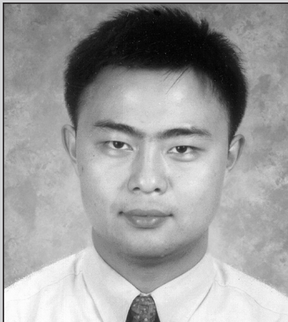
Introducing the SAP Java IDoc Class Library Part 1: An Easier Way to Write Java Programs That Create and Send IDocs to an SAP System

Robert Chu



Robert Chu joined SAP at the end of 1996. He currently works for the Integration and Certification Center at SAP Labs, where he focuses on the SAP integration technologies. Robert has been regularly teaching classes in this area at SAP training centers and is the main author of the BIT531 training course and a few other internal workshops.

(complete bio appears on page 76)

Writing programs that exchange Intermediate Documents (IDocs) with an SAP system has traditionally been a difficult task, requiring you to either spend considerable time and effort writing complex custom code that is not only difficult to develop, but also difficult to maintain, or license some pretty expensive third-party middleware. To address increasing customer demand, SAP now offers the Java IDoc Class Library, which will significantly ease the difficulties of writing Java programs that handle IDocs. This two-part article series introduces the new SAP Java IDoc Class Library and discusses the ways in which it can be used for IDoc programming.

Here in this first installment, I will discuss the basic concepts of the SAP IDoc technology and the challenges of writing Java programs to handle IDocs. I will then introduce the new SAP Java IDoc Class Library. After that, I will discuss in detail how to use the Java IDoc Class Library to write efficient Java programs that compose and send IDocs to an SAP system (the second installment covers how to receive and process IDocs coming from an SAP system). I will also provide many working examples to facilitate the discussions.

The intended readers of this article series include SAP integration architects, SAP integration developers, SAP technical consultants, and any Java developers who are interested in learning how to write Java programs that handle IDoc exchange with an SAP system.

The SAP Java IDoc Class Library works with all supported versions of the SAP R/3 system, as well as any other SAP applications that send and receive IDocs, such as CRM (Customer Relationship Management), APO (Advanced Planner and Optimizer), and EBP (Enterprise Buyer Professional).

A Brief Introduction to the SAP IDoc Technology

The IDoc technology was invented by SAP in the early 1990s to enable asynchronous messaging between SAP systems, and between an SAP system and an external system. It was first used by the EDI (Electronic Data Interchange) subsystem interface of the SAP R/3 system to exchange documents between SAP R/3 and an external EDI translator. Soon after that, SAP introduced the ALE (Application Link Enabling) technology,[1] which again used IDocs as the primary means for asynchronous messaging. Many years have passed since then, and IDoc technology has established itself alongside BAPI and RFC (Remote Function Call) as one of the most important SAP integration technologies.[2]

In the SAP system, an *IDoc message type* defines the business message to be exchanged — i.e., the semantics of the application data (e.g., IDoc message type ORDERS for a sales order or purchase order). An *IDoc type* defines the data segments and fields that are actually being used for the data exchange — i.e., the syntax of the data (e.g., IDoc type ORDERS02). One logical IDoc message type usually can be implemented by one of several different but similar technical IDoc types; sometimes one IDoc type is used in different IDoc message types. SAP transaction *WE82* shows the message type/ IDoc type relations.

An *IDoc* is a concrete instance of an IDoc type. An IDoc can be logically seen as a data container that carries related and self-contained business data to be exchanged between different SAP or non-SAP systems. An IDoc consists of a single *control record*, a number of *data records* containing the actual business data, and some *status records* that are appended to the IDoc during processing to indicate its status. When an IDoc is exchanged between systems, only the control record and the data records are transmitted.

> ✔ ***Note!***
>
> *While most of the time only the control record and data records are exchanged between systems, the receiving system can be instructed to report the status of the IDoc processing to the sending system by separately transmitting an IDoc status record. We will look at this in more detail in the second installment of this two-part article series.*

An IDoc data record contains the data of an IDoc segment. Logically, an IDoc consists of many data segments; a data segment not only contains its own data fields, it may also contain nested child data segments and their fields. Together, all data segments of an IDoc form a tree-like structure (a segment tree). The IDoc type defines the IDoc segment tree for a particular type of IDoc. You can use SAP transaction *WE60* to view the IDoc type definition (the IDoc metadata), as shown in **Figure 1**.

In the initial screen of transaction *WE60*, you can also select menu path *Documentation* → *HTML Page* to generate the HTML documentation of the IDoc type. The generated HTML page contains the same information shown in *WE60* (Figure 1), but it can be viewed in a browser, which allows for easy search and navigation.

The *WE60* transaction also allows the generation of a machine-parsable text file containing the same IDoc metadata, which can be programmatically retrieved by RFC-enabled function modules (RFMs) in the SAP system.

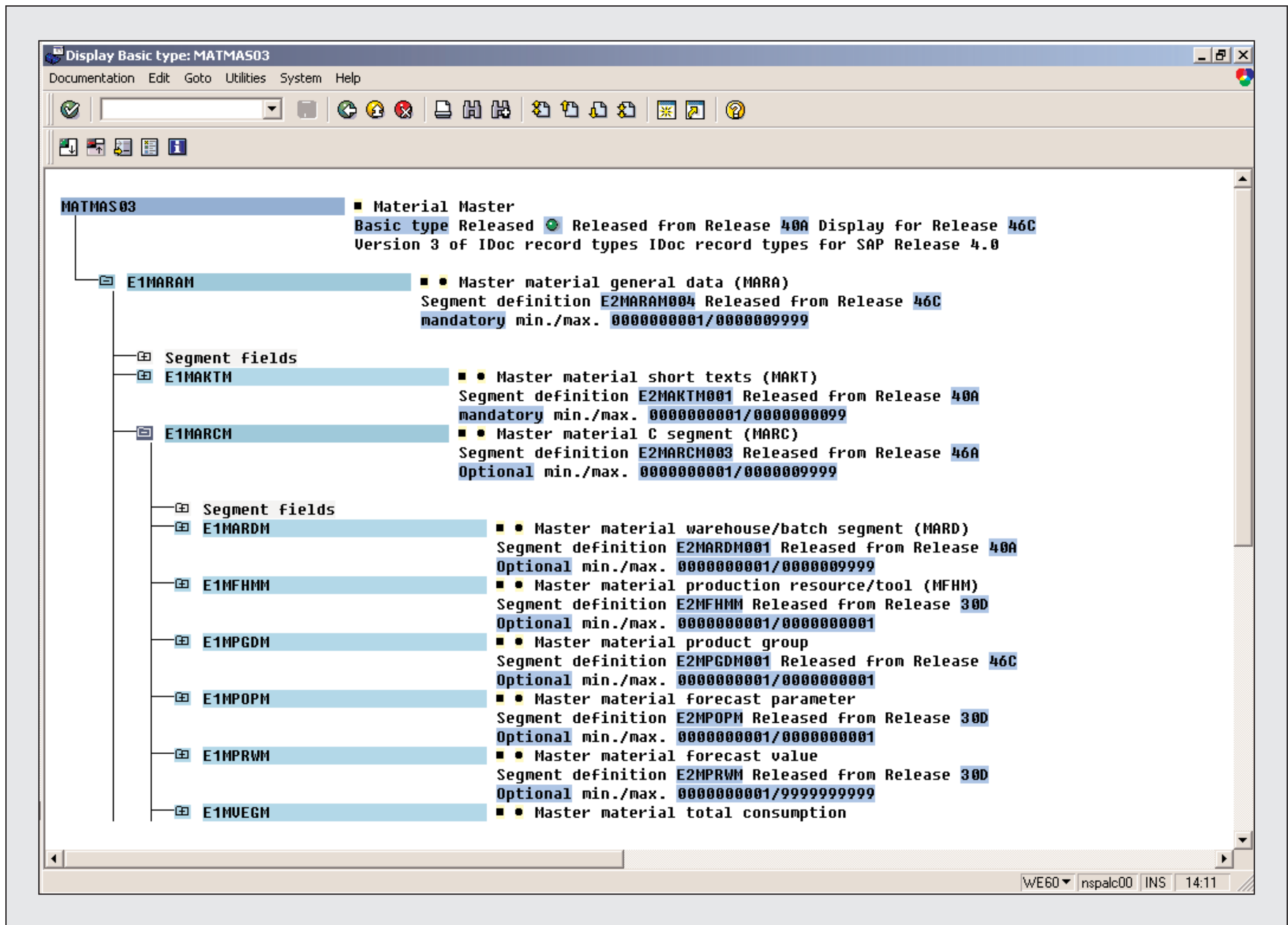
The SAP system provides a modeling and configuration tool (transaction code *SALE*) to model IDoc

[1] ALE is an SAP business process technology that enables distributed business processes across multiple separated, but logically coupled, SAP and non-SAP application systems.

[2] With the recent announcement of SAP ESA (Enterprise Service Architecture), SAP will gradually move its applications toward a services-based architecture that will allow enterprise-scale use of Web Services. However, the grand vision and great promises of Web Services will take another few years to truly mature and become widely adapted in the SAP customer base.

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 1 *Use SAP Transaction WE60 to View the IDoc Type Definition*



distribution between several systems. All modeling-related activities can be easily carried out using this modeling tool.

First, you need to give each system involved (the client within the SAP system as well as the external system) a logical system name. Then you can maintain the distribution model view, in which you define the IDoc exchanges by specifying the sender logical system, the receiver logical system, the IDoc message type, and, optionally, data filters.[3] Once the distribution model view is defined, you can either generate the systems' partner profiles based on the distribution model view (the easier and recommended way) or maintain the partner profiles manually. For *inbound* IDoc messages (IDocs received by the SAP system), the partner profile defines a process code, which indicates how the inbound IDoc should be processed, by either invoking an ABAP function module or starting an SAP workflow task. For *outbound* IDoc messages (IDocs sent by the SAP system), the partner profile defines the IDoc type to be used for the IDoc message, whether the IDoc will be transferred immediately or collected with others into an IDoc packet[4] before transfer, and, most important, the receiver port through which the IDoc will be sent.

