Example of a distributed transaction that uses JTA methods

Db2 11.1

Distributed transactions typically involve multiple connections to the same data source or different data sources, which can include data sources from different manufacturers.

The best way to demonstrate distributed transactions is to contrast them with local transactions. With local transactions, a JDBC application makes changes to a database permanent and indicates the end of a unit of work in one of the following ways:

- By calling the `Connection.commit` or `Connection.rollback` methods after executing one or more SQL statements
- By calling the `Connection.setAutoCommit(true)` method at the beginning of the application to commit changes after every SQL statement

Figure 1 outlines code that executes local transactions.

Related topics:

Setting the transaction timeout value for and XAResource instance

Do you want to...

Open a ticket and downlo
fixes at the **IBM Support F**

Figure 1. Example of a local transaction

```
con1.setAutoCommit(false);  // Set autocommit off
// execute some SQL
…
con1.commit();              // Commit the transaction
// execute some more SQL
…
con1.rollback();            // Roll back the transaction
con1.setAutoCommit(true);   // Enable commit after every SQL statement
…
// Execute some more SQL, which is automatically committed after
// every SQL statement.
```

In contrast, applications that participate in distributed transactions cannot call the `Connection.commit`, `Connection.rollback`, or `Connection.setAutoCommit(true)` methods within the distributed transaction. With distributed transactions, the `Connection.commit` or `Connection.rollback` methods do not indicate transaction boundaries. Instead, your applications let the application server manage transaction boundaries.

Figure 2 demonstrates an application that uses distributed transactions. While the code in the example is running, the application server is also executing other EJBs that are part of this same distributed transaction. When all EJBs have called `utx.commit()`, the entire distributed transaction is committed by the application server. If any of the EJBs are unsuccessful, the application server rolls back all the work done by all EJBs that are associated with the distributed transaction.

Figure 2. Example of a distributed transaction under an application server

```
javax.transaction.UserTransaction utx;
// Use the begin method on a UserTransaction object to indicate
// the beginning of a distributed transaction.
utx.begin();
…
// Execute some SQL with one Connection object.
// Do not call Connection methods commit or rollback.
…
// Use the commit method on the UserTransaction object to
// drive all transaction branches to commit and indicate
// the end of the distributed transaction.

utx.commit();
…
```

Figure 3 illustrates a program that uses JTA methods to execute a distributed transaction. This program acts as the transaction manager and a transactional application. Two connections to two different data sources do SQL work under a single distributed transaction.

