Understanding JTA—the Java® Transaction API

Introduction

The Java® Transaction API (JTA) allows applications to perform distributed transactions, that is, transactions that access and update data on two or more networked computer resources. The JTA specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system: the application, the application server, and the resource manager that controls access to the shared resources affected by the transactions. This document provides an overview of that process and how the DataDirect Connect® *for* JDBC® drivers relate to it.

A transaction defines a logical unit of work that either completely succeeds or produces no result at all. A *distributed transaction* is simply a transaction that accesses and updates data on two or more networked resources, and therefore must be coordinated among those resources. In this document, we are concerned primarily with transactions that involve relational database systems.

The components involved in the distributed transaction processing (DTP) model that are relevant to our discussion are:

- The application
- The application server
- The transaction manager
- The resource adapter
- The resource manager

In the following sections, we describe these components and their relationship to JTA and database access.

Accessing Databases

It is best to think of the components involved in distributed transactions as independent *processes*, rather than in terms of location on a particular computer. Several of the components may reside on one machine, or they may be spread among several machines. The diagrams in the following examples may show a component on a particular computer, but the relationship among the processes is the primary consideration.

The Simplest Case: Application to Database Local Transactions

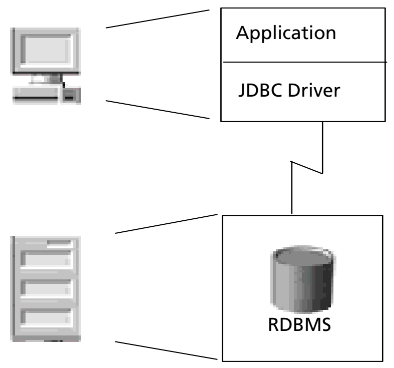
The simplest form of relational database access involves only the application, a resource manager, and a resource adapter. The application is simply the end-user access point to send requests to, and obtain data from, a database.

The resource manager in our discussion is a relational database management system (RDBMS), such as Oracle or SQL Server. All of the actual database management is handled by this component.

The resource adapter is the component that is the communications channel, or request translator, between the "outside world," in this case the application, and the resource manager. For our discussion, this is a JDBC driver.

The following description is of a *resource manager local transaction*, that is, one transaction that is confined to a single, specific enterprise database.

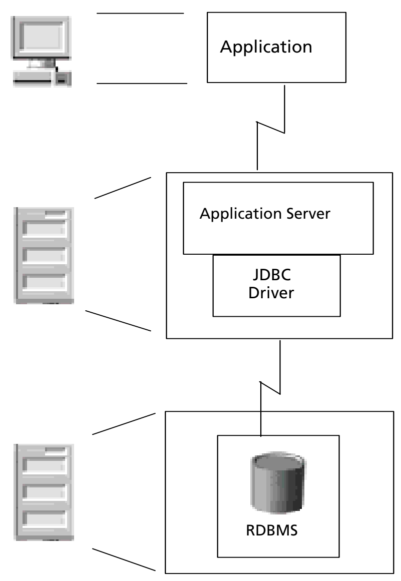
The application sends a request for data to the JDBC driver, which then translates the request and sends it across the network to the database. The database returns the data to the driver, which then translates the result to the application, as illustrated in the following diagram:



This example illustrates the basic flow of information in a simplified system; however, the enterprise of today uses application servers, which adds another component to the process.

Application Servers

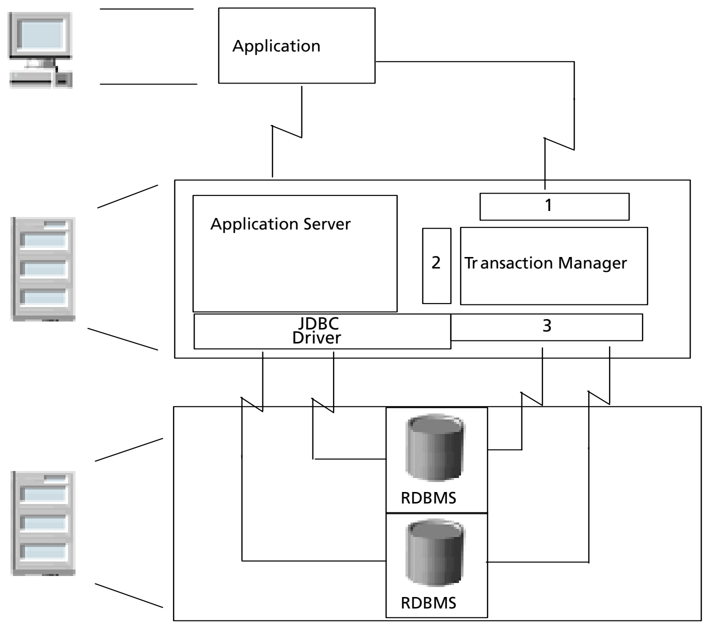
The application server is another component of the transaction process that is addressed by the JTA. Application servers handle the bulk of application operations and take some of the load off of the end-user application. Building on the preceding example, we see that the application server adds another process tier to the transaction:



