC.

The last of the coupling types applies to the location (that is, the URL) of the partners. Although an SOA is allegedly loosely coupled (in other words, independent of the location of the service), the code snippet in Figure 8 shows that the URL of the Web service is defined explicitly in the WSDL (see the `location` element). This approach is similar to the client-side RFM specification in RFC via the configuration file saprfc.ini. Because of this explicit URL, a true loose coupling with respect to the service location is not possible using WSDL alone. Here is another area where Web services concepts promise more than the current implementations actually deliver. More complex tools such as UDDI and related Web services

Note!

In SAP Web AS 6.20 and 6.40, the `style` element can have a value of `rpc` or `document`. For best interoperability (especially with Microsoft-based systems), choose the `document` style. The `rpc` style is provided mostly for historical reasons and has no real advantages over `document`.

[9] For more details about attribute values, refer to the WSDL specification at www.w3.org/TR/wsdl.

```
import ws_spj.*;

import org.apache.axis.client.*;
import org.apache.axis.*;

public class SPJ_WS_TestClient_640 {

   ws_spj.Z_WS_SPJSoapBindingStub stub;

   SPJ_WS_TestClient_640 () {     /* constructor */
      try {
         stub = (ws_spj.Z_WS_SPJSoapBindingStub)
               new ws_spj.Z_WS_SPJServiceLocator().getZ_WS_SPJSoapBinding();

         stub.setUsername("developer");
         stub.setPassword("******");
         stub.setMaintainSession(true);
      } catch (Exception ex) {
         System.out.println("Error in Constructor");
         ex.printStackTrace();
   }
    }
   ...
```

Figure 9 Initializing the Java client for accessing the Web service Z_WS_SPJ

standards are necessary to achieve a higher level of location independence.

Developing the Java Web service client

As I mentioned earlier, you do not need to examine the WSDL document in order to create a Java client. Almost all Web service frameworks provide tools for creating Java code from a WSDL document. The Apache Axis framework that I used to create this example comes with the WSDL-to-Java (WSDL2Java) tool. It reads the WSDL document and creates all of the relevant Java files for communicating with the ABAP Web service. You call this tool at the operating system level with a command similar to the following:

```
java org.apache.axis.wsdl.WSDL2Java
-W -p ws_spj Z_WS_SPJ.wsdl
```

The name of the WSDL2Java program is org.apach.axis.wsdl.WSDL2Java. The option -W instructs the tool to create Java code that is best suited for the SAP environment. The option -p is included for convenience. It takes one argument (in this example, ws_spj), which is the name of the subdirectory in which to create the Java classes. The final argument (in this example, Z_WS_SPJ.wsdl) is the name of the WSDL document.

This tool generates the Java code for all of the necessary data structures (such as the method parameters) and the code that facilitates calling the Web service. You simply call the relevant methods in the Java client code. **Figure 9** shows the code for

initializing the Java client to access the example Web service that we created. First the basic Java packages are imported. Then the constructor allocates a stub (in this example, `ws_spj.Z_WS_SPJSoapBindingStub`) that gives access to all of the methods provided by the Web service Z_WS_SPJ and hides the actions of establishing an HTTP connection to the server. This stub also provides some standard methods (`setUsername` and `SetPassword`) for setting the authentication token name and password when logging in to the SAP system. In SAP Web AS 6.40, the target client in the SAP system is already encoded in the URL in the WSDL (refer back to Figure 8).[10]

[10] In SAP Web AS 6.20, you must explicitly specify the target client in the client code as well. The capabilities for building client-side applications are limited in SAP Web AS 6.20 and the implementation specifics depend upon the toolchain that you are using. This topic is therefore beyond the scope of this article.

Finally, the `stub.setMaintainSession(true)` statement sets the frame for stateful communication by creating something like a session context. This call is required for transactional behavior, which is covered in the next section.

