C0304 - Enlist and handle multiple JTA aware resources in Global Transactions

INDEX

1 Introduction:

A transaction is a series of operations that must be performed atomically. In other words, each operation in the series must succeed for the entire transaction to be successful. If any operation in the transaction does not succeed, the entire transaction fails. At that point, any operations which have succeeded must be "rolled back" so that the end state matches what was in place before the transaction started.

Unit-of-work:

A transaction is the execution of a set of related operations that must be completed together. This set of operations is referred to as a *unit-of-work.* A transaction is said to *commit* when it completes successfully. Otherwise it is said to *roll back*.

Example: *Fund* Transfer between bank accounts

A business transaction would be a two-step process involving subtraction (debit) from one account and addition (credit) to another account. Both operations are part of the same transaction and both must succeed in order to complete the transaction. If one of these operations fails, the account balances must be restored to their original states. Here the fund transfer operation is a unit-of-work, composed of debiting one account and crediting another.

Transaction, and a transaction manager (or a transaction processing service) simplifies construction of such enterprise level distributed applications while maintaining integrity of data in a unit of work.

2 Transaction Properties:

Transactional properties are ACID properties.

Ø **A**tomicity

Ø **C**onsistency

Ø **I**solation

Ø **D**urability

These properties are used as a way of gauging how well a particular technology and product performs a transaction.

2.1 Atomicity:

All operations of a transaction either succeed or fail as a unit. That is, there cannot be partial successes among the operations that are performed.

2.2 Consistency:

Whether a transaction succeeds or not, the data should remain in consistent state. For example, a failure in bank transfer example should not leave the account in negative number.

2.3 Isolation