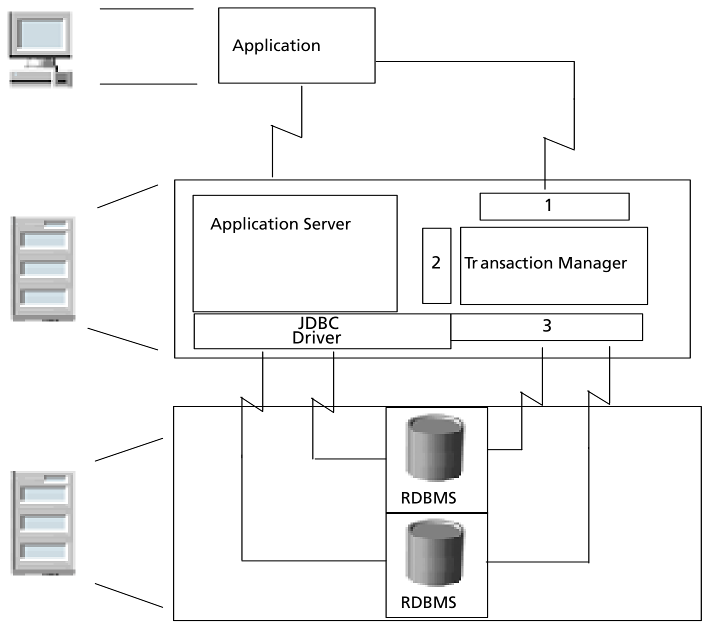
Up to this point, our examples illustrate a single, local transaction and describe four out of the five components of the distributed transaction model. The fifth component, the transaction manager, comes into consideration only when transactions are to be distributed.

Distributed Transactions and the Transaction Manager

As we stated previously, a distributed transaction is a transaction that accesses and updates data on two or more networked resources. These resources could consist of several different RDBMSs housed on a single sever, for example, Oracle, Microsoft SQL Server, and Sybase; or they could include several instances of a single type of database residing on a number of different servers. In any case, a distributed transaction involves coordination among the various resource managers. This coordination is the function of the transaction manager.



The transaction manager is responsible for making the final decision either to *commit* or *rollback* any distributed transaction. A commit decision should lead to a successful transaction; rollback leaves the data in the database unaltered. JTA specifies standard Java interfaces between the transaction manager and the other components in a distributed transaction: the application, the application server, and the resource managers. This relationship is illustrated in the following diagram:



The numbered boxes around the transaction manager correspond to the three interface portions of JTA:

1—UserTransaction—The `javax.transaction.UserTransaction` interface provides the application the ability to control transaction boundaries programmatically. The `javax.transaction.UserTransaction` method starts a global transaction and associates the transaction with the calling thread.

2—Transaction Manager—The `javax.transaction.TransactionManager` interface allows the application server to control transaction boundaries on behalf of the application being managed.

3—XAResource—The `javax.transaction.xa.XAResource` interface is a Java mapping of the industry standard XA interface based on the X/Open CAE Specification (Distributed Transaction Processing: The XA Specification).

Notice that a critical link is support of the XAResource interface by the JDBC driver. The JDBC driver must support both normal JDBC interactions, through the application and/or the application server, as well as the XAResource portion of JTA. DataDirect Connect *for* JDBC drivers provide this support.

Developers of code at the application level should not be concerned about the details of distributed transaction management. This is the job of the distributed transaction infrastructure—the application server, the transaction manager, and the JDBC driver.