Once the client is initialized, you call the methods provided by the Web service in the Java client code as shown in **Figure 10**. The Java methods in this example (`get_customer_data()`, `insert_customer`, and `insert_upd`) are prototypes for accessing the methods of a Web service. Their basic structure is the same. First they build the list of input parameters, which is named `params` in all three methods. For the `get_customer_data` method, this parameter does not contain any values because this method simply requests all entries in the ZKNA1DEMO table. The code for the method `insert_customer` exemplifies the case where input parameters are present. In our

```
...
public void get_customer_data() {
   try {
      ws_spj._Z_GET_CUSTOMER params = new _Z_GET_CUSTOMER();

      /* calling the Web service method ZGetCustomer */
      ws_spj._Z_GET_CUSTOMERResponse resp = stub.z_GET_CUSTOMER( params );

      /* printing the table rows */
      ws_spj.Zkna1Demo[] table = resp.getEtCustomerData().getItem();
      for (int i = 0; i < table.length; i ++) {
         System.out.println(table[i].getKunnr().getValue() + "     " +
            table[i].getName1().getValue());
      }
   } catch (Exception ex) {
      ex.printStackTrace();
}
}

public void insert_customer () {
   try {
      ws_spj._Z_INSERT_CUSTOMER params = new _Z_INSERT_CUSTOMER();
```

Figure 10 Main body of the Java Web service client code

Continues on next page

Figure 10 *continued*

```
         /* preparing the input parameter isZkna1Demo */
         ws_spj.Zkna1Demo isZkna1Demo = new Zkna1Demo();
         isZkna1Demo.setMandt( new Clnt3("000"));
         isZkna1Demo.setKunnr( new Char10("      1234"));
         isZkna1Demo.setName1( new Char35("Gandalf the Grey"));
         params.setIsZkna1Demo(isZkna1Demo);

         /* calling the Web service method Z_INSERT_CUSTOMER */
         ws_spj._Z_INSERT_CUSTOMERResponse resp =
            stub.z_INSERT_CUSTOMER( params );

      } catch (Exception ex) {
         ex.printStackTrace();
   }

   }

   public void insert_upd () {
      try {
         ws_spj._Z_INSERT_UPD params = new _Z_INSERT_UPD ();
         ws_spj.Zkna1Demo isZkna1Demo = new Zkna1Demo();
         isZkna1Demo.setMandt( new Clnt3("000"));
         isZkna1Demo.setKunnr( new Char10("      1235"));
         isZkna1Demo.setName1( new Char35("Aragorn"));

         params.setIsZkna1Demo(isZkna1Demo);
         ws_spj._Z_INSERT_UPDResponse resp = stub.z_INSERT_UPD(params);
      } catch (Exception ex) {
         ex.printStackTrace();
      }
      }

   ...

   public static void main (String[] args) {

   SPJ_WS_TestClient_640 app = new SPJ_WS_TestClient_640();

   System.out.println("First Select");
   app.get_customer_data();

   ...

    }
}
```


example, the single input parameter consists of the database record is `Zkna1Demo`. After setting up the parameter — in this case with one of the mightier wizards, Gandalf the Grey — the Web service is called via the `stub.z_INSERT_CUSTOMER(params)` statement. The result of the call (if present) is accessed via the variable `resp`. For the method `get_customer_data`, this variable holds all of the table rows. Finally, the purpose of the `main` method is to call the methods `get_customer_data()`, `insert_customer`, and `insert_upd`. For space reasons, this example only shows the call to `get_customer_data()`.

Although these code snippets might seem complicated, remember that the WSDL2Java tool handles most of the development for you. Once you have written your first Java Web service client, writing parts of your own code becomes almost automatic because the basic structure of the code is essentially the same for all Web service clients. Obviously the ability to reuse large chunks of code really shortens the development cycle. This aspect may become still more prominent when you consider dynamically constructing Web service calls using advanced standards such as the Web Services Invocation Framework (WSIF), which takes care of much of the setup work. Implementations are already available.[11]

The final step in creating a Web service client consists of the usual compilation of the Java code. You can either perform this step manually or use an integrated development environment such as SAP NetWeaver Developer Studio.[12]

Developing the RFC client using SAP JCo

To gain a better understanding of the similarities and differences between the Web service and RFC approaches, we now turn our attention to developing an RFC client. This part of our example illustrates some structural aspects of the client code and is also used in the comparison tests later in the article. **Figure 11** shows the code for an example Java client that uses native RFC via the SAP JCo interface.