[3] Data filters are conditions that an IDoc has to satisfy in order to be distributed — e.g., we can define a data filter for the material master (MATMAS) IDoc that allows the distribution of material data to only one particular plant.

[4] An IDoc packet contains many IDocs of the same type.

The receiver port can be generated automatically by the SAP system[5]; it can also be maintained manually using transaction *WE21*. There are several different receiver port types, each with its own way of transmitting the IDoc data to other systems:

- **Transactional RFC (tRFC):** In this case, the IDoc data is passed to the target system as parameters of a tRFC call to a certain function module. The port definition defines which version of the IDoc record types[6] will be used (version 2 for SAP R/3 3.x; version 3 for SAP R/3 4.x). It also defines which RFC destination will be used for the tRFC call. tRFC is the most commonly used port type for transmitting IDocs between systems. (This port type will be discussed in greater detail in the next section, "The tRFC Protocol.")
- **File:** With this port type, the IDoc data is written to a data file in the designated SAP application server file system directory. It isn't going anywhere automatically — it needs to be transmitted by other means to the target system. As the developer, you will have to understand the IDoc metadata to be able to correctly interpret the data contained in the file and extract the relevant portions. This port type is used by the SAP R/3 EDI subsystem interface.
- **CPIC:** This port type is used for direct access to a mainframe-based SAP R/2 system via Common user Programming Interface Communication. It is very rarely used.
- **Internet:** "Email" would be a more appropriate name for this port type — the IDoc data is sent as an Internet email attachment. Rarely used.
- **ABAP:** This port type allows you to provide an ABAP function module to further process the IDoc. It is very rarely used.
- **XML:** Despite its appealing name, this is an experimental port type — the IDoc data is written to an XML file according to the SAP IDoc-XML[7] vocabulary. The XML file sits in the SAP application server file system directory, not going anywhere — it needs to be transmitted by other means to the target system. Because of these limitations, it is rarely used.

Most of the time, tRFC calls are used to transmit IDocs. Let's have a closer look at the tRFC protocol and the function modules used with it to transmit IDocs.

The tRFC Protocol

Transactional RFC (tRFC) is a flavor of SAP's Remote Function Call (RFC) protocol. Similar to Remote Procedure Call (RPC), the RFC protocol allows a program running on one system to call function modules on another system.

The tRFC protocol is specifically designed to achieve guaranteed delivery — i.e., to guarantee that the called function module on the target system will execute once and only once. It achieves guaranteed delivery by using a transaction ID (TID), which is a globally unique 24-character ID (similar to Microsoft's global unique identifier, or GUID, concept). Before making the tRFC call, the client first needs to generate a TID. Then the client makes the tRFC call, passing the TID along with the parameter data.

When the server receives a tRFC call, it first checks the received TID against its own log of TIDs that have already been processed. If the TID is new, the server logs it and processes the call. Once the call is successfully processed, the server sends an acknowledgement (ACK) back to the client (the tRFC protocol does not allow the server to return parameter data to the client). If the TID had been previously processed, the server skips the processing and sends an ACK back to the client.

---

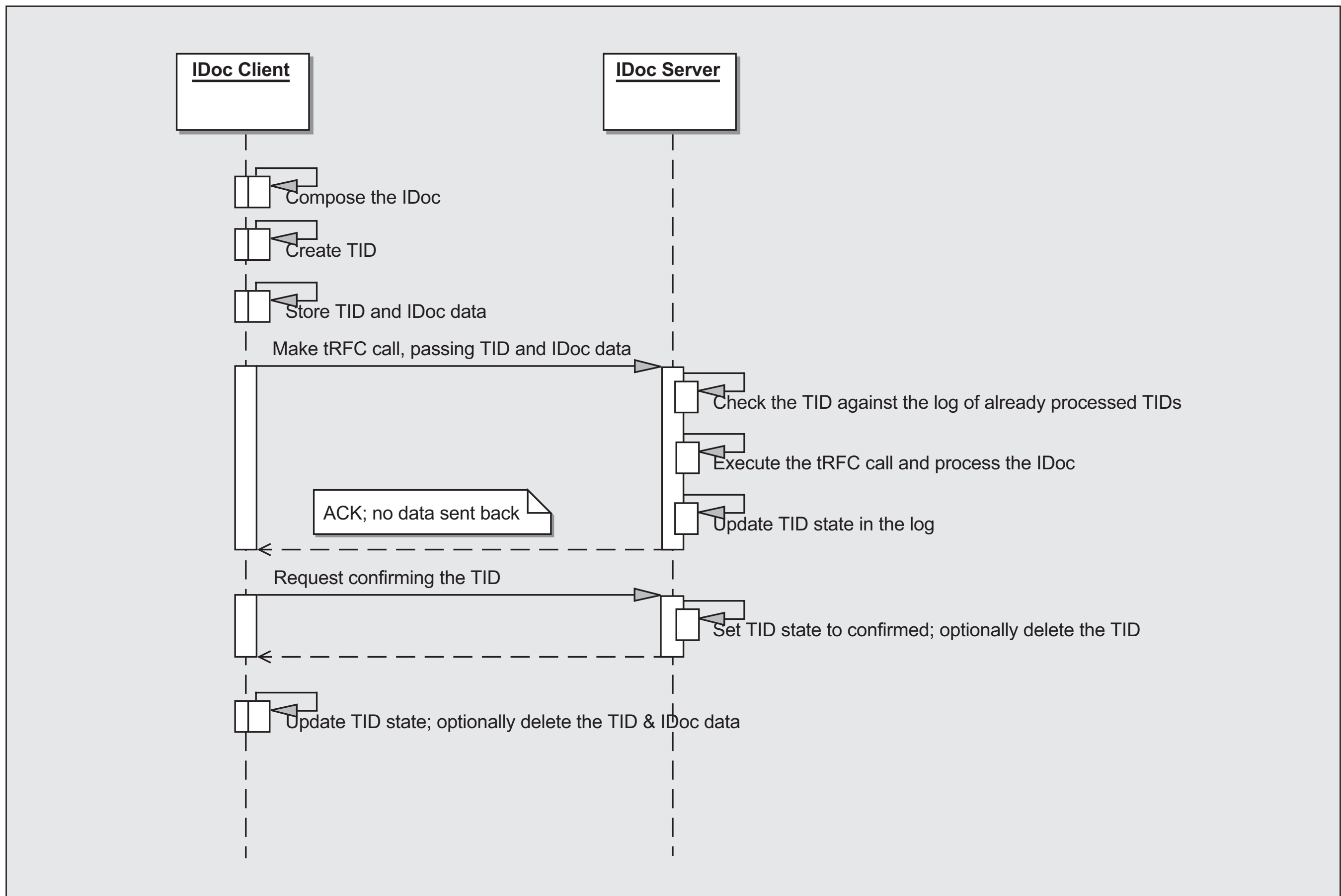
[5] If you have defined the RFC destination with the same name as the receiver logical system name.

[6] The field definitions of the IDoc control record and data record. (See the upcoming section "The Function Modules Used to Transmit IDocs with tRFC" for more details.)

[7] This is SAP's XML vocabulary for IDocs. For details, please see **http://ifr.sap.com/home/Documents/XML_Kap3_E.htm#323**.

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 2 UML Sequence Diagram of Interactions Between a tRFC Client and a tRFC Server



Once the client receives the ACK, it makes a TID confirmation call asking the server to set the TID state to "confirmed" (and perhaps to delete the TID from the server's TID log to save some space) and return an ACK.

If the client gets an RFC protocol error message or does not receive an acknowledgement in a timely fashion, it must schedule a retry of the same tRFC call with the same data and the same TID — it is the client's responsibility to make sure that failed tRFC calls are retried!

The diagram in **Figure 2** depicts the normal interactions between a tRFC client and a tRFC server.

Now let's turn our attention to the function modules used with the tRFC protocol to transmit IDoc data between systems.

The Function Modules Used to Transmit IDocs with tRFC

For version 2 IDocs (used for SAP R/3 3.x), function module INBOUND_IDOC_PROCESS is used with the tRFC protocol to transmit IDoc data between systems; for version 3 IDocs (used for SAP R/3 4.x), IDOC_INBOUND_ASYNCHRONOUS is used. These two function modules are almost identical, except for the length of some fields, which is longer in the parameters of IDOC_INBOUND_ASYNCHRONOUS.

IDOC_INBOUND_ASYNCHRONOUS has two table parameters:

- IDOC_CONTROL_REC_40 (with reference to the SAP dictionary type EDI_DC40) — This table parameter passes the IDoc's control record. If

Figure 3 **Data Record Fields**

Field Name	Data Type	Length	Description
SEGNAM	CHAR	30	Segment name
MANDT	CLNT	3	SAP client number
DOCNUM	CHAR	16	IDoc number
SEGNUM	CHAR	6	Segment sequence number
PSGNUM	NUMC	6	The sequence number of the segment's parent
HLEVEL	CHAR	2	Hierarchy level of the segment
SDATA	LCHR	1,000	Application data

there is only one IDoc to transmit, the table parameter will contain only one row for the IDoc's control record. If there are multiple IDocs (an IDoc packet) to transmit, the table parameter will contain multiple rows — one for each IDoc's control record.