The only caveat for application code is that it should not invoke a method that would affect the boundaries of a transaction while the connection is in the scope of a distributed transaction. Specifically, an application should not call the `Connection` methods `commit`, `rollback`, and `setAutoCommit(true)` because they would interfere with the infrastructure's management of the distributed transaction.

The Distributed Transaction Process

The transaction manager is the primary component of the distributed transaction infrastructure; however, the JDBC driver and application server components should have the following characteristics:

- The driver should implement the JDBC 2.0 API (including the Optional Package interfaces `XADataSource` and `XAConnection)` or higher and the JTA interface `XAResource`.
- The application server should provide a `DataSource` class that is implemented to interact with the distributed transaction infrastructure and a connection pooling module (for improved performance).

The first step of the distributed transaction process is for the application to send a request for the transaction to the transaction manager. Although the final commit/rollback decision treats the transaction as a single logical unit, there can be many *transaction branches* involved. A transaction branch is associated with a request to each resource manager involved in the distributed transaction. Requests to three different RDBMSs, therefore, require three transaction branches. Each transaction branch must be committed or rolled back by the local resource manager. The transaction manager controls the boundaries of the transaction and is responsible for the final decision as to whether or not the total transaction should commit or rollback. This decision is made in two phases, called the Two-Phase Commit Protocol.

In the first phase, the transaction manager polls all of the resource managers (RDBMSs) involved in the distributed transaction to see if each one is ready to commit. If a resource manager cannot commit, it responds negatively and rolls back its particular part of the transaction so that data is not altered.

In the second phase, the transaction manager determines if any of the resource managers have responded negatively, and, if so, rolls back the whole transaction. If there are no negative responses, the translation manager commits the whole transaction, and returns the results to the application.

Developers of transaction manager code must be conversant with all three interfaces of JTA: UserTransaction, TransactionManager, and XAResource, which are described in the Sun Java Transaction API (JTA) specification. The *JDBC API Tutorial and Reference, Third Edition* is also a useful reference. JDBC driver developers need only be concerned with the XAResource interface. This interface is a Java mapping of the industry standard X/Open XA protocol that allows a resource manager to participate in a transaction. The component of the driver connected with the XAResource interface is responsible for "translating" between the transaction manager and the resource manager. The following section provides examples of XAResource calls.

The JDBC Driver and XAResource

To simplify the explanation of XAResource, these examples illustrate how an application would use JTA when there is no application server and transaction manager involved. Basically, the application in these examples is also acting as application server and transaction manager. Most enterprises use transaction managers and application servers because they manage distributed transactions much more efficiently than an application can. By following these examples, however, an application developer can test the robustness of JTA support in a JDBC driver. Some examples may not work for a particular database because of inherent problems associated with that database.

Before using JTA, you must first implement an Xid class for identifying transactions (this would normally be done by the transaction manager). The Xid contains three elements: formatID, gtrid (global transaction ID), and bqual (branch qualifier ID).

The formatID is usually zero, meaning that you are using the OSI CCR (Open Systems Interconnection Commitment, Concurrency, and Recovery standard) for naming. If you are using another format, the formatID should be greater than zero. A value of –1 means that the Xid is null.

The gtrid and bqual can each contain up to 64 bytes of binary code to identify the global transaction and the branch transaction, respectively. The only requirement is that the gtrid and bqual taken together must be globally unique. Again, this can be achieved by using the naming rules specified in the OSI CCR.

The following example illustrates implementation of an Xid:

```
import javax.transaction.xa.*;
public class MyXid implements Xid
{
    protected int formatId;
    protected byte gtrid[];
    protected byte bqual[];

    public MyXid()
    {
    }

    public MyXid(int formatId, byte gtrid[], byte bqual[])
    {
        this.formatId = formatId;
        this.gtrid = gtrid;
        this.bqual = bqual;
    }

    public int getFormatId()
    {
        return formatId;
    }

    public byte[] getBranchQualifier()
    {
        return bqual;
    }

    public byte[] getGlobalTransactionId()
    {
        return gtrid;
    }

}
```