[11] For more information about WSIF, go to http://ws.apache.org/wsif.

[12] For a detailed introduction to SAP NetWeaver Developer Studio, see the article "Get Started Developing, Debugging, and Deploying Custom J2EE Applications Quickly and Easily with SAP NetWeaver Developer Studio" in the May/June 2004 issue of *SAP Professional Journal*.

```
import com.sap.mw.jco.*;

public class JCO_transact {

   private JCO.Client mconnection;
   private IRepository mrepo;

   private final String RID = "SAPIF";
   private final String connfile= "connection.properties";

   public SPJ_JCO () {

      /* Connect infos */
      Properties connprops = new Properties();
      try {
         FileInputStream pf = new FileInputStream(connfile);
         connprops.load(pf);
      } catch (Exception ex) {
         ex.printStackTrace();
```

Figure 11 Code snippets for the SAP JCo client

Continues on next page

Figure 11 *continued*

```
            return;
         }
         try {
            mconnection = JCO.createClient(connprops);
            mrepo = JCO.createRepository(RID, mverbindung);
            mconnection.connect();
         } catch (Exception exc) {
            exc.printStackTrace();
         }
      }

      private int do_get() {
         IFunctionTemplate mift = null;
         JCO.Function mf2 = null;

         try {
            mift = mrepo.getFunctionTemplate("Z_GET_CUSTOMER");
         } catch (Exception exc) {
            exc.printStackTrace();
         }
         try {
            mf2 = new JCO.Function(mift);
            mconnection.execute(mf2);

            JCO.Table mexptab =
               mf2.getTableParameterList().getTable("ET_CUSTOMER_DATA");

            /* Creating Output Header line */
            System.out.println("KUNNO " + "NAME " + "Country");

            for (int i = 0; i < mexptab.getNumRows(); i++) {
               mexptab.setRow(i);
               System.out.println(mexptab.getString("KUNNR") + " " +
                                  mexptab.getString("NAME1") + " " +
                                  mexptab.getString("LAND1"));
            }
         } catch (Exception exc) {
            exc.printStackTrace();
         }
         return(0);
      }

      ...

      public  static void main(String[] args) {

         SPJ_JCO sapif = new SPJ_JCO();
         System.out.println("First GET");
         sapif.do_get();
         ...
      }
   }
```


Because the code presented here reflects the usual SAP JCo approach, this section focuses on the key points that are relevant to our comparison of Web services and RFC.[13] Most of the code-level differences between the Java and RFC clients are associated with different syntactical conventions. For example, you use a stub to connect to the Web service in the Web service client, while you use a connection object in the SAP JCo client.

One major difference with the RFC client is that you do not explicitly specify the location of the RFM in the client code. Instead, you specify the location in an external property file named connection.properties that resides on the client. This file specifies the parameters that are necessary to establish a network connection to the SAP system. A typical entry might look like:

```
jco.client.ashost=/H/<host>/S/<port of
gateway>
```

The name jco.client.ashost indicates the location of the application server to which the client will connect. The /H/ option specifies the host typically via an IP address and the /S/ option indicates the target ports. This approach is similar to specifying the URL in the WSDL file. Therefore, Web services and RFC are currently quite comparable in terms of coupling to a specific location.

Another RFC client difference is that all SAP JCo calls use the same connection named `mconnection` (see the highlighted code in Figure 11). SAP JCo therefore has a built-in way of identifying related calls and combining them into a transaction, which means that you can easily achieve transactional behavior. Despite some naming and calling convention differences, the rest of the code is fundamentally not very different from the code for the Web service client. The constructor `SPJ_JCO()` sets up the connection to the SAP system. For example, the method `do_get()` uses this connection to execute a specific RFM — in this code snippet, the query module `Z_GET_CUSTOMER` provided by the Web service. The remaining code in the method `do_get()` handles the output of the table lines. The `main()` method then calls the do_`get()` method, exactly as in the `main` method in the Web service client.