A control record has 36 fields containing information such as the IDoc number, IDoc type, message type, sender partner number (name), receiver partner number (name), IDoc creation date and time, and so on. (You will see a list of all the control record fields later in Figure 5.)

- IDOC_DATA_REC_40 (with reference to the SAP dictionary type EDI_DD40) — This table parameter passes the IDoc's data records. If there is only one IDoc to transmit, the table parameter will still contain multiple rows, one for each of the IDoc's data records. If there are multiple IDocs (an IDoc packet) to transmit, the table parameter will contain even more rows to accommodate all the data records of all the IDocs in the packet.

  A data record contains seven fields, as shown in **Figure 3**. The first six fields are sometimes called the data record's *header* fields. SEGNAM is the name of the data segment, MANDT is the SAP client number, and the DOCNUM field indicates which IDoc the segment belongs to. The SEGNUM, PSGNUM, and HLEVEL fields contain the hierarchy information of the segment, which indicates the position of the segment in the IDoc segment tree.

  So then where are all the *data* fields of the segment? They are actually concatenated together, field after field according to the segment definition, and the result is crammed into the 1,000-character SDATA field![8]

The Challenges of Handling IDocs in External Programs

Now that you have a good understanding of IDoc technology and the tRFC protocol used to transmit IDocs between systems, let's take a look at the challenges involved in exchanging IDocs with external programs.

First, though conceptually the IDoc structure is clearly a tree form, when the IDocs are actually transmitted, the picture gets a little muddier. You will have to somehow retrieve the IDoc metadata. When creating and sending an IDoc, you need to use the

[8] The SDATA field contains *only* the combined data of the segment fields — no metadata, field names, or even field delimiters! This demonstrates the influence of the EDI message format. Since the IDoc technology was invented in the previous millennium (in the early 90s, way before XML), we probably shouldn't expect too much from it anyway. Note also that the SAP system does not allow a data segment's fields to have a concatenated length of more than 1,000 characters.

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

IDoc's metadata to compute the data record's header fields (such as HLEVEL, SEGNUM, and PSGNUM) and concatenate all the data fields into the long SDATA string. When receiving and processing an IDoc, you need the IDoc metadata so you can use the data record's header fields to navigate and locate the segments of interest and interpret the long SDATA string to extract the desired field values.

Second, while the tRFC protocol itself is not very complex, using it correctly requires a thorough understanding of the interactions between the tRFC client and the tRFC server, as well as careful programming and testing.

Obviously, these tasks are not trivial. Before the SAP Java IDoc Class Library was released, there were a few options for exchanging IDocs between an external program and the SAP system, but none were easy to use or cost-effective.

One option was to write lots of custom code using one of the existing SAP connectivity toolkits, such as:

- RFC-SDK[9] for the C/C++ programming languages
- SAP Java Connector (JCo) without the Java IDoc Class Library
- SAP .NET Connector[10] for the Microsoft .NET environment

You could also use the SAP Business Connector (SAP BC)[11] and take advantage of its powerful XML capabilities, but SAP BC is mainly designed to facilitate XML messaging over the Internet/intranet.[12] Though technically SAP BC can be used as a connectivity toolkit, it is generally not a very efficient approach in terms of performance if you don't need XML as the source or target format because it adds overhead while translating back and forth between SAP IDocs and XML messages.

Another option is to use the SAP Exchange Infrastructure (SAP XI).[13] However, if all you want to do is establish a direct point-to-point data exchange between your program and the SAP system, XI (a full-scale, XML-based, enterprise-wide, asynchronous message exchange infrastructure) is probably much more than you need. In addition to the implementation costs associated with XI, there are performance costs due to the additional layers of processing involved.

Of course, you could also license a third-party middleware that is certified as an SAP CA-ALE or CA-AMS interface,[14] but usually they are pretty expensive and, like SAP XI, might be overkill if you just want to exchange IDocs point-to-point with the SAP system.

If you program in Java and want to employ IDocs in one of your projects, you probably want to write programs that efficiently handle IDocs without much overhead. You likely also want to focus on the business logic, not the nitty-gritty technical details like retrieving IDoc metadata and interpreting the SDATA field. For these requirements, the SAP Java IDoc Class Library is the perfect solution.

---

[9] The C++ IDoc Class Library and the C++ RFC Class Library have been obsolete for quite a few years. Please do not use them anymore, unless you have the courage and time to maintain and bug-fix them yourself!

[10] The SAP .NET Connector doesn't offer any APIs for creating or parsing IDocs. It does offer classes that can be used to send and receive IDocs. For an in-depth discussion of the SAP .NET Connector, I strongly recommend you read Thomas G. Schuessler's article, "SAP .NET Connector for C# Programmers," in the July/August 2003 issue of this publication. Note that support and maintenance for the SAP .NET Connector's predecessor, the SAP DCOM Connector, will cease at the end of 2004. Please see SAP Note 533055 for details.

[11] The latest version of SAP BC, 4.7, actually uses the SAP Java IDoc Class Library internally to handle IDocs.

[12] For details on using SAP BC for XML messaging, please read my articles on this topic in the March/April 2003 issue of this publication.

[13] SAP XI 2.0 was released in May 2003. SAP XI contains an IDoc adapter that allows the exchange of IDocs between an SAP application system and SAP XI. SAP XI also provides a variety of other adapters to connect to various other SAP and non-SAP systems. For more details on SAP XI, visit **www.service.sap.com/xi**.

[14] CA-ALE interface certification applies to third-party ALE converter middleware and verifies that it can send, receive, and convert SAP IDocs. CA-AMS interface certification applies to third-party ALE message-handler middleware and verifies that it can send, receive, and route SAP IDocs. For more information about the SAP interface certification program, please visit the SAP Integration and Certification Center web site at **www.sap.com/icc**.

Introducing the SAP Java IDoc Class Library

The SAP Java IDoc Class Library is an add-on package based on the SAP Java Connector (JCo). You can download the library free of charge from the SAP Service Marketplace at **www.service.sap.com/connectors** (select *SAP Java Connector → Tools and Services → SAP Java IDoc Class Library*).

The SAP Java IDoc Class Library consists of two separate parts (which are also separate downloads):

- SAP Java Base IDoc Class Library (IDocBase) — This is the middleware-*independent* part. It defines a set of general base classes and interfaces, which need to be implemented by the middleware-dependent part.
- SAP Java Connector IDoc Class Library (JCoIDoc) — This is the middleware-*dependent* part. It uses SAP JCo as the connection middleware and implements all the interfaces defined in the IDocBase.

JCoIDoc is currently the only known middleware-dependent implementation for enabling the IDocBase, so we can consider the two parts as essentially one library, which I will refer to as the "IDoc library."

To install and use the IDoc library, you will need JDK (Java Development Kit) or JRE (Java Runtime Environment) version 1.2 or higher and SAP JCo version 2.1 or higher installed on your machine. After downloading the IDoc library files, it is recommended that you extract them to the same directory in which the SAP JCo directory is located so the JavaDoc URL cross-references will work.

To use the IDoc library, you need to set the Java classpath to include these downloaded files:

- **sapjco.jar:** The SAP JCo software
- **sapidoc.jar:** The SAP Java Base IDoc Class Library (IDocBase)
- **sapidocjco.jar:** The SAP Java Connector IDoc Class Library (JCoIDoc)

Navigate to the following files to view the IDoc library documentation (note that <JCoDir> indicates the location of the JCo directory in your system):

- **<JCoDir>\docs\jco\intro.html:** Contains links to the JCo release notes, installation information, etc.
- **<JCoDir>\docs\jco\index.html:** Contains links to the specifications of the JCo API
- **<JCoDir>\docs\idoc\intro.html:** Contains links to the IDocBase release notes, installation information, etc.
- **<JCoDir>\docs\idoc\index.html:** Contains the specifications of the IDocBase API
- **<JCoDir>\docs\idoc\jco\intro.html:** Contains links to the JCoIDoc release notes, installation information, etc.
- **<JCoDir>\docs\idoc\jco\index.html:** Contains the specifications of the JCoIDoc API

This article series discusses the most important aspects of using the IDoc library APIs; if you are interested in the complete API details, please refer to the appropriate JavaDoc (included with the downloads).

Now let us look at how we can write Java programs to create and send IDocs to the SAP system in a very easy way.

Writing Client Programs That Create and Send IDocs to the SAP System

Figure 4 shows a sample client program that creates a very simple IDoc of type ALEREQ01[15] and sends it to the SAP system.

To use the IDoc library, in lines 3-5 we first import the necessary Java packages, including the JCo

[15] ALEREQ01 is probably the simplest IDoc type. I use it here for illustration purposes because of its simplicity.