Figure 3. Example of a distributed transaction that uses the JTA

```
class XASample
{
  javax.sql.XADataSource xaDS1;
  javax.sql.XADataSource xaDS2;
  javax.sql.XAConnection xaconn1;
  javax.sql.XAConnection xaconn2;
  javax.transaction.xa.XAResource xares1;
  javax.transaction.xa.XAResource xares2;
  java.sql.Connection conn1;
  java.sql.Connection conn2;

  public static void main (String args []) throws java.sql.SQLException
  {
    XASample xat = new XASample();
    xat.runThis(args);
  }
  // As the transaction manager, this program supplies the global
  // transaction ID and the branch qualifier. The global
  // transaction ID and the branch qualifier must not be
  // equal to each other, and the combination must be unique for
  // this transaction manager.
  public void runThis(String[] args)
  {
    byte[] gtrid = new byte[] { 0x44, 0x11, 0x55, 0x66 };
    byte[] bqual = new byte[] { 0x00, 0x22, 0x00 };
    int rc1 = 0;
    int rc2 = 0;

    try
    {

      javax.naming.InitialContext context = new javax.naming.InitialContext();
      /*
        * Note that javax.sql.XADataSource is used instead of a specific
        * driver implementation such as com.ibm.db2.jcc.DB2XADataSource.
        */
      xaDS1 = (javax.sql.XADataSource)context.lookup("checkingAccounts");
      xaDS2 = (javax.sql.XADataSource)context.lookup("savingsAccounts");

      // The XADatasource contains the user ID and password.
      // Get the XAConnection object from each XADataSource
      xaconn1 = xaDS1.getXAConnection();
```

```
        xaconn2 = xaDS2.getXAConnection();

        // Get the java.sql.Connection object from each XAConnection
        conn1 = xaconn1.getConnection();
        conn2 = xaconn2.getConnection();

        // Get the XAResource object from each XAConnection
        xares1 = xaconn1.getXAResource();
        xares2 = xaconn2.getXAResource();
        // Create the Xid object for this distributed transaction.
        // This example uses the com.ibm.db2.jcc.DB2Xid implementation
        // of the Xid interface. This Xid can be used with any JDBC driver
        // that supports JTA.
        javax.transaction.xa.Xid xid1 =
                                new com.ibm.db2.jcc.DB2Xid(100, gtrid, bqual);

        // Start the distributed transaction on the two connections.
        // The two connections do NOT need to be started and ended together.
        // They might be done in different threads, along with their SQL operations.
        xares1.start(xid1, javax.transaction.xa.XAResource.TMNOFLAGS);
        xares2.start(xid1, javax.transaction.xa.XAResource.TMNOFLAGS);
…
        // Do the SQL operations on connection 1.
        // Do the SQL operations on connection 2.
…
        // Now end the distributed transaction on the two connections.
        xares1.end(xid1, javax.transaction.xa.XAResource.TMSUCCESS);
        xares2.end(xid1, javax.transaction.xa.XAResource.TMSUCCESS);

        // If connection 2 work had been done in another thread,
        // a thread.join() call would be needed here to wait until the
        // connection 2 work is done.

        try
        { // Now prepare both branches of the distributed transaction.
          // Both branches must prepare successfully before changes
          // can be committed.
          // If the distributed transaction fails, an XAException is thrown.
          rc1 = xares1.prepare(xid1);
          if(rc1 == javax.transaction.xa.XAResource.XA_OK)
          { // Prepare was successful. Prepare the second connection.
            rc2 = xares2.prepare(xid1);
            if(rc2 == javax.transaction.xa.XAResource.XA_OK)
            { // Both connections prepared successfully and neither was read-only.
              xares1.commit(xid1, false);
              xares2.commit(xid1, false);
            }
            else if(rc2 == javax.transaction.xa.XAException.XA_RDONLY)
            { // The second connection is read-only, so just commit the
```

```
        // first connection.
        xares1.commit(xid1, false);
      }
    }
    else if(rc1 == javax.transaction.xa.XAException.XA_RDONLY)
    { // SQL for the first connection is read-only (such as a SELECT).
      // The prepare committed it. Prepare the second connection.
      rc2 = xares2.prepare(xid1);
      if(rc2 == javax.transaction.xa.XAResource.XA_OK)
      { // The first connection is read-only but the second is not.
        // Commit the second connection.
        xares2.commit(xid1, false);
      }
      else if(rc2 == javax.transaction.xa.XAException.XA_RDONLY)
      { // Both connections are read-only, and both already committed,
        // so there is nothing more to do.
      }
    }
  }      catch (javax.transaction.xa.XAException xae)
  { // Distributed transaction failed, so roll it back.
    // Report XAException on prepare/commit.
    System.out.println("Distributed transaction prepare/commit failed. " +
                       "Rolling it back.");
    System.out.println("XAException error code = " + xae.errorCode);
    System.out.println("XAException message = " + xae.getMessage());
    xae.printStackTrace();
    try
    {
      xares1.rollback(xid1);
    }
    catch (javax.transaction.xa.XAException xae1)
    { // Report failure of rollback.
      System.out.println("distributed Transaction rollback xares1 failed");
      System.out.println("XAException error code = " + xae1.errorCode);
      System.out.println("XAException message = " + xae1.getMessage());
    }
    try
    {
      xares2.rollback(xid1);
    }
    catch (javax.transaction.xa.XAException xae2)
    { // Report failure of rollback.
      System.out.println("distributed Transaction rollback xares2 failed");
      System.out.println("XAException error code = " + xae2.errorCode);
      System.out.println("XAException message = " + xae2.getMessage());
    }
  }

  try
  {
```

```
      }
        conn1.close();
        xaconn1.close();
      }
      catch (Exception e)
      {
        System.out.println("Failed to close connection 1: " + e.toString());
        e.printStackTrace();
      }
      try
      {
        conn2.close();
        xaconn2.close();
      }
      catch (Exception e)
      {
        System.out.println("Failed to close connection 2: " + e.toString());
        e.printStackTrace();
      }
    }
    catch (java.sql.SQLException sqe)
    {
      System.out.println("SQLException caught: " + sqe.getMessage());
      sqe.printStackTrace();
    }
    catch (javax.transaction.xa.XAException xae)
    {
      System.out.println("XA error is " + xae.getMessage());
      xae.printStackTrace();
    }
    catch (javax.naming.NamingException nme)
    {
      System.out.println(" Naming Exception: " + nme.getMessage());
    }
```