As with the Web service client, the final task in developing the RFC client is to compile the Java code. Then we can move on to examining the runtime behavior of both clients in order to compare some of their features.

Putting the examples to work

Now that we have built a Web service and two different Java clients to use as examples, we turn our attention to answering the question that I posed at the beginning of the article. Choosing whether to use Web services or RFC in a given situation requires some understanding of how the two different solutions function. We begin our comparison with a qualitative look at the performance of the Web service and RFC clients in order to obtain practical information that is fundamental to this decision. Then we add the RFMs that perform table updates to the Web service and RFC clients in order to demonstrate the transactional capabilities of the two approaches. This addition introduces a new level of complexity to the example. Two Web service calls now depend on each other for correct execution, which may require additional efforts in order to achieve logical consistency.

Evaluating Web service and RFC performance

The best way of examining the performance of Web services is to use the simple query method Z_GET_CUSTOMER because of its simplicity and the option to change the size of the returned answer easily. For demonstration purposes, I added time-stamping code (*not* the notoriously inexact method

[13] SAP JCo is a middleware component that is the standard for accessing RFC when you use the Java programming language. This section assumes that you are familiar with how to use this interface. Refer to *SAP Interface Programming* by J. Meiners and W. Nüßer (SAP PRESS, 2004) or go to the SAP Library at http://help.sap.com for more information. See also the *SAP Professional Journal* articles "Repositories in the SAP Java Connector (JCo)" (March/April 2003), "Server Programming with the SAP Java Connector (JCo)" (September/October 2003), and "Tips and Tricks for SAP Java Connector (JCo) Client Programming" (January/February 2004).

`System.currentTimeMillis`)[14] directly before and after the call to the RFC modules in the Web service and RFC clients, prepared identical operating environments for each client, and ran both clients several hundred times. The timing code recorded the length of time one call to the Z_GET_CUSTOMER method takes in the Web service client or the SAP JCo client respectively. I repeated this test several times using increasingly larger tables measured in terms of number of rows. **Figure 12** shows the relative performance overhead of Web service calls over native RFC/JCo calls. The term *runtime ratio* is defined as the time one call to Z_GET_CUSTOMER takes in the Web service client divided by the time one call to the same RFM takes via native RFC. For example, the first row in the table in Figure 12 shows that selecting 10 records of the ZKNA1DEMO table via the Web service client takes 10.33 times longer than the same action in the SAP JCo client.

Number of records in the ZKNA1DEMO table	Runtime ratio of Web service to RFC call
10	10.33
100	9.84
1000	10.82
10000	12.94

Figure 12 Comparison of Web service and RFC performance

Obviously running these tests on different machines will produce slightly different values. However, in my experience the performance of native RFC client calls is generally about 10 times better than that of Web services in the SAP environment. This conclusion reflects other performance measurements for non-SAP Web service implementations.[15]

Furthermore, the test results in Figure 12 seem to indicate that the performance overhead of the Web service client compared to the native RFC client becomes even worse with an increasing payload (that is, the number of table rows returned). This fact is well-known and by no means SAP-specific. Processing XML messages is best for transporting small to medium-sized data packets because XML processing takes time. When designing a Web service interface today, be careful that the size of the call parameters does not extend a few hundred kilobytes. You can transport larger amounts of data as attachments to SOAP messages. However, you need to decide how to handle the attachments because there is currently no standardized approach.[16] This option may also lead to reduced application interoperability because some Web service frameworks do not support all attachment formats.

So what does all this data really mean for your integration project? Remember that all current Web service implementations — including SAP's own — exhibit significantly reduced performance. And the problem becomes worse as the size of the transmitted data packages increases. If performance is a critical issue, carefully consider the full set of pros and cons before choosing Web services.

