RC24090 (W0610-148) October 26, 2006
Computer Science

IBM Research Report

TxBeans: Light-Weight Enablement of Transactional, Intermittently Connected, XML-Based Applications

Avraham Leff, James T. Rayfield
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

TxBeans: Light-Weight Enablement Of Transactional, Intermittently Connected, XML-Based Applications

Avraham Leff James T. Rayfield

Abstract

XML documents are the heart of many light-weight (e.g., Web 2.0) applications. TXBEANS is a light-weight and unintrusive way for such applications to transactionally manipulate their XML documents. TXBEANS also enables such applications to execute while disconnected from the server and then transparently merge this work with the master-copy maintained on the server. TXBEANS is designed to support XML documents in general, but is customized for Apache XMLBeans. Our implementation is easily modified to support other XML Document APIs. This white-paper explains the function provided by TXBEANS, discusses the relationship between XMLBeans and TXBEANS, and gives detailed examples of the TXBEANS API and programming model.

Index Terms

XMLBeans, TxBeans, XML components, disconnected applications, method replay

I. Introduction

XML documents are the heart of many light-weight (e.g., Web 2.0) applications, but are typically manipulated without the benefit of persistence and transaction services. Although the benefits of such services are well-known in the context of server-side applications, it is often assumed that they are too “heavy-weight” for use in client environments. Also, while developers would like their applications to execute in both “connected” and “disconnected” mode, in practice, this is difficult to accomplish. Because the master copy of the application’s data is stored on the server, applications that cache their data in order to execute in disconnected mode must somehow propagate their work when they reconnect to the server. It is often assumed that this task (a “reconnection” service) is too difficult to perform for applications in a general fashion.

TXBEANS shows that these services can be provided in a light-weight and unintrusive way to Java developers. It provides:

- Light-weight file-based persistence and transactions. Two styles of transactions are provided to developers:
  - A JTA [2] implementation providing fine-grained transactions, under programmer control.
  - “Auto-magical” transactions provided automatically to all methods declared on a specified interface.
- Middleware that captures a disconnected client’s work and propagates it to the server. Sophisticated “method-replay” algorithms merge the client’s work to the master-copy maintained on the server.

A. TxBeans, XMLBeans, & XML Documents

TXBEANS supports applications whose state derives from XML documents (in contrast, for example, to relational databases). Although TXBEANS is designed for XML documents in general, it is customized for

IBM T.J. Watson Research Center email: avraham@ibm.com
IBM T.J. Watson Research Center email: jtray@ibm.com

use with XMLBeans applications. XMLBeans [5] provides a component model for XML documents using a Java-language API. A document's "types" are specified using an XML Schema [8]: when this schema is compiled, XMLBeans generates Java classes corresponding to the schema's types. Developers can then access and navigate document instances with an API that is more "Java-friendly" (e.g., JavaBeans-style accessors) than the XML Document Object Model (DOM) [7]. In this *XmlObject*-based approach, XMLBeans provides strongly-typed access to an XML document. XMLBeans also provides efficient low-level access to the XML Infoset through an *XmlCursor*. More information about XMLBeans can be found at [6].

Our experience with XMLBeans leads us to conclude that XMLBeans is the technology of choice for building Java applications that use a "model" represented in XML. The TXBEANS middleware enhances the XMLBeans technology to provide transactional support [1] for access to the XML documents that are encapsulated by XMLBeans. XMLBeans does not provide isolation between concurrent access to an XML document, nor does it ensure that the document is updated in an atomic manner. TXBEANS supplies this function in a way that is minimally invasive to existing XMLBeans applications.

Typically, an application using XMLBeans begins by using an XMLBeans "Factory" to parse an XML document that is stored in a file (or string, HTTP URL, etc.) into a "Document" that corresponds to the top-level element of a schema (see Code Fragment 1).

Code Fragment 1 Hydrating a Stored XMLBeans Document

```
File orderEntry = new File("c:/orderEntry.xml");
OrderEntryDocument oeDocument = null;
try {
  // Bind the stored XML to an XMLBeans type.
  oeDocument = OrderEntryDocument.Factory.parse(orderEntry);
}
catch (Exception e) {
  e.printStackTrace();
}
```

The application's business logic then instantiates XMLBeans components as needed from the document (see Code Fragment 2).

Finally, if the document is modified (e.g., an agent places a new order) the document is saved to a file as shown in Code Fragment 3).

II. TRANSACTIONAL XMLBEANS APPLICATIONS

In a sense, one solution for transactionally enabling XMLBeans applications is trivial: simply store the XML document in a database such as Derby or MySQL and require that every access to the corresponding XMLBeans components be scoped in a transaction. The problem with this approach is that it runs somewhat counter to the light-weight flavor of XMLBeans since it requires that a database be installed everywhere that an XMLBeans application is developed and deployed. TXBEANS therefore takes the approach of requiring only that the XML document be stored in the local file system. Although a file-based solution is less robust than a database solution, it is very simple to install.

