X/Open DTP (Local Environment)

The X/Open Distributed Transaction Processing (DTP) model allows application programs to share resources provided by multiple resource managers and to coordinate their work in global transactions using the two phase commit protocol.

The X/Open DTP model distinguishes three functional components involved in distributed transaction processing, namely the Application Program (AP), the Transaction Manager (TM), and Resource Managers (RMs) as shown in Figure 1. The AP defines the start and end of global transactions, accesses resources within transaction boundaries, and normally makes the decision whether to commit or roll back the transaction. The transaction manager (TM) assigns unique identifiers (XID) to transactions, monitors their progress and coordinates participating Resource Managers during transaction completion (commit or rollback). A Resource Manager (RM) manages a shared resource that may be accessed by the AP using services that the RM provides. Examples of RMs are database management systems (DBMSs), file access methods such as X/Open ISAM, or print servers.

\[
\text{tx\_begin( )} \quad \text{Begin a global transaction.}
\]
Middleware for Heterogeneous and Distributed Information Systems

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xa_start( )</td>
<td>Start or resume a transaction branch - associate an XID with future work that the thread requests of the RM.</td>
</tr>
<tr>
<td>ax_reg( )</td>
<td>Register an RM with a TM to join a transaction.</td>
</tr>
<tr>
<td>xa_prepare( )</td>
<td>Ask the RM to prepare to commit a transaction branch.</td>
</tr>
<tr>
<td>xa_commit( )</td>
<td>Tell the RM to commit a transaction branch.</td>
</tr>
<tr>
<td>xa_rollback( )</td>
<td>Tell the RM to roll back a transaction branch.</td>
</tr>
</tbody>
</table>

Table 1: Functions of the TX interface (not complete)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tx_commit()</td>
<td>Commit a global transaction.</td>
</tr>
<tr>
<td>tx_rollback()</td>
<td>Roll back a global transaction.</td>
</tr>
</tbody>
</table>

Table 2: Functions of the XA interface (not complete)

The components of the X/Open DTP model interact by means of standardized interfaces. The TX (Transaction Demarcation) interface\(^1\) allows the AP to call the TM to demarcate global transactions and direct their completion. An (incomplete) list of the functions of the TX interface is found in Table 1.

The XA interface\(^2\) is a bidirectional interface between the TM and the RM. It is used by the RM to join a transaction managed by the TM. It is further used by the TM to coordinate multiple RMs during transaction completion. An (incomplete) list of the functions of the XA interface is found in Table 2.

**X/Open DTP (Distributed Environment)**

The X/Open DTP model has been generalized to support transactions in distributed environments. That is, multiple TM domains may be involved in a global transaction. A dedicated component, referred to as Communication Resource Manager (CRM), is used to propagate transaction information across TM domains as shown in Figure 2.

![Figure 2: Functional Components and Interfaces](http://www.opengroup.org/onlinepubs/009680699/toc.pdf)

---

The XA interface has been extended to allow the CRM to propagate coordination information to subordinate TMs. The revised interface is referred to as XA+ interface\(^3\). An (incomplete) list of the functions of the XA+ interface is found in Table 2.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ax_commit( )</td>
<td>Propagate transaction branch commitment to a transaction manager.</td>
</tr>
<tr>
<td>ax_prepare( )</td>
<td>Propagate transaction branch prepare to commit to a transaction manager.</td>
</tr>
<tr>
<td>ax_rollback ( )</td>
<td>Propagate transaction rollback to a transaction manager.</td>
</tr>
<tr>
<td>ax_start( )</td>
<td>Notify the transaction manager to propagate or resume a transaction branch association with this thread of control.</td>
</tr>
</tbody>
</table>

Table 3: Functions of the XA+ interface (not complete)

Figure 3 illustrates the interactions between the functional components during transaction processing in a distributed environment. The application program begins a global transaction and accesses both, a local and a remote RM. The application then asks the local TM to commit the transaction. No errors occur during the 2PC and hence, the transaction completes successfully.

---

A transaction identifier (XID) is assigned by the TM. It represents the unique relationship between an AP, the work it issues to RMs, and the global transaction which the TM manages on behalf of the AP. The XID lets the TM track and coordinate all of the work associated with a global transaction.

Figure 3: Sample distributed transaction involving two TM domains (simplified representation)
Application server middleware

1) Shift from a client/server architecture to a 3-tier architecture.
   - The application does no longer require a JDBC driver.
   - The location of the databases is transparent to the application.

2) Each database has to be registered to the application server to be accessible by application programs. The following explanation was taken from the Sun J2EE Tutorial (p. 1109)

   "In the JDBC API, databases are accessed via DataSource objects. A DataSource has a set of properties that identify and describe the real world data source that it represents. These properties include information such as the location of the database server, the name of the database, the network protocol to use to communicate with the server, and so on. In the Application Server, a data source is called a JDBC resource.