Second, you need to create a data source for the database that you are using:

```
public DataSource getDataSource()
    throws SQLException
{
    SQLServerDataSource xaDS = new
        com.ddtek.jdbc.sqlserver.SQLServerDriver.SQLServerDataSource();
    xaDS.setDataSourceName("SQLServer");
    xaDS.setServerName("server");
    xaDS.setPortNumber(1433);
    xaDS.setSelectMethod("cursor");
    return xaDS;
}
```

Example 1—This example uses the two-phase commit protocol to commit one transaction branch:

```
XADataSource xaDS;
XAConnection xaCon;
XAResource   xaRes;
Xid          xid;
Connection   con;
Statement    stmt;
int          ret;

xaDS = getDataSource();
xaCon = xaDS.getXAConnection("jdbc_user", "jdbc_password");
xaRes = xaCon.getXAResource();

con = xaCon.getConnection();
stmt = con.createStatement();

xid = new MyXid(100, new byte[]{0x01}, new byte[]{0x02});

try {
    xaRes.start(xid, XAResource.TMNOFLAGS);
    stmt.executeUpdate("insert into test_table values (100)");
    xaRes.end(xid, XAResource.TMSUCCESS);

    ret = xaRes.prepare(xid);
    if (ret == XAResource.XA_OK) {
        xaRes.commit(xid, false);
    }
}
catch (XAException e) {
    e.printStackTrace();
}
finally {
    stmt.close();
    con.close();
    xaCon.close();
}
```

Because the initialization code is the same, or similar, for all the examples, code that is significantly different is represented from this point forward in this document.

Example 2—This example, similar to Example 1, illustrates a rollback:

```
xaRes.start(xid, XAResource.TMNOFLAGS);
stmt.executeUpdate("insert into test_table values (100)");
xaRes.end(xid, XAResource.TMSUCCESS);

ret = xaRes.prepare(xid);
if (ret == XAResource.XA_OK) {
    xaRes.rollback(xid);
}
```

Example 3—This example shows how a distributed transaction branch suspends, lets the same connection do a local transaction, and then resumes the branch later. The two-phase commit actions of distributed transaction do not affect the local transaction.

```
xid = new MyXid(100, new byte[]{0x01}, new byte[]{0x02});

xaRes.start(xid, XAResource.TMNOFLAGS);
stmt.executeUpdate("insert into test_table values (100)");
xaRes.end(xid, XAResource.TMSUSPEND);

// This update is done outside of transaction scope, so it
// is not affected by the XA rollback.
stmt.executeUpdate("insert into test_table2 values (111)");

xaRes.start(xid, XAResource.TMRESUME);
stmt.executeUpdate("insert into test_table values (200)");
xaRes.end(xid, XAResource.TMSUCCESS);

ret = xaRes.prepare(xid);
if (ret == XAResource.XA_OK) {
    xaRes.rollback(xid);
}
```

Example 4—This example illustrates how one XA resource can be shared among different transactions. Two transaction branches are created, but they do not belong to the same distributed transaction. JTA allows the XA resource to do a two-phase commit on the first branch even though the resource is still associated with the second branch.

```
xid1 = new MyXid(100, new byte[]{0x01}, new byte[]{0x02});
xid2 = new MyXid(100, new byte[]{0x11}, new byte[]{0x22});

xaRes.start(xid1, XAResource.TMNOFLAGS);
stmt.executeUpdate("insert into test_table1 values (100)");
xaRes.end(xid1, XAResource.TMSUCCESS);

xaRes.start(xid2, XAResource.TMNOFLAGS);

// Should allow XA resource to do two-phase commit on
// transaction 1 while associated to transaction 2
ret = xaRes.prepare(xid1);
if (ret == XAResource.XA_OK) {
    xaRes.commit(xid1, false);
}

stmt.executeUpdate("insert into test_table2 values (200)");
xaRes.end(xid2, XAResource.TMSUCCESS);

ret = xaRes.prepare(xid2);
if (ret == XAResource.XA_OK) {
    xaRes.rollback(xid2);
}
```