Comparing transactional capabilities in Web services and RFC

Many middleware systems support the important concept of a transaction, which simply refers to the ability to group actions into a logical unit in order to ensure overall consistency. In SAP, the logical unit of work and the update mechanism constitute a powerful implementation of transaction support. This mechanism allows you to combine several actions such as calls to function modules or RFMs into an atomic unit. Changes made in one call become visible in the database only after all of the calls have been confirmed (in other words, "committed"). In particular, if you read from a table that was changed before all changes were committed, you should get back the

[14] The `System.currentTimeMillis` method that is provided by the Java platform does not yield exact timestamps because the method execution is left to the Java virtual machine.

[15] As a starting point for a deeper study of Web services performance, visit www-128.ibm.com/developerworks/webservices.

[16] One possible approach is based on Multipurpose Internet Mail Extensions (MIME); another is based on the W3C SOAP MTOM specification. For more information, see www.w3.org/TR/soap12-mtom/.

original unchanged values. This valuable behavior prevents inadvertent access to unconfirmed and possibly erroneous data.

The SAP environment has long offered full transaction support for ABAP and RFC clients that access remote function modules in the same SAP system. As you will soon see, SAP Web AS 6.40 supports this feature for Web service clients as well. However, Web service frameworks that involve multiple applications and systems face much more complex situations. For example, suppose the changes that are made by a Web service in one dataset such as a CRM system should become visible only when the changes made by another Web service in the basic ERP system are successful, perhaps in a transaction that updates several customer-related tables. Since the typical Web service environment does not have a central coordinator, aggregating the two calls is not an easy task. Currently the solution to this problem exists only in specifications and approaches, without any real implementation.[17] In my opinion, all SOAs — including SAP ESA — must be able to address this problem in order to succeed in typical business-to-business (B2B) projects.

In an SAP environment, developing transaction-safe Web service clients for SAP systems is a comparatively easy task. Stateful Web services are fully supported starting with SAP Web AS 6.40. Remember that we enabled this feature for our example Web service back in Figure 6 by checking the Select Feature option in the Web service definition. Now we also need to add some code to the Web service client in order to support the transaction concept. **Figure 13** shows the enhanced code for the Web service client that we created back in Figure 9 and Figure 10. The added method `commit()` calls the standard RFM `BAPI_TRANSACTION_COMMIT`, which

[17] For more information about the evolving specifications for coordinating Web service transactions across systems, go to www.oasis-open.org.

```
...
public void commit() {
   try {
      ws_spj._BAPI_TRANSACTION_COMMIT params =
            new _BAPI_TRANSACTION_COMMIT();
      params.setWait(new Char1("Y"));
      ws_spj._BAPI_TRANSACTION_COMMITResponse resp =
            stub.transactionCommit(params);
   } catch (Exception ex) {
      ex.printStackTrace();
   }
}
...
public static void main (String[] args) {

   SPJ_WS_TestClient_640 app = new SPJ_WS_TestClient_640();
   System.out.println("First Select");
   app.get_customer_data();
   System.out.println("Inserting: Z_INSERT_CUSTOMER");
   app.insert_customer();
   System.out.println("Second Select");
   app.get_customer_data();

   System.out.println("Inserting using Update: Z_INSERT_UPD");
```

Figure 13 Adding transaction support to the Web service client

Continues on next page

Figure 13 *continued*

```
      app.insert_upd();
      System.out.println("Third Select: before Commit");
      app.get_customer_data();        // King not there
      System.out.println("Now Committing");
      app.commit();
      System.out.println("Fourth Select: after Commit");
      app.get_customer_data();        // Now, we have a king
   }

}
```

```
First Select
0         Orc0
1         Orc1

Inserting: Z_INSERT_CUSTOMER

Second Select
0         Orc0
1         Orc1
1234      Gandalf the Grey

Inserting using Update: Z_INSERT_UPD

Third Select: before Commit
0         Orc0
1         Orc1
1234      Gandalf the Grey

Now Committing

Fourth Select: after Commit
0         Orc0
1         Orc1
1234      Gandalf the Grey
1235      Aragorn
```