©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 4 **Creating an IDoc and Sending It to the SAP System**

```
 1 package com.spj.idocclient;
 2
 3 import com.sap.mw.jco.*;
 4 import com.sap.mw.idoc.*;
 5 import com.sap.mw.idoc.jco.*;
 6 import java.util.*;
 7 import java.io.*;
 8
 9 // simple IDoc client sample program, without TID management
10 public class IDocClient1 {
11   public static void main(String[] args) {
12     final String POOL = "MyPool";
13
14     try {
15       // load the connection properties from file
16       String filename = System.getProperty("user.dir") +
           System.getProperty("file.separator") + "default.properties";
17       Properties props = new Properties();
18       props.load(new FileInputStream(filename));
19       // create connection pool
20       JCO.addClientPool(POOL, 3, props);
21
22       // create an IDoc.Repository instance which is connected to the
            SAP system
23       IDoc.Repository idocRep =
           JCoIDoc.createRepository("MyIDocRepository", POOL);
24
25       // instantiate an IDoc.Document
26       IDoc.Document doc = JCoIDoc.createDocument(idocRep, "ALEREQ01");
27
28       // set IDoc control data
29       doc.setMessageType("MATFET");
30       doc.setRecipientPartnerType("LS");
31       doc.setRecipientPartnerNumber("SAPSYSTEM");
32       doc.setSenderPort("JCOWS");
33       doc.setSenderPartnerType("LS");
34       doc.setSenderPartnerNumber("JCOWS-00");
35
36       // create segments and populate fields
37       IDoc.Segment rootSegment = doc.getRootSegment();
38       IDoc.Segment segE1ALER1 = rootSegment.addChild("E1ALER1");
39       segE1ALER1.setField("MESTYP", "MATMAS");
40       segE1ALER1.setField("MESTYP40", "MATMAS");
41
42       IDoc.Segment segE1ALEQ1 = segE1ALER1.addChild("E1ALEQ1");
43       segE1ALEQ1.setField("OBJVALUE", "MATNR");
44       segE1ALEQ1.setField("SIGN", "I");
45       segE1ALEQ1.setField("OPTION", "EQ");
46       segE1ALEQ1.setField("LOW", "P-100");
```

(continued on next page)

Figure 4 *(continued)*

```
47
48         // check IDoc syntax
49         try {
50           doc.checkSyntax();
51         } catch (IDoc.SyntaxException ex) {
52           System.out.println("Syntax error: " + ex);
53           System.exit(0);
54         }
55
56         // get a connection from the pool
57         JCO.Client conn = JCO.getClient(POOL);
58         // create the TID
59         String tid = conn.createTID();
60         System.out.println("TID: " + tid);
61
62         // send the IDoc with the TID using tRFC protocol
63         conn.send(doc, tid);
64
65         // if send is successful, confirm the TID
66         conn.confirmTID(tid);
67         System.out.println("IDoc sent successfully to the SAP system.");
68
69         // release the connection
70         JCO.releaseClient(conn);
71         // shutdown the connection pool
72         JCO.removeClientPool(POOL);
73       } catch (Exception ex) {
74        ex.printStackTrace();
75       }
76     }
77 }
```

package (*com.sap.mw.jco.**), the IDocBase package (*com.sap.mw.idoc.**), and the JCoIDoc package (*com.sap.mw.idoc.jco.**). Then we instantiate and load a *java.util.Properties* object from the properties file in the user directory (lines 15-18). The properties object is used as one of the parameters to create a JCo connection pool (lines 19-20),[16] which is then used to create an *IDoc.Repository*[17] instance (lines 22-23). With the *IDoc.Repository* instance, we call *JCoIDoc.createDocument()* to create an IDoc instance of type *ALEREQ01* (lines 25-26). Then we call the setter methods of the IDoc instance to set the necessary control record fields (lines 28-34).

To populate the data segments, we first get the "virtual" root segment[18] of the IDoc instance (line 37) and then add the "real" parent data segment *E1ALER1* as a child of the virtual root segment (line 38). The

[16] Since the *IDoc.Repository* instance will need one JCo connection to the SAP system to retrieve the IDoc metadata, and our client program needs one JCo connection to send the IDoc, using a JCo connection pool is recommended here.

[17] The IDoc library's *IDoc.Repository* interface retrieves the IDoc metadata from the SAP system as necessary and caches it in memory.

[18] The root segment is not a real segment defined in the IDoc type. It is simply a virtual parent of the entire segment tree added by the IDoc library API to facilitate navigation through the tree (more on this later).

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

setField() method is invoked to set the segment fields to the desired values (lines 39-40). Since we know that the *E1ALER1* segment has a child segment called *E1ALEQ1*, we instantiate the *E1ALEQ1* segment and add it as a child of *E1ALER1* (line 42). Then the *setField()* method is called for the fields of the *E1ALEQ1* segment (lines 43-46). After all the IDoc segments and fields are populated with the desired data, we check the structural correctness of the IDoc instance by calling its *checkSyntax()* method, which will throw *IDoc.SyntaxException* if it encounters any structural problems (lines 48-54).

Now the IDoc is ready to be sent to the SAP system via the tRFC protocol. We first get a JCo connection from the pool (lines 56-57), use the connection to generate a TID (lines 58-59), then use the connection's *send()* method to transmit the IDoc together with the TID to the SAP system via the tRFC protocol (lines 62-63). If there is no error during the transmission, we confirm the TID (lines 65-66) and finally release the JCo connection and shut down the connection pool (lines 69-72). After running the program successfully, you can see in transaction *WE60* that the IDoc has been received by the SAP system.

> ✓ ***Tip***
>
> *You may see that the IDoc has an error status code in your SAP system, which means the IDoc was successfully posted to the SAP system, but the SAP system is not configured to correctly process the incoming IDoc. Ask your local ALE guru for help if you would like the application to process the IDoc.*

This simple example should give you a basic idea of IDoc client programming, which includes:

- Creating the IDoc
- Sending the IDoc using the tRFC protocol

Over the course of the rest of this article, we'll examine these aspects more closely.

Creating the IDoc

In your IDoc client program, you will need to create the appropriate IDoc and populate it using data from external sources. There are five tasks the client program needs to perform:

- Create an IDoc repository instance
- Create an IDoc instance of the appropriate type
- Set the IDoc's control record fields
- Create the IDoc's segment tree and populate the segment fields
- Perform syntax checks on the IDoc instance

Let's take a closer look at how the sample program in Figure 4 is written to address these needs.

Create an IDoc Repository Instance

To create an IDoc, you need to have an *IDoc.Repository* instance, which caches the necessary IDoc metadata. You can create this instance using the static *createRepository()* method of class *JCoIDoc* by specifying a name for the IDoc repository along with a *JCO.Client* object or JCo connection pool:

```
IDoc.Repository idocRep =
  JCoIDoc.createRepository
  ("MyIDocRepository", POOL);
```

The *IDoc.Repository* instance (i.e., the IDoc repository) will use the connection or connection pool to retrieve the IDoc metadata from the SAP system as needed. The retrieved metadata will be cached in memory by the IDoc repository. Later, if the same IDoc metadata is requested, the IDoc repository will serve the cached data directly, which improves the performance of subsequent metadata requests. Of

course, the first time an IDoc's metadata is requested, it is retrieved from the SAP system rather than the cache, which is why creating an IDoc (or processing a received IDoc) for the first time takes a little bit longer than subsequent accesses of the data.

> ✔ ***Note!***
>
> *IDoc.Repository is an interface; JCoIDoc.createRepository() actually returns an instance of JCoIDoc.SAPRepository, which implements IDoc.Repository along with its java.io.Serializable interface. This means you can serialize an already-populated repository object and later deserialize it if you need to improve the initial performance of your program by saving the metadata retrieval time. However, in doing so you run the risk of "stale" repository data, which will happen if you are connecting to an SAP system with a release other than the one used to retrieve the metadata, or if the IDoc definition has been modified in the SAP system since the last metadata retrieval, as might be the case for custom-developed, extended, or reduced IDocs. You need to weigh the risk against the benefit — usually the amount of time you will save is too trivial to justify the risk.*

Create an IDoc Instance of the Appropriate Type

With an instance of the IDoc repository created, you can now call the static *createDocument()* method of class *JCoIDoc* to create an IDoc instance, providing the *IDoc.Repository* instance and specifying the IDoc type name:

```
IDoc.Document doc =
  JCoIDoc.createDocument(idocRep,
  "ALEREQ01");
```

The call will retrieve the metadata of the specified IDoc type from the IDoc repository and use that metadata to instantiate an empty *IDoc.Document* instance of that type.

Set the IDoc's Control Record Fields

You then need to set the control record fields for the IDoc instance. Class *IDoc.Document* provides getter and setter methods for all the control record fields, as shown in **Figure 5**.

Figure 5 IDoc Control Record Fields and Their Getter and Setter Methods

Field	Description	Getter Method	Setter Method
TABNAM	Name of table structure	getTableStructureName()	N/A
MANDT	Client	getClient()	setClient()
DOCNUM	IDoc number	getIDocNumber()	setIDocNumber()
DOCREL	SAP release for IDoc	getIDocSAPRelease()	setIDocSAPRelease()
STATUS	Status of IDoc	getStatus()	setStatus()
DIRECT	Direction	getDirection()	setDirection()
OUTMOD	Output mode	getOutputMode()	setOutputMode()
EXPRSS	Overriding in inbound processing	getExpressFlag()	setExpressFlag()
TEST	Test flag	getTestFlag()	setTestFlag()
IDOCTYP	Name of basic type	getIDocType()	setIDocType()

 www.SAPpro.com ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 5 *(continued)*