In TXBEANS, a document's global XMLBeans `XmlObject` is stored in a `XmlDataBase`, and the association between the database and the file system is done by a `XmlDataBaseFactory` (see Code Fragment 4).

TXBEANS requires only that XMLBeans developers:

Code Fragment 2 Using an XMLBeans Document

```
  static Customer
    getCustomerById(final OrderEntryDocument document, final String customerId)
  {
    final StringBuffer          pathExpression =
      new StringBuffer("\$this/ord:OrderEntry/ord:customers[ord:id=");
    pathExpression.append(SQ+customerId+SQ);
    pathExpression.append(’]’);
    final XmlObject[]    results = // Execute the query
      document.selectPath(_namespaceDeclaration + pathExpression.toString());
    Customer             customer = null;
    boolean              hasResults = false;
    if (1 == results.length)
      customer = (Customer) results[0];
    else
      throw new RuntimeException
        ("Could not find Customer with id = "+customerId);
    return (customer);
  }
```

Code Fragment 3 Saving an XMLBeans Document

```
try {
  final OutputStream      outputStream =
    new FileOutputStream("c:/orderEntry.xml");
  document.save(outputStream);
  outputStream.close();
}
catch (IOException e) {
  System.err.println ("Problem writing the xml document: "+e);
}
```

1. Access an XMLBeans document (the top-level `XmlObject`) through the `XmlDataBase` API discussed below.
2. Wrap all such access in a transaction (including the initial `XmlDataBase.setDocument()`).

A. *Using an XmlDataBase*

Code Fragment 5 shows how an XMLBeans document is read from a XmlDataBase and, once read, used with the existing XMLBeans API.

As shown in Code Fragment 6, after an XMLBeans application reads and modifies the XML document, it can write it back to the database.

Finally, the schema type of the documents stored in an XmlDataBase can be queried through the `getSchemaType()` API.

Figure 1 shows the architecture through which TXBEANS provides transactional support for XMLBeans applications.

B. *XmlDataBase Access & Transactions*

As mentioned above, all application access of an XmlDataBase must be mediated through a transaction. As shown in Code Fragment 7, TXBEANS provides an implementation of the standard `javax.transaction.UserTransaction` interface [2].

Code Fragment 4 Creating a TXBEANS Database

```
    final String directoryName = "c:/tmp/orderEntry";
    final java.io.File dbDirectory = new File(directoryName);
    dbDirectory.mkdir();
    XmlDataBase xmldb = XmlDataBaseFactory.getSingleton().create
      (directoryName, OrderEntryDocument.type);

    xmldb.setDocument(OrderEntryDocument.Factory.newInstance());
```

Code Fragment 5 An XMLBeans Application Reading a Document From a XmlDataBase

```
XmlObject db = xmldb.getDocument();
OrderEntry orderEntry = db.getOrderEntry();
int numberOfCurrentOrders = orderEntry.sizeOfOrdersArray();
```

Code Fragment 6 Writing an XML Document To a XmlDataBase

```
XmlDateTime entryDate = XmlDateTime.Factory.newInstance();
entryDate.setCalendarValue(currentTime);
newOrder.xsetEntryDate(entryDate);
orderEntry.insertNewOrders(numberOfCurrentOrders);
orderEntry.setOrdersArray(numberOfCurrentOrders, newOrder);
```

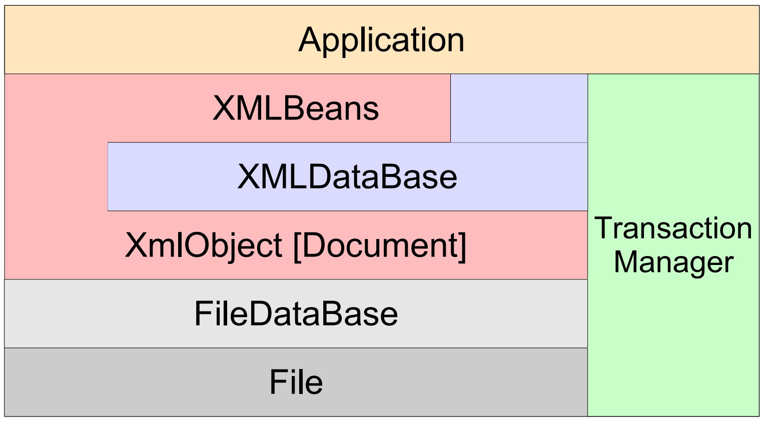

Fig. 1. TXBEANS: Providing Transactions for XMLBeans

Code Fragment 7 TXBEANS API for Creating and Using Transactions

```
final UserTransaction userTran = UTFactory.getUserTransaction();
userTran.begin();
db.setDocument(document);
userTran.commit();
```

If using this approach, developers can take advantage of the `com.ibm.oats.txnengine.TxUtils` convenience methods such as `TxUtils.begin()` and `TxUtils.commit()`.