Example 5—This example illustrates how transaction branches on different connections can be joined as a single branch if they are connected to the same resource manager. This feature improves distributed transaction efficiency because it reduces the number of two-phase commit processes. Two XA connections to the same database server are created. Each connection creates its own XA resource, regular JDBC connection, and statement. Before the second XA resource starts a transaction branch, it checks to see if it uses the same resource manager as the first XA resource uses. If this is case, as in this example, it joins the first branch created on the first XA connection instead of creating a new branch. Later, the transaction branch can be prepared and committed using either XA resource.

```
xaDS = getDataSource();

xaCon1 = xaDS.getXAConnection("jdbc_user", "jdbc_password");
xaRes1 = xaCon1.getXAResource();
con1 = xaCon1.getConnection();
stmt1 = con1.createStatement();

xid1 = new MyXid(100, new byte[]{0x01}, new byte[]{0x02});
xaRes1.start(xid1, XAResource.TMNOFLAGS);
stmt1.executeUpdate("insert into test_table1 values (100)");
xaRes1.end(xid, XAResource.TMSUCCESS);

xaCon2 = xaDS.getXAConnection("jdbc_user", "jdbc_password");
xaRes2 = xaCon2.getXAResource();
con2 = xaCon2.getConnection();
stmt2 = con2.createStatement();

if (xaRes2.isSameRM(xaRes1)) {
    xaRes2.start(xid1, XAResource.TMJOIN);
    stmt2.executeUpdate("insert into test_table2 values (100)");
    xaRes2.end(xid1, XAResource.TMSUCCESS);
}
else {
    xid2 = new MyXid(100, new byte[]{0x01}, new byte[]{0x03});
    xaRes2.start(xid2, XAResource.TMNOFLAGS);
    stmt2.executeUpdate("insert into test_table2 values (100)");
    xaRes2.end(xid2, XAResource.TMSUCCESS);
    ret = xaRes2.prepare(xid2);
    if (ret == XAResource.XA_OK) {
        xaRes2.commit(xid2, false);
    }
}

ret = xaRes1.prepare(xid1);
if (ret == XAResource.XA_OK) {
    xaRes1.commit(xid1, false);
}
```

Example 6—This example shows how to recover prepared or heuristically completed transaction branches during failure recovery. It first tries to roll back each branch; if it fails, it tries to tell resource manager to discard knowledge about the transaction.

```
MyXid[] xids;

xids = xaRes.recover(XAResource.TMSTARTRSCAN | XAResource.TMENDRSCAN);
for (int i=0; xids!=null && i<xids.length; i++) {
    try {
        xaRes.rollback(xids[i]);
    }
    catch (XAException ex) {
        try {
            xaRes.forget(xids[i]);
        }
        catch (XAException ex1) {
            System.out.println("rollback/forget failed: " +
ex1.errorCode);
            }
     }
}
```

Conclusion

Providing JTA support in a JDBC driver greatly increases data access power. The DataDirect Connect *for* JDBC drivers provide this support. In combination with the other components of the distributed transaction process, DataDirect drivers enhance the capability, speed, and efficiency of the modern enterprise.

References

Cheung & Matena, *Java Transaction API (JTA)*, 1999, Sun Microsystems, Inc.

Maydene Fisher, Jon Ellis, and Jonathan Bruce, *JDBC API Tutorial and Reference, Third Edition*, 2003, Addison-Wesley.

X/Open CAE Specification, Distributed Transaction Processing: The XA Specification, 1991, The X/Open Company.

We welcome your feedback! Please send any comments concerning documentation, including suggestions for other topics that you would like to see, to:

docgroup@datadirect.com

FOR MORE INFORMATION

800-876-3101

Worldwide Sales

Belgium (French)	0800 12 045
Belgium (Dutch)	0800 12 046
France	0800 911 454
Germany	0800 181 78 76
Japan	0120.20.9613
Netherlands	0800 022 0524
United Kingdom	0800 169 19 07
United States	800 876 3101

DataDirect
TECHNOLOGIES

DataDirect Technologies is focused on data access, enabling software developers at both packaged software vendors and in corporate IT departments to create better applications faster. DataDirect Technologies offers the most comprehensive, proven line of data connectivity components available anywhere. Developers worldwide depend on DataDirect Technologies to connect their applications to an unparalleled range of data sources using standards-based interfaces such as ODBC, JDBC and ADO.NET, as well as cutting-edge XML query technologies. More than 250 leading independent software vendors and thousands of enterprises rely on DataDirect Technologies to simplify and streamline data connectivity. DataDirect Technologies is an operating company of Progress Software Corporation (Nasdaq: PRGS).

www.datadirect.com