Field	Description	Getter Method	Setter Method
CIMTYP	Extension (defined by the customer)	getIDocTypeExtension()	setIDocTypeExtension()
MESTYP	Message type	getMessageType()	setMessageType()
MESCOD	Message code	getMessageCode()	setMessageCode()
MESFCT	Message function	getMessageFunction()	setMessageFunction()
STD	EDI standard, flag	getEDIStandardFlag()	setEDIStandardFlag()
STDVRS	EDI standard, version and release	getEDIStandardVersion()	setEDIStandardVersion()
STDMES	EDI message type	getEDIMessageType()	setEDIMessageType()
SNDPOR	Sender port (SAP system, external subsystem)	getSenderPort()	setSenderPort()
SNDPRT	Partner type of sender	getSenderPartnerType()	setSenderPartnerType()
SNDPFC	Partner function of sender	getSenderPartnerFunction()	setSenderPartnerFunction()
SNDPRN	Partner number of sender	getSenderPartnerNumber()	setSenderPartnerNumber()
SNDSAD	Sender address	getSenderAddress()	setSenderAddress()
SNDLAD	Logical address of sender	getSenderLogicalAddress()	setSenderLogicalAddress()
RCVPOR	Receiver port	getRecipientPort()	setRecipientPort()
RCVPRT	Partner type of recipient	getRecipientPartnerType()	setRecipientPartnerType()
RCVPFC	Partner function of recipient	getRecipientPartnerFunction()	setRecipientPartnerFunction()
RCVPRN	Partner number of recipient	getRecipientPartnerNumber()	setRecipientPartnerNumber()
RCVSAD	Recipient address	getRecipientAddress()	setRecipientAddress()
RCVLAD	Logical address of recipient	getRecipientLogicalAddress()	setRecipientLogicalAddress()
CREDAT	Created on	getCreationDate(), getCreationDateAsString()	setCreationDate()
CRETIM	Time created	getCreationTime(), getCreationTimeAsString()	setCreationTime()
REFINT	Transmission file (EDI interchange)	getEDITransmissionFile()	setEDITransmissionFile()
REFGRP	Message group (EDI message group)	getEDIMessageGroup()	setEDIMessageGroup()
REFMES	Message (EDI message)	getEDIMessage()	setEDIMessage()
ARCKEY	Key for external message archive	getArchiveKey()	setArchiveKey()
SERIAL	Serialization	getSerialization()	setSerialization()
DOCTYP	IDoc type (only for version 2 IDocs)	getIDocCompoundType()	setIDocCompoundType()

Most of the getter methods return a *String* value, except *getCreationDate()* and *getCreationTime()*, which return a *java.util.Date* object. Similarly, most of the setter methods take *String* arguments, except *setCreationDate()* and *setCreationTime()*, which can take either a *String* argument or a *java.util.Date* argument.

For an IDoc to be sent to the SAP system, the following control fields are mandatory and must be populated appropriately:

- **MESTYP (IDoc message type):** This is the message type of the IDoc you are sending to the SAP system. It must match the message type used in the ALE modeling.
- **SNDPOR (sender port):** This field contains a string that describes which system is sending the IDoc. While this value will be ignored by the receiving system, a value must be set because the IDoc library requests one.
- **SNDPRT (sender partner type):** This field must be set to string *LS* to indicate that the sender is a logical system.
- **SNDPRN (sender partner number):** The sender partner number is the logical system name of the Java client program. It must match the logical system name used in the ALE modeling.
- **RCVPRT (receiver partner type):** This field must be set to string *LS* to indicate that the receiver is a logical system.
- **RCVPRN (receiver partner number):** The receiver partner number is the logical system name of the SAP system client.

Lines 28-34 of Figure 4 demonstrate setting these control record fields.

Create the IDoc's Segment Tree and Populate the Segment Fields

Once you have populated the control record's fields, you next need to create and populate the IDoc data segments. It's technically possible to write very intelligent programs to query the IDoc metadata from the *IDoc.Repository* instance and dynamically populate the IDoc data segments and fields using an appropriate data source. However, such programs are highly sophisticated and require significant skill and effort to write. Unless you are a middleware developer, in most cases you will already know what kind of IDoc you want to create anyway, so you can study the IDoc metadata beforehand (using SAP transaction *WE60*) and write straightforward programs using the IDoc library API to populate the segments and fields.

Using the *getRootSegment()* method of the *IDoc.Document* instance, you can get the "virtual" root of the segment tree. The virtual root segment is not a real data segment and does not contain any fields or data; it is used in the IDoc library API as a general entry point to make navigating through the IDoc segment tree easier. Each IDoc instance has one virtual root segment that serves as the parent of the "real" top-level parent segments.

The real parent data segments can be created as child segments of the virtual root segment (segments one level below the root) using the following methods of the *IDoc.Segment* interface:

- *IDoc.Segment addChild(String segmentType)* — Creates and adds to the segment tree a child segment of the specified segment type.
- *IDoc.Segment addChild(String segmentType, boolean subsequent)* — Creates and adds to the segment tree a child segment of the specified segment type. The boolean argument *subsequent* specifies the position at which the child segment is to be added: it will be positioned as the first (*false*) or the last (*true*) leaf (a segment with no segments below it) within the child segment group of its own type.

To then create a child segment for one of the real parent data segments, you would simply call the *addChild()* method on the desired parent segment in the same way.

Lines 37-38 and line 42 of Figure 4 illustrate

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

the use of the *getRootSegment()* and *IDoc.Segment addChild()* methods.

It is also possible to add a sibling segment of the current segment (a segment that has the same parent as the specified segment) by using one of the following methods:

- *IDoc.Segment addSibling()* — Creates and adds to the segment tree a sibling segment of the same type as the currently specified segment.
- *IDoc.Segment addSibling(boolean subsequent)* — Creates and adds to the segment tree a sibling segment of the same type as the currently specified segment. The new sibling segment will be positioned as the first (*false*) or the last (*true*) leaf within this segment type's group.
- *IDoc.Segment addSibling(String segmentType)* — Creates and adds to the segment tree a sibling segment of the specified type.
- *IDoc.Segment addSibling(String segmentType, boolean subsequent)* — Creates and adds to the segment tree a sibling segment of the specified type. The new sibling segment will be positioned as the first (*false*) or the last (*true*) leaf within a sibling segment group of its own type.

To populate the fields of a newly created data segment, you can use one of the following *setField()* methods of the *IDoc.Segment* interface:

- *void setField(int index / String[19] name, String value)*
- *void setField(int index / String name, char value)*
- *void setField(int index / String name, short value)*
- *void setField(int index / String name, int value)*
- *void setField(int index / String name, long value)*
- *void setField(int index / String name, double value)*
- *void setField(int index / String name, byte[] value)*
- *void setField(int index / String name, java.lang.Object value)*

The first argument of these *setField()* methods is either the index position or the name of the field to be set, and the second argument provides the actual value to be set. Among all these *setField()* methods, the one that takes a *String* value is used most often; others are provided for convenience. Lines 39-40 and 43-46 of Figure 4 are examples of populating the fields of the segment using the *setField()* method with a *String* value.

With an understanding of the segment tree structure of the IDoc type used, you can easily write programs using these API methods to create a segment tree and populate the fields with desired values coming from an appropriate data source.

Perform Syntax Checks on the IDoc Instance

The IDoc library also provides a syntax-checking feature that allows us to validate the correctness of the IDoc instance we just created and populated using the following methods:

- *void checkSyntax()* — Checks the syntax of the IDoc and its segments using all possible syntax checks.
- *void checkSyntax(int options)* — Checks the syntax of the IDoc and/or its segments according to the specified options, which include:
  - *IDoc.CHECK_MANDATORY_DOCUMENT_FIELDS*
  - *IDoc.CHECK_DOCUMENT_FIELD_VALUES*
  - *IDoc.CHECK_MANDATORY_SEGMENTS*
  - *IDoc.CHECK_SEGMENTS_OCCURRENCE_LIMITS*
  - *IDoc.CHECK_SEGMENT_FIELD_VALUES*

[19] To save some space, I use the *int / String* notation to indicate that the argument can be either an integer or a string.

Before sending the IDoc to the SAP system, you should make sure the IDoc is syntactically correct, so you will almost always want to perform all possible syntax checks by calling *checkSyntax()* without any argument. If a syntax error is detected, the method will throw *IDoc.SyntaxException*. You can use the following methods to examine the exception to see what went wrong and fix your program accordingly:

- *String getFieldName()* — Returns the name of the field in which the syntax error was detected.
- *IDoc.Record getRecord()* — Returns the record in which the syntax error was detected.
- *IDoc.Segment getSegment()* — Returns the data segment in which the syntax error was detected.
- *int getGroup()* — Returns the error group.
- *String getKey()* — Returns the error key.
- *String toString()* — Returns a short description of the exception.

Lines 48-54 of Figure 4 illustrate checking the syntax of an IDoc instance by calling *checkSyntax()* without an argument.

If there is no syntax error, the IDoc is ready to be sent to the SAP system.

Sending the IDoc Using the tRFC Protocol

As discussed earlier, IDocs are usually transmitted to the SAP system using the tRFC protocol. Class *JCO.Client* provides the following methods to facilitate tRFC IDoc transmission:

- *String createTID()* — Creates a new transaction ID (TID).
- *void send(java.lang.Object idoc, String tid)* — Sends an IDoc or an IDoc packet[20] to the remote server. The sending is done synchronously, but the processing of the IDoc on the server is done asynchronously.
- *void confirmTID(String tid)* — Confirms that the tRFC call associated with the specified TID has finished successfully.

Lines 58-67 of Figure 4 illustrate creating a TID, sending the IDoc with the TID, and confirming the TID.

However, remember that it's still your client program's responsibility to track the state of the tRFC transmission's TID and to retry it if an error occurs. The IDoc library does not do this for you! To make sure that tRFC client calls are conducted properly and that the IDocs are retransmitted after a transmission error, it's crucial to have the right TID management mechanism in place. For the TID management to survive a program crash or system restart, you must use some kind of persistent storage to store the TID, the transmission state, and the IDoc data itself. A relational database table is the ideal choice for such persistent storage.

In the following sections, we will look at an example of a robust tRFC client TID management implementation. We will discuss how to:

- Define a database table (*TidOut*) to store the state of the outgoing TID and the IDoc data.
- Write a Java helper class to enable the IDoc client program to access the *TidOut* table.
- Add code to the IDoc client program to track the state of the outgoing TID.
- Add code to the IDoc client program to implement automatic resending of the IDoc if transmission errors occur.

[20] An IDoc packet is represented by an instance of *IDoc.DocumentList*, which will be discussed in detail in the upcoming section "Sending Multiple IDocs in a Packet."

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Define a Database Table (TidOut) to Store the State of the Outgoing TID and the IDoc Data

To provide persistent storage for the TID and IDoc data, we can define a database table called *TidOut* with the column names shown in **Figure 6**.