Figure 14 Web service client output for a stateful Web service

works exactly like the familiar ABAP statement `COMMIT WORK`. It closes a logical unit of work and triggers the SAP update service to write the changes to the database permanently. Notice that the `main()` method calls the `commit()` method in order to commit the changes to the database after calling the `Z_INSERT_UPD` module to make a database insert via the update module `Z_UPDATE`. (See the highlighted line near the end of the code.)

When working with a stateful Web service, the call sequence that is defined in the `main()` method in Figure 13 produces the output shown in **Figure 14**. Use the `println` statements in the `main()` method to

match the output with the corresponding position in the code. Let's assume that the table ZKNA1DEMO is initially filled with two nasty customers, Orc0 and Orc1. Directly inserting one mighty wizard (here, Gandalf the Grey) via `Z_INSERT_CUSTOMER` causes the value to immediately appear in the table without any further action. However, this behavior is undesirable in situations where operations must be combined into a logical unit (such as when you are updating related tables) because it may lead to inconsistencies. Thus, inserting a king (here, Aragorn) into the table using the SAP update mechanism (that is, via the combination of `Z_INSERT_UPD` and the `commit()` method) causes the value to become visible in the database only after the final `commit` statement is executed. This approach allows the aggregation of several actions to be written to the database in a single atomic action.

The result is slightly different for Web services in SAP Web AS 6.20 or for stateless Web services that are created in SAP Web AS 6.40 (that is, Web services that do not have the Select Feature option checked as shown back in Figure 6). The first select, inserting, and second select operations produce exactly the same output as shown in Figure 14. However, the `commit()` call does not produce an additional row with the inserted king (Aragorn). When working in a stateless mode, SAP Web AS cannot associate the `Z_INSERT_UPD` call and the following `commit()` call because it does not have anything like a context that relates to both calls.

Turning briefly to RFC for a moment, as I mentioned earlier native RFC provides many well-known and powerful ways to create similar transaction-safe call sequences. Transactional RFC (tRFC) and queued RFC (qRFC) are the most important options.[18] Because these concepts are widely documented elsewhere, for simplicity I focus here primarily on implementing transactional capabilities in Web services to provide a basis for comparison.

In summary, in an SAP environment Web services and native RFC are on equal footing in terms of providing transactional capabilities. Other Web service frameworks such as Axis require significantly more effort to implement the same feature.

[18] For more information about tRFC and qRFC, refer to *SAP Interface Programming* by J. Meiners and W. Nüßer (SAP PRESS, 2004).

The final analysis — Web services vs. RFCs

The previous section compared and evaluated the merits of several important characteristics of Web services and RFCs in the SAP environment. Using this knowledge as a foundation, we can analyze the results somewhat more systematically to help you evaluate which technology is best for your integration projects. (See **Figure 15** on the next page for a summary.)

Arguments for using Web services

Let's start by examining the Web services features that are largely unique and therefore provide an argument in favor of adopting Web services technology in both SAP and non-SAP environments. First, Web services are based on communication standards that are generally accepted in the industry. Like SOAP and WSDL, these XML-based standards are at the core of integrating very heterogeneous systems. Consequently, we can say that:

- You might want to use Web services when you are dealing with very heterogeneous systems. Important examples are situations in which you can't predetermine the types of communication partners. For example, you definitely face this type of situation if you are placing an ABAP-based service in the public Internet where you simply cannot assume that the clients are RFC-enabled. This consideration also applies in many other B2B scenarios such as providing a business service as a Web service to your vendors.
- You might want to use Web services when you are already using XML in your applications and/or network communications. In my experience, XML-based solutions tend to evolve naturally into Web service solutions for many reasons. One important reason is that your internal data formats have already been converted to XML. Moving from

Scenario	Web services	RFC/RFM
Your technology environment is heterogeneous.	X	
Your technology environment is very homogeneous.		X
You are currently using XML in your applications and/or network communications.	X	
Your application scenario requires frequent changes.	X	
You have a strategic need for loose coupling sometime in the future.	X	
Your application scenario demands genuine real-time or high performance.		X
Your application transports large packets of data.		X
Your application scenario requires complex middleware features such as cross-system coordination.		X (currently)

Figure 15 Choosing between Web services and RFCs

XML to a full Web service implementation is quite a small step in comparison.