Alternatively, TXBEANS allows developers to specify that one or more Java interfaces be automatically enhanced so that their methods execute transactionally. This is the same approach that TXBEANS uses to support intermittent client-side connectivity (Section III). Figure 8 shows how a developer:

1. Specifies a "PlaceOrder" interface,
2. Provides an implementation of that interface,
3. Gets an instance of the "PlaceOrder" interface from the "ProxyFactory" provided by TXBEANS,
4. Invokes "createOrder" and "placeOrder" methods in the application. At runtime, if the methods are not already executing in a transaction, TXBEANS will automatically begin and commit the required transactions.

Code Fragment 8 TXBEANS Declarative TxBeans Transactions

```
ProxyFactory  proxyFactory = ProxyFactory.getSingleton();
boolean       logMethodExecution = true;
PlaceOrder    placeOrder = (PlaceOrder) proxyFactory.get
  (new PlaceOrderImpl(), logMethodExecution);
placeOrder.createOrder(...);
placeOrder.placeOrder();
```

C. Implementation

The two main constructs in the TXBEANS implementation are the *FileDataBase* and the *XmlDataBase*.

A FileDataBase:

- Manages a single file.
- Provides locking within the thread of one JVM.
- Provides locking across JVMs on a single machine by using a `java.nio.Channel` class.
- Uses a Log file to implement atomic writes.

A XmlDataBase:

- Manages a single XML document that is stored in a single file. It thus builds on (and hides) the FileDataBase API.
- Exposes a document as the top-level XMLBeans XmlObject construct

Listing 1 shows the relative size of the TXBEANS and XMLBeans jar files. (Note that `oatsutils` and `txnengine` contain many classes that are not needed for TXBEANS, and could potentially be reduced made even smaller.)

```
===========================   TxBeans ================
       9865       jta.jar
      91100       oatsutils.jar
      60547       txbeans.jar
      52219       txnengine.jar

===========================   XMLBeans ================
      23630       jsr173_1.0_api.jar
    2289378       saxon8.jar
    2635867       xbean.jar
       5168       xbean_xpath.jar
```

Listing 1: TXBEANS and XMLBeans Code Size (Jar Files in Bytes)

III. Intermittently Disconnected XMLBeans Applications

Currently, an XMLBeans application that executes on a disconnected device, must hand-code function to propagate the modified document to the version that resides on the server. Ideally, such propagation should be transactional: i.e., maintaining the transactional boundaries of the work performed on the disconnected device *and* the work that was concurrently performed on the server. TXBEANS provides generic middleware for this task and thus transparently supports intermittently disconnected XMLBeans applications. In contrast to a "data-replication" approach, we use the method-based approach discussed in [4][3].

In order for developers to exploit this TXBEANS function, developers must provide one or more Java interfaces with the methods that are to be (1) logged on client and (2) replayed on the server. The application must then get instances of these interfaces *via* the "ProxyFactory" singleton's `get` method, passing it an object (provided by the developer) that implements the required interface. Nothing else is done by the developer! Whether in connected or disconnected mode, methods are invoked in the same fashion. TXBEANS automatically supplies (optional transactional) and logging behavior for each method method invocation. Because TXBEANS uses Java's dynamic proxy support, no code generation is required.

The processing flow for an intermittently disconnected application running on top of the TXBEANS middleware is the following:

1. Client device copies XML documents in preparation for disconnected operation.
2. Client executes one or more XMLBeans applications that access XML documents. TXBEANS provides automatic transaction begin/commit (begin/rollback when exceptions occur). In contrast to the approach described in Section II, application code is *not* modified to specify transaction behavior. Instead, a developer specifies which classes should be TXBEANS-enhanced, and the TXBEANS middleware automatically provides the required transactional behavior on a per-method basis. In addition, TXBEANS automatically logs the execution of top-level transactions for subsequent replay on the server. The log records are stored in an TXBEANS XML database.
3. Upon reconnection to the server:
   (a) The client accesses its set of logged methods from the `com.ibm.txbeans.methodlog.MethodLogger` singleton
   (b) The client transmits the logged methods to the server by transmitting them to the (`com.ibm.txbeans.s`
   server middleware.
   (c) The `SyncServlet` replays the client's logged methods against the server's master copy of the XML documents. This automatically propagates the client's disconnected work to the server.

References

[1] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco, CA, USA, 1993.
[2] Java Transaction API. http://java.sun.com/products/jta, 2006.
[3] A. Leff and J. T. Rayfield. Programming model alternatives for disconnected business applications. *IEEE Internet Computing*, 10(3):50–57, May/June 2006.
[4] A. Leff and J. T. Rayfield. Programming models and synchronization techniques for disconnected business applications. *Advances in Computers*, 67:85–130, 2006.
[5] XMLBeans. http://xmlbeans.apache.org, 2006.
[6] XMLBeans Documentation. http://xmlbeans.apache.org/documentation/index.html, 2006.
[7] Document object model (DOM). http://www.w3.org/TR/DOM-Level-2-Core/, 2006.
[8] XMLSchema. http://www.w3.org/XML/Schema, 2006. See also 'XML Schema' published by O'Reilly.