Write a Java Helper Class to Enable the IDoc Client Program to Access the TidOut Table

To facilitate access to the *TidOut* table by the TID management functions of the IDoc client program, we write a Java class called *TidOutDAO* (**Figure 7**).

Figure 6 *TidOut Table Definition*

Column Name	Description	Data Type and Length	Key Column?
tid	Transaction ID (TID)	Character, 24	Yes
state	IDoc transmission tRFC call state	Integer	—
idoc	IDoc data	BLOB	—

Figure 7 *The TidOutDAO.java Helper Class*

```
 1 package com.spj.idocclient;
 2 import java.io.*;
 3 import java.sql.*;
 4 import java.util.*;
 5
 6 // DAO class for the TID management table
 7 public class TidOutDAO {
 8   public static final int IDOC_OUT_NEW = 11;
 9   public static final int IDOC_OUT_SENDING = 12;
10   public static final int IDOC_OUT_SENT = 13;
11   public static final int IDOC_OUT_CONFIRMED = 14;
12   public static final int IDOC_OUT_FAILED = 15;
13   public static final int IDOC_OUT_GAVEUP = 16;
14   public static final int IDOC_OUT_UNKNOWN = 19;
15
16   private Connection sqlConn = null;
17   private PreparedStatement writeIDocStmt = null;
18   private PreparedStatement readIDocStmt = null;
19   private PreparedStatement readStateStmt = null;
20   private PreparedStatement updateStateStmt = null;
21   private PreparedStatement readFailedTidsStmt = null;
22
23   public TidOutDAO() {
24     try {
25       Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
26       sqlConn = DriverManager.getConnection(
           "jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=TidStates",
           "sa", "");
27       writeIDocStmt = sqlConn.prepareStatement(
           "INSERT INTO tidout (tid, state, idoc) VALUES (?, ?, ?)");
```

(continued on next page)

Figure 7 *(continued)*

```
28        readIDocStmt = sqlConn.prepareStatement(
            "SELECT idoc FROM tidout WHERE tid = ?");
29        readStateStmt = sqlConn.prepareStatement(
            "SELECT state FROM tidout WHERE tid = ?");
30        updateStateStmt = sqlConn.prepareStatement(
            "UPDATE tidout SET state = ? WHERE tid = ?");
31        String sqlString = "SELECT tid FROM tidout WHERE state = "
            + IDOC_OUT_FAILED;
32        readFailedTidsStmt = sqlConn.prepareStatement(sqlString);
33      } catch (ClassNotFoundException cnfex) {
34        cnfex.printStackTrace();
35        System.exit(1);
36      } catch (SQLException sqlex) {
37        sqlex.printStackTrace();
38        System.exit(1);
39      }
40    }
41
42    // write a new TID and the IDoc/IDoc packet data with state NEW to the
        TidOut table
43    public int writeNewIDoc(String tid, Object doc) throws SQLException {
44      int rowsUpdated = -1;
45      try {
46        // serialize the IDoc
47        ByteArrayOutputStream baos = new ByteArrayOutputStream();
48        ObjectOutputStream oos = new ObjectOutputStream(baos);
49        oos.writeObject(doc);
50        oos.close();
51        byte[] idocBytes = baos.toByteArray();
52        baos.close();
53        writeIDocStmt.setString(1, tid);
54        writeIDocStmt.setInt(2, IDOC_OUT_NEW);      // the state is NEW
55        writeIDocStmt.setBytes(3, idocBytes);
56        rowsUpdated = writeIDocStmt.executeUpdate();
57      } catch (IOException ioex) {
58        ioex.printStackTrace();
59        System.exit(1);
60      }
61      return rowsUpdated;
62    }
63
64    // retrieve the IDoc/IDoc packet associated with the given TID
65    public Object getIDoc(String tid) throws SQLException {
66      Object obj = null;
67      try {
68        readIDocStmt.setString(1, tid);
69        ResultSet rs = readIDocStmt.executeQuery();
70        if (rs.next()) {
71          InputStream is = rs.getBinaryStream("idoc");
72          ObjectInputStream ois = new ObjectInputStream(is);
```

    www.SAPpro.com   ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 7 *(continued)*

```
 73         obj = ois.readObject();
 74       }
 75     } catch (ClassNotFoundException cnfex) {
 76       cnfex.printStackTrace();
 77       System.exit(1);
 78     } catch (IOException ioex) {
 79       ioex.printStackTrace();
 80       System.exit(1);
 81     }
 82     return obj;
 83   }
 84
 85   // retrieve the list of TIDs with state FAILED
 86   public String[] getFailedTids() throws SQLException {
 87     ResultSet rs = readFailedTidsStmt.executeQuery();
 88     Vector vector = new Vector();
 89     while (rs.next()) {
 90       String tid = rs.getString("tid");
 91       vector.add(tid);
 92     }
 93     if (vector.size() == 0) {
 94       return null;
 95     } else {
 96       String[] tids = new String[vector.size()];
 97       for (int i=0; i<vector.size(); i++) {
 98         tids[i] = (String) vector.get(i);
 99       }
100       return tids;
101     }
102   }
103
104   // retrieve the state of the given TID
105   public int getState(String tid) throws SQLException {
106     int idocState = IDOC_OUT_UNKNOWN;
107     readStateStmt.setString(1, tid);
108     ResultSet rs = readStateStmt.executeQuery();
109     if (rs.next()) {
110       idocState = rs.getInt("state");
111     }
112     return idocState;
113   }
114
115   // update the state of a TID
116   public int updateState(String tid, int state) throws SQLException {
117     updateStateStmt.setInt(1, state);
118     updateStateStmt.setString(2, tid);
119     int rowsUpdated = updateStateStmt.executeUpdate();
120     return rowsUpdated;
121   }
122 }
```

Figure 8 ***Constants for the Outgoing IDoc's TID State***

Constant	Meaning
IDOC_OUT_NEW	The outgoing (to SAP) IDoc has just been created.
IDOC_OUT_SENDING	The IDoc is being transmitted to the SAP system.
IDOC_OUT_SENT	The IDoc was successfully sent to the SAP system.
IDOC_OUT_CONFIRMED	The IDoc (TID) has been confirmed.
IDOC_OUT_FAILED	The previous IDoc transmission attempt failed.
IDOC_OUT_GAVEUP	Too many failed IDoc transmission attempts; gave up.
IDOC_OUT_UNKNOWN	Unknown state.

In the beginning of the *TidOutDAO* Java class shown in Figure 7 (lines 8-14), we define several constants for the state of the tRFC call's TID (see **Figure 8**).

The diagram in **Figure 9** illustrates the state transitions of an outgoing TID.

In the *TidOutDAO* class shown in Figure 7, we define the following methods to create, read, and update TID status records:

- *public int writeNewIDoc(String tid, Object doc) throws SQLException* — This method serializes the second argument, which is either an IDoc instance or an IDoc packet, and writes the TID and the serialized IDoc/IDoc packet to the *TidOut* table with state *IDOC_OUT_NEW* (lines 42-62).

- *public Object getIDoc(String tid) throws SQLException* — This method retrieves the IDoc/IDoc packet associated with the given TID and returns *null* if no matching object is found (lines 64-83).

- *public String[] getFailedTids() throws SQLException* — This method retrieves the list of TIDs with state *IDOC_OUT_FAILED* and returns *null* if no such TID is found (lines 85-102).

- *public int getState(String tid) throws SQLException* — This method retrieves the state of the given TID and returns *IDOC_OUT_UNKNOWN* if no matching record is found (lines 104-113).

- *public int updateState(String tid, int state) throws SQLException* — This method updates the state associated with the given TID to the state value given (lines 115-121).