Second, Web service implementations typically come with comfortable tools that support and accelerate the development cycle. In this respect, we can assert that:

- You might want to use Web services in situations that entail frequent application or client changes. Frequent changes also usually lead to strong technology heterogeneity, which further reinforces Web services as the appropriate choice.

Finally, any discussion of the general characteristics of an SOA stresses the aspect of loose coupling. Here we need to differentiate between the various forms of coupling. In both SAP and non-SAP environments, a Web service approach currently provides almost the same level of loose coupling as a native RFC approach. Changing the interface or the address of the target service also requires changing the client, either its code or the WSDL. On the other hand, UDDI and other standards such as Web Services Addressing (WS-Addressing)[19] are currently focused on further decoupling Web services. Web services implementations may soon have a flexible framework for finding and integrating services dynamically. So, we can conclude that:

- You might want to use Web services when loose coupling is an important strategic feature, even if this choice means that today you have to make many settings manually. Don't expect immediate results in this area. However, in the long term dynamical binding to and decoupling of Web service partners will mature, resulting in inherently flexible architectures and solutions.

Arguments against using Web services

The results of testing our examples hint that RFC is superior to Web services in some areas, notably performance. Based on this quantitative data, we can agree that:

- You might *not* want to use Web services in a homogeneous SAP environment that does not heavily rely on software from other vendors. RFC often provides a faster and functionally more powerful solution because of its long

[19] For more information about WS-Addressing, go to www.w3.org/Submission/ws-addressing/.

history and tight integration into the SAP system. This long history has also led to significant RFC programming expertise in the SAP development community.

- You might *not* want to use Web services for true real-time applications such as controlling production lines. The performance overhead when processing Web service calls may prohibit using a Web service approach in this situation. However, most business applications are almost never really time-critical. In many situations, the network latency in wide-area communications conceals the Web services overhead to a great degree. But this argument applies equally to the transport of a native RFC call. In other words, although Web services suffer from significantly reduced performance, this problem is not crucial in many business situations. Network latency and human interaction times may hide the Web services overhead.

- You might *not* want to use Web services without further testing when you are dealing with large amounts of data. The tests with our example hint that Web services work best with small to medium-sized XML messages. However, you may have better options for transporting unstructured bulk data. SOAP attachments are one option, although they are not fully standardized. Microsoft initially began supporting, but eventually never released, the Direct Internet Message Encapsulation (DIME) standard for attachments back in 2002. Its successor, SOAP Message Transmission Optimization Mechanism (MTOM), is based on the W3C specification. SOAP MTOM was released in 2005, but is not yet in widespread use.

Moving on to our evaluation of transactional support, this aspect presents one of the major problems with Web services today. While many interesting standardization efforts are under way, only a few have solid implementations. This drawback applies equally to SAP and non-SAP systems. In particular, the lack of a central infrastructure for Web services translates into rudimentary support for some key middleware services that are readily available in environments such as CORBA or RFC. So, with respect to transactional support:

- You *currently* might *not* want to use Web services when the situation requires complex middleware features such as transaction coordination or composition of Web services. This situation will surely change in the near future. In fact, SAP NetWeaver Exchange Infrastructure (XI) is already working on it. But today many of these standards are still not fully implemented for Web services, including support for distributed transactions.[20]

SAP is better than most other middleware solutions because of its support of transactional behavior for Web services, albeit in a somewhat restricted sense. (Remember from the discussion of creating our example Web service client that only calls within the same application are supported easily.) However, the communication model concentrates on synchronous calls. The situation with RFC is slightly better because both transactional and queued RFC are fully established.