   Applications access a data source using a connection, and a DataSource object can be thought of as a factory for connections to the particular data source that the DataSource instance represents. In a basic DataSource implementation, a call to the getConnection method returns a connection object that is a physical connection to the data source.

   If a DataSource object is registered with a JNDI naming service, an application can use the JNDI API to access that DataSource object, which can then be used to connect to the data source it represents."
3) Creating a connection using JDBC:

```java
try {
    Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
    Connection connection = DriverManager.getConnection(url, username, password);
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (SQLException e) {
    e.printStackTrace();
}
```

Obtaining a connection using a JNDI lookup:

```java
Connection conn=null;
try {
    Context ctx=new InitialContext();
    Object datasourceRef=ctx.lookup("java:comp/env/jdbc/mydatasource");
    DataSource ds=(Datasource)datasourceRef;
    conn=ds.getConnection();
} catch(Exception e) {
    e.printStackTrace();
}
```

4) When a JDBC database gateway is used, transaction are demarcated using the methods of the Connection interface:

```java
    connection.setAutoCommit(false);
    connection.commit();
    connection.rollback();
```

In J2EE explicit transaction demarcation is performed by means of the JTA User-Transaction interface.

```java
    userTransaction.begin();
    userTransaction.commit();
    userTransaction.rollback();
    userTransaction.setRollbackOnly();
```

5) Advantages:

- The database location and type is transparent to the client. Shifting to another database does not require any modification of the application code.
- Connection pooling
- Distributed transactions are feasible

Drawbacks:
• Communication overhead
• Application server middleware is by no means a light-weight solution (may be an overhead for rather small applications)

Implicit Transaction Demarcation
The following explanation has been taken from See J2EE Transaction Frameworks, Part 3⁴:

“Almost all enterprise beans perform transactional work by accessing some form of resource like an EIS or an RDBMS via JDBC. The recommended use of different transaction attributes for enterprise beans taking part in transactional work is given below.

• The default recommended choice for a transaction attribute is Required since it ensures that the methods of an enterprise bean are invoked under a JTA transaction. In addition, enterprise beans with the Required transaction attribute can be easily composed to perform work under the scope of a single JTA transaction.

• If an enterprise bean method needs to commit its results unconditionally, whether or not a transaction is already in progress, the RequiresNew transaction attribute is recommended. An example of this requirement is a bean method that performs logging. This bean method should be invoked with RequiresNew transaction attribute so that the logging records are created even if the calling client's transaction is rolled back.

• The NotSupported transaction attribute can be used when the resource being accessed for a transaction cannot be part of a JTA transaction. This is the case when the resource manager responsible for the transaction is not supported by the J2EE product. For example, if a bean method is invoking an operation on an enterprise resource planning system that is not integrated with the J2EE server, the server has no control over that system's transactions. In this case, it is best to set the transaction attribute of the bean to be NotSupported to clearly indicate that the enterprise resource planning system is not accessed within a JTA transaction.

• Using the transaction attribute Supports isn’t recommended. An enterprise bean with this attribute would have transactional behavior that differed depending on whether the caller is associated with a transaction context, possibly leading to a violation of the ACID rules for transactions.

• The transaction attributes Mandatory and Never can be used when it is necessary to verify the transaction association of the calling client. They reduce the composability of a component by putting constraints on the calling client's transaction context.”

⁴ available at http://www.onjava.com/pub/a/onjava/2001/06/06/j2ee_trans.html
Bean1
a() { none | T1
... bean2.b();
... bean2.c();
...
}

Bean2
b() { T1 | T1
... bean3.d();
... bean3.e();
...
}
c() { T1 | T1
... bean4.f();
...
}

Bean3
d() { T1 | none
... }
e() { T1 | T1
... }

Bean4
f() { T1 | T1
... error!
...

5 a() { none | T1 here indicates that the caller does not run in any transaction while the called method a() creates and runs in a transaction T1.
Bean1

```java
Bean1
  a() { none | T1
      ...
      bean2.b();
      ...
      bean2.c();
      ...
  }  
```

Bean2

```java
Bean2
  b() { T1 | T2
      ...
      bean3.d();
      ...
      bean3.e();
      ...
  }
  c() { T1 | none
      ...
      bean4.f();
      ...
  }
```

Bean3

```java
Bean3
  d() { T2 | T2
      ...
  }
  e() { T2 | T3
      ...
  }
```

Bean4

```java
Bean4
  f() { none | T4
      ...
  }
```

<table>
<thead>
<tr>
<th>Required</th>
<th>RequiresNew</th>
<th>Mandatory</th>
<th>NotSupported</th>
<th>Supports</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>bean1.a()</td>
<td>bean2.b()</td>
<td>bean3.d()</td>
<td>bean2.c()</td>
<td>bean4.f()</td>
<td>bean3.e()</td>
</tr>
</tbody>
</table>