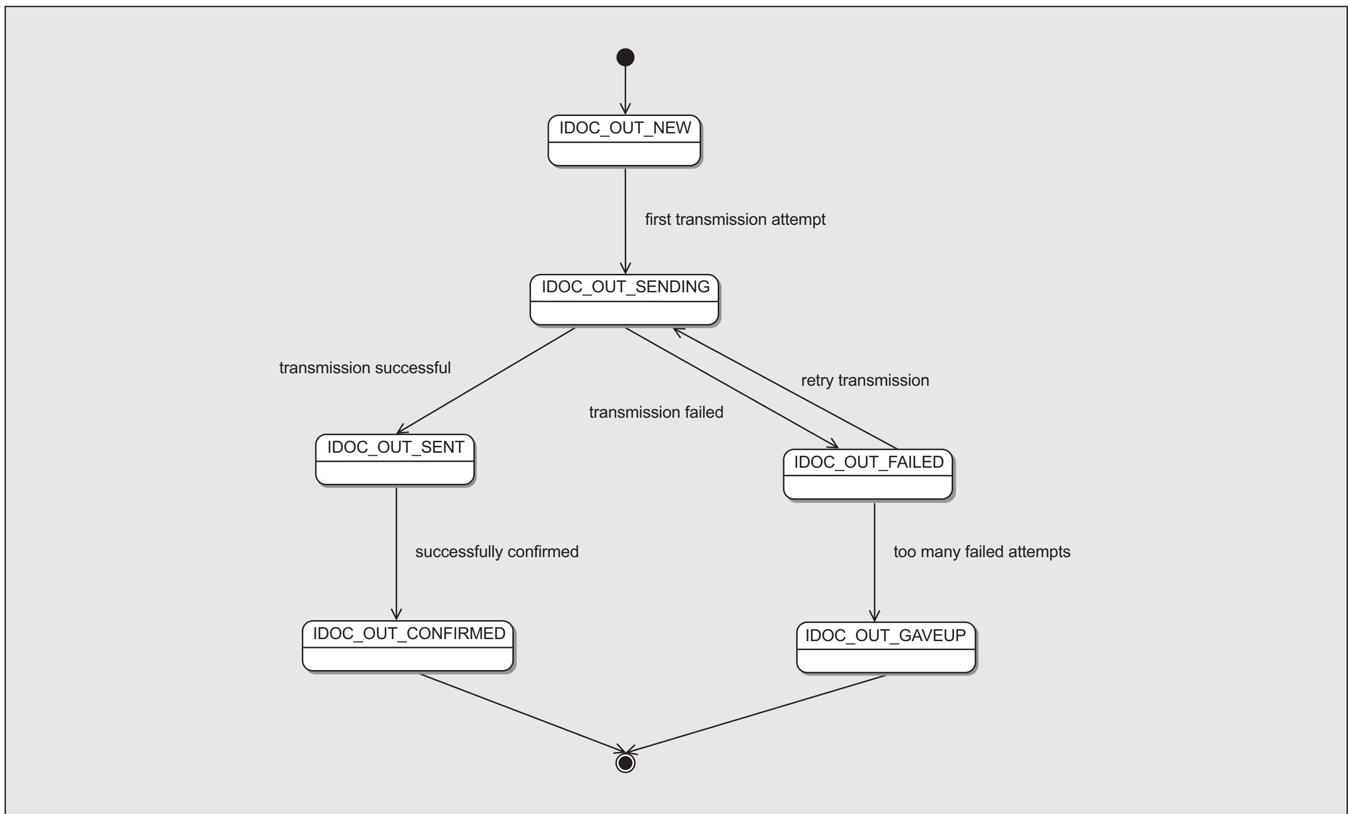
> ✓ ***Note!***
>
> *The implementation of these methods uses prepared JDBC (Java Database Connectivity) statements, which are initialized in the TidOutDAO() constructor, to improve the performance of repetitive database access. The sample code shown in Figure 7 uses the Microsoft SQL Server JDBC driver; you may need to change the name of the JDBC driver class as well as the JDBC connection string according to your own settings.*

Add Code to the IDoc Client Program to Track the State of the Outgoing TID

With the *TidOutDAO* helper class providing access to the TID's state table and tracking functions, we can

 www.SAPpro.com ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 9 UML State Transition Diagram of an Outgoing TID



enable our IDoc client program to track the state of the outgoing IDoc TID, by replacing lines 58-67 in Figure 4 with the code shown in **Figure 10**.

First, we instantiate the *TidOutDAO* helper class (lines 4-5) we created in Figure 7. Immediately after the IDoc is created and the TID is generated, we write the TID and the IDoc data to the *TidOut* table with the state *IDOC_OUT_NEW* (lines 6-7). Right before sending the IDoc, we update the TID state to *IDOC_OUT_SENDING* (lines 9-10). If an exception occurs, the TID state becomes *IDOC_OUT_FAILED* (lines 14-23). If the IDoc is sent successfully, we update the TID state to *IDOC_OUT_SENT* (lines 25-26). After confirming the TID, we set the TID state to *IDOC_OUT_CONFIRMED* (lines 27-29).

Figure 10 Using the TidOutDAO Helper Class to Track the State of the Outgoing IDoc's TID

```
1 // create the TID
2 String tid = conn.createTID();
3
4 // use the TidDAO instance to track the TID state in DB table
5 TidOutDAO tidOutDAO = new TidOutDAO();
6 // write the TID and IDoc to DB with state IDOC_OUT_NEW
7 tidOutDAO.writeNewIDoc(tid, doc);
```

(continued on next page)

Figure 10 (continued)

```
 8
 9 // i'm going to send the IDoc now ...
10 tidOutDAO.updateState(tid, TidOutDAO.IDOC_OUT_SENDING);
11 try {
12   // send the IDoc with the TID using tRFC protocol
13   conn.send(doc, tid);
14 } catch (JCO.Exception jcoex) {      // communication error
15   tidOutDAO.updateState(tid,
       TidOutDAO.IDOC_OUT_FAILED);      // set the state to FAILED
16   jcoex.printStackTrace();
17   System.exit(1);                    // retry-after-error to be
                                           implemented later
18 } catch (RuntimeException rtex) {   // error parsing the IDoc
19   tidOutDAO.updateState(tid,
       TidOutDAO.IDOC_OUT_FAILED);      // set the state to FAILED
20   // this exception shouldn't happen here, because checkSyntax()
        has been done earlier; notify administrator!
21   rtex.printStackTrace();
22   System.exit(1);
23 }
24
25 // no exception, IDoc sent successfully!
26 tidOutDAO.updateState(tid, TidOutDAO.IDOC_OUT_SENT);
27 // confirm the TID
28 conn.confirmTID(tid);
29 tidOutDAO.updateState(tid, TidOutDAO.IDOC_OUT_CONFIRMED);
30 System.out.println("IDoc sent successfully to the SAP system.");
```

Being able to track the TID state correctly is just one step toward automatically resending the IDoc if an error occurs, which we will look at next.

Add Code to the IDoc Client Program to Implement Automatic Resending of the IDoc If Transmission Errors Occur

Remember, if the client gets an RFC protocol error message or does not receive an acknowledgement from the server in a timely fashion, it is the client's responsibility to schedule a retry of the same tRFC call with the same data and the same TID.

The error-resending functionality can be implemented using one of two approaches:

- Resend in the current thread using a loop and then sleep before retrying.
- Delegate resending to a background monitoring daemon.

The sample code in **Figure 11** illustrates the first approach. If you compare the code in Figure 11 to that in Figure 10, you will see we use a *for* loop (line 15) to resend the IDoc if a transient error occurs. If such an error occurs (line 21), we log the *IDOC_OUT_FAILED* state (line 22) and check whether the retry limit is reached (line 24). If the retry limit is not reached, we sleep for a predefined period (lines 25-27) and try again (line 31). Otherwise, we log the state as *IDOC_OUT_GAVEUP* (line 29) and stop trying.

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 11 ***Resending the IDoc with a Loop If an Error Occurs***

```
 1 static final int RETRY_LIMITS = 3;
 2 static final int RETRY_INTERVAL = 10000;    // 10 seconds
 3 // ...
 4 // get a connection from the pool
 5 JCO.Client conn = JCO.getClient(POOL);
 6 // create the TID
 7 String tid = conn.createTID();
 8
 9 TidOutDAO tidOutDAO = new TidOutDAO();
10
11 // write the TID and IDoc to DB with state IDOC_OUT_NEW
12 tidOutDAO.writeNewIDoc(tid, doc);
13
14 // loop to try sending the IDoc until success or reaching retry limit
15 for (int numOfTry = 0; numOfTry < RETRY_LIMITS; numOfTry++) {
16   // try to send the IDoc now ...
17   tidOutDAO.updateState(tid, TidOutDAO.IDOC_OUT_SENDING);
18   // send the IDoc with the TID using tRFC protocol
19   try {
20     conn.send(doc, tid);
21   } catch (JCO.Exception jcoex) {           // communication error
22     tidOutDAO.updateState(tid,
         TidOutDAO.IDOC_OUT_FAILED);           // set the state to FAILED
23     jcoex.printStackTrace();
24     if (numOfTry < RETRY_LIMITS - 1) {      // if retry limit not
                                                  reached, wait for the
                                                  RETRY_INTERVAL seconds
25       try {
26         Thread.sleep(RETRY_INTERVAL);
27       } catch (InterruptedException irex) {  }
28     } else {                                // retry limit reached, give up
29       tidOutDAO.updateState(tid,
           TidOutDAO.IDOC_OUT_GAVEUP);         // set the state to GAVEUP
30     }
31     continue;                               // retry sending
32   } catch (RuntimeException rtex) {         // error parsing the IDoc
33     tidOutDAO.updateState(tid,
         TidOutDAO.IDOC_OUT_FAILED);           // set the state to FAILED
34     // this exception shouldn't happen here, because checkSyntax()
          has been done earlier; notify administrator!
35     rtex.printStackTrace();
36     System.exit(1);
37   }
38
39   // no exception, IDoc sent successfully!
40   tidOutDAO.updateState(tid, TidOutDAO.IDOC_OUT_SENT);
41   // confirm the TID
42   conn.confirmTID(tid);
43   tidOutDAO.updateState(tid, TidOutDAO.IDOC_OUT_CONFIRMED);
44   System.out.println("IDoc sent successfully to the SAP system.");
45   break;                                    // leave the loop
46 }
```

While this approach is easy to implement, it can only recover from transient errors like a temporary network outage. A more robust approach is to use a background daemon to monitor the IDoc transmission and resend after an error.

For example, we can create a daemon that monitors the *TidOut* table. If the daemon finds a TID with an *IDOC_OUT_FAILED* state, it automatically tries to resend the IDoc. The sample code in **Figure 12** illustrates an implementation of this approach using the JDK 1.3 classes *java.util.Timer* and *java.util.TimerTask*.

In this *TidOutManager* class, we first define a top-level static inner class, *IDocResender*, which extends *TimerTask* (line 13). In its constructor, we instantiate a *TidOutDAO* object (lines 14-16). In the *run()* method (lines 19-60), we use the *getFailedTids()* method of the *TidOutDAO* class to sweep through the *TidOut* table for TIDs with state *IDOC_OUT_FAILED* (lines 25-26). If we find any such TID, we try to send the associated IDoc to the target SAP system with the same TID and log the TID state changes along the way (lines 30-53). In the *main()* method of *TidOutManager* (lines 62-79), we first create the JCo connection pool (lines 65-70) and then use a *Timer* instance (line 72) to schedule the *IDocResender* instance thread to execute periodically at the interval defined by constant *SWEEPINTERVAL* (lines 73-75).