So, that's all that stands against using Web services in your enterprise? Yes, almost. Just one final statement that might sound trivial, but:

- You might *not* want to use Web services when your existing solution is already working fine. This observation is simply a consequence of the pragmatic approach that I mentioned at the beginning of the article. In many cases, there is no obvious or immediate need to change to using Web services technology. Therefore, take the time to study its evolution before switching.

Arguments neutral to Web services and RFCs

As we all know, technology decisions are rarely completely clear-cut and this one is no different. We conclude our evaluation with some facts that do not argue convincingly for either the Web services or native RFC approach. Both technologies are very comparable in these aspects. You therefore need to decide individually for each situation. We start by

[20] For more information about these fundamental aspects of Web services, see *Web Services – Concepts, Architectures and Applications* by G. Alonso et al. (Springer, 2003).

examining granularity, which is an often-mentioned feature in SOA discussions. Services in an SOA should exhibit a coarser granularity that is better-adapted to a discussion of business processes than the common object or component approach. In other words, services should map directly to business objects such as invoices. Business analysts and developers should easily find common ground when talking about services.

The SAP environment achieves this ideal, at least to some extent. Remote function modules are at the core of both RFC and Web services. You can combine RFMs into bigger components in either approach. In addition, SAP business objects allow for coarse granularity in both RFC and Web services models. Business objects are usually a valuable starting point for implementing coarse-grained services. However, developing Java or Microsoft .NET-based Web services currently requires careful design to achieve adequate granularity in an SAP environment.

Finally, no technology evaluation is complete without a discussion of security — in this case, the security of distributed applications. It is commonly known that RFC can readily be used behind firewalls and also allows for encrypted communications. SAProuter[21] enables the secure transport across the firewall. For Web services, SOAP messages are transported using HTTPS. Firewalls are usually open to HTTP messages and therefore usually do not pose a transport problem for Web services. However, note that the security mechanisms available with native RFC do not address all the requirements of a real B2B scenario. For example, exchanging security policies is not part of the RFC security portfolio. Several Web service standards are proposed,[22] specifying among other things how partners communicate their security policies. Yet the implementation of these standards and their interoperability is still in a very early state.

[21] SAProuter is an application gateway for the SAP Network Interface. For more information, go to the SAP Library at http://help.sap.com and navigate to the SAProuter (BC-CST-NI) area.

[22] For more information about the evolving security standards, refer to www.oasis-open.org. Navigate to the Web Services area and look for Web Services Security (WS-Security) and other related standards.

With the advent of advanced security solutions, Web services will likely become the preferred technology in security-sensitive scenarios.

Conclusion

To recap our journey, the purpose of this article was twofold:

- First I showed how to create two types of Java clients (Web service and native RFC) to access an ABAP-based Web service using the widely used Apache Axis framework. This scenario represents a typical SAP integration project. In this respect, this article serves as a tutorial for accessing ABAP-based Web services using Java and gives you a jumpstart to developing your own Java-based Web service clients to access functions in an SAP system.
- Second, I used these examples as the basis for evaluating when to use Web services vs. RFC in your projects. I evaluated some general statements regarding the nature of Web services and tested the examples to relate these assertions to the SAP environment. The results demonstrate the inherent strengths of Web services (such as interoperability) and some current problems (such as rudimentary middleware capabilities and slow performance). Armed with this information, you can make confident choices regarding when to use Web services vs. native RFCs for your SAP integration projects.

Expect to see significant developments in the middleware area in the short to mid-term. SAP, many other software vendors, and the Open Source Initiative (OSI) and its affiliated community are very active in defining and implementing new specifications. If you plan to work with Web services in the future, do yourself a favor and get a head start today. Begin playing with Web services in SAP Web AS 6.40 now and check frequently for ongoing developments in the standards area. As vendors continue to stress the interoperability aspect, Web services may well become a central technology in almost all integration projects.