Of course, we could further enhance the *TidOut* table definition and this daemon program to track the number of sending attempts and cease retry after the retry limit is reached. I will leave this to you to implement.

So far, we've seen how to create and send a single

Figure 12 ***Creating a Background Daemon to Monitor the IDoc Transmission and Resend If an Error Occurs***

```
 1 package com.spj.idocclient;
 2
 3 import com.sap.mw.jco.*;
 4 import java.util.*;
 5 import java.io.*;
 6 import java.text.*;
 7
 8 // sample program that monitors the TidOut table periodically and
      resends IDocs with state IDOC_OUT_FAILED
 9 public class TidOutManager {
10   static final String POOL = "MyPool";
11
12   // definition of the worker thread that sweeps through TidOut table
        and resends IDoc with FAILED state
13   public static class IDocResender extends TimerTask {
14     TidOutDAO tidDAO = null;
15     public IDocResender() {
16       tidDAO = new TidOutDAO();
17     }
18
19     public void run() {
20       JCO.Client conn = null;
21       String[] tids = null;
22
```

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 12 *(continued)*

```
23        System.out.println(DateFormat.getTimeInstance().format(new
            java.util.Date()) + ": Sweeping through the TidOut table");
24        try {
25          // search TID table for IDocs with state IDOC_OUT_FAILED
26          tids = tidDAO.getFailedTids();
27          if (tids == null)
28            return;
29
30          // get the JCO connection and try to resend the IDoc(s)
31          conn = JCO.getClient(POOL);
32          for (int i=0; i<tids.length; i++) {
33            String tid = tids[i];
34            // get the IDoc/IDoc packet associated with the TID
35            Object idoc = tidDAO.getIDoc(tid);
36            // try to send the IDoc now ...
37            tidDAO.updateState(tid, TidOutDAO.IDOC_OUT_SENDING);
38            try {
39              conn.send(idoc, tid);
40              // no exception, IDoc sent successfully!
41              tidDAO.updateState(tid, TidOutDAO.IDOC_OUT_SENT);
42              // confirm the TID
43              conn.confirmTID(tid);
44              tidDAO.updateState(tid, TidOutDAO.IDOC_OUT_CONFIRMED);
45              System.out.println("IDoc(s) with TID " + tid + "
                successfully sent to the SAP system.");
46            } catch (JCO.Exception jcoex) {      // communication error
47              tidDAO.updateState(tid,
                TidOutDAO.IDOC_OUT_FAILED);      // set the state to FAILED
48              jcoex.printStackTrace();
49            } catch (RuntimeException rtex) {    // error parsing the IDoc;
                                                      shouldn't happen here
50              tidDAO.updateState(tid,
                TidOutDAO.IDOC_OUT_FAILED);      // set the state to FAILED
51              rtex.printStackTrace();
52            }
53          }                                      // end for
54        } catch (Exception ex) {
55          ex.printStackTrace();
56        } finally {
57          JCO.releaseClient(conn);
58        }
59      }
60    }
61
62    public static void main(String[] args) {
63      final int SWEEPINTERVAL = 60000;           // 60 seconds
64      try {
65        // load the connection properties from file
66        String filename = System.getProperty("user.dir") +
            System.getProperty("file.separator") + "default.properties";
```

(continued on next page)

Figure 12 (continued)

```
67        Properties props = new Properties();
68        props.load(new FileInputStream(filename));
69        // create connection pool
70        JCO.addClientPool(POOL, 3, props);
71
72        Timer myTimer = new Timer();
73        IDocResender idocResender = new IDocResender();
74        // schedule the IDoc resender to run periodically
75        myTimer.schedule(idocResender, 0, SWEEPINTERVAL);
76      } catch (Exception ex) {
77        ex.printStackTrace();
78      }
79    }
80 }
```

IDoc to the SAP system. If you need to simultaneously transmit multiple IDocs of the same type to the SAP system, it's better to send them in an IDoc packet — all the IDocs will be sent together in just one tRFC call, which improves performance.

Our *TidOut* table and *TidOutDAO* classes also can handle *IDoc.DocumentList* in the same way they handle *IDoc.Document*. So once the IDoc packet is created and populated, it can be sent to the SAP system using the same techniques discussed earlier.

Sending Multiple IDocs in a Packet

The IDoc library uses *IDoc.DocumentList* to represent an IDoc packet. It's very easy to create an IDoc packet and add individual IDocs to it, as illustrated in **Figure 13**.

> ✓ ***Note!***
>
> *All the IDocs in an IDoc packet have to be of the same IDoc type. Otherwise, the SAP system will not be able to process them correctly.*

The *send()* method of class *JCO.Client* can take an *IDoc.DocumentList* instance as its first argument. In that case, all the IDocs in the *IDoc.DocumentList* will be transmitted in one packet — in one tRFC call.

Helpful Hints

The following is a quick reference of useful tips presented throughout this article.

☑ Use SAP transaction *WE60* to view the IDoc segment tree metadata.

☑ The tRFC protocol is typically used to transmit IDocs between the SAP system and the external program.

☑ The RFMs (RFC-enabled function modules) used are IDOC_INBOUND_ASYNCHRONOUS for version 3 IDocs (for SAP R/3 4.x systems) or INBOUND_IDOC_PROCESS for version 2 IDocs (for SAP R/3 3.x systems).

☑ Create an *IDoc.Repository* instance first, then use it to create the *IDoc.Document* instance.

☑ IDoc control record fields can be set using the

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 13 ***Creating an IDoc Packet and Adding IDocs to It***

```
 1 // instantiate an IDoc.DocumentList instance
 2 IDoc.DocumentList docList = JCoIDoc.createDocumentList(idocRep);
 3
 4 // create an IDoc instance and add it to the list
 5 IDoc.Document doc = ...
 6 docList.add(doc);
 7
 8 // create another IDoc instance and add it to the list
 9 doc = ...
10 docList.add(doc);
11
12 // create additional IDoc instances and add them to the IDoc list ...
```

Dealing with IDoc Versions and SAP Releases

Selecting the Appropriate IDoc Version

As mentioned earlier in the article, there are two versions of IDoc record types — version 2 for SAP R/3 3.x systems and version 3 for SAP R/3 4.x systems. Since the IDoc repository contains the IDoc metadata retrieved from the target SAP system, the SAP Java IDoc Class Library uses this information to automatically choose the appropriate version for the IDoc transmission to the target SAP system. Class JCO.Client also provides a send() method that allows you to specify the IDoc version to be used (though it's not necessary to do so):

void send(java.lang.Object idoc, String tid, char idoc_version)

The idoc_version argument indicates the IDoc version to be used. Possible options are:

- JCO.IDOC_VERSION_DEFAULT (version 2 for R/3 3.x, version 3 for R/3 4.x)
- JCO.IDOC_VERSION_2
- JCO.IDOC_VERSION_3

Using Segment Definitions from Different SAP Releases

Depending on the SAP system release, an IDoc segment may have slightly different segment definitions. The SAP Java IDoc Class Library only allows us to use release-independent logical segment names (a.k.a. "E1" segment names) instead of release-dependent segment names (a.k.a. "E2" segment names). It also automatically chooses the most suitable definition for a segment based on the metadata of the target SAP system. For these reasons, the segment definitions are hidden from the developer and thus taken care of automatically by the SAP Java IDoc Class Library.

An older segment definition that is available in an earlier SAP system release is usually compatible with the newer segment definitions in later SAP releases. So, if you write an IDoc program for a certain target SAP release, the program should work just as well with later SAP releases. The reverse may not be true. In any case, the checkSyntax() method of the IDoc.Document class will report to you (by throwing an exception) should any problems arise as a result of segment definition incompatibilities.

setter methods. There are six mandatory control record fields.

☑ The "virtual" root segment is not a real data segment. It's the navigational entry point to the IDoc segment tree.

☑ You can easily add segments to the segment tree and set segment field values by using the various methods of *IDoc.Segment*.

☑ Make sure you call the *checkSyntax()* method on the IDoc instance before sending it!

☑ You must implement TID management for your IDoc client program.

☑ Use a database table to track the state of the outgoing TID. Write a Java helper class to facilitate the IDoc client program's access to the table.

☑ You must implement error handling/transmission retries yourself, by either coding a loop into the same thread or delegating it to a background daemon. In the latter case, the JDK classes *java.util.Timer* and *java.util.TimerTask* come in handy.

☑ Send multiple IDocs of the same type in an IDoc packet to improve performance!

Conclusion

The SAP Java IDoc Class Library has made the traditionally difficult, time-consuming, and sometimes expensive task of handling IDocs with Java programs very easy.

Here in the first part of this article series, I discussed the basic concepts of the SAP IDoc technology and introduced the IDoc library. I then showed you how to write efficient Java programs to compose and send IDocs to an SAP system and how to implement robust TID management for IDoc client programs. In the following installment, I will discuss how to use the IDoc library to write Java programs that receive and process IDocs from an SAP system.

I hope I've convinced you how easy it is to write IDoc client programs and that now you are ready to start writing your own.

Robert Chu joined SAP at the end of 1996. He currently works for the Integration and Certification Center at SAP Labs in Palo Alto, California. Previously, he worked in technical consulting and training at SAP America and SAP Asia. Robert's current focus is SAP's integration technologies. He has been regularly teaching classes in this area at SAP training centers and is the main author of the BIT531 training course, as well as a few other internal workshops. Robert has spoken at the past three SAP TechEd events. In addition to his SAP expertise, he is also an SCEA (Sun Certified Enterprise Architect) for J2EE, an MCSD (Microsoft Certified Solution Developer), and an MCSE (Microsoft Certified System Engineer). Robert can be reached at robert.chu@sap.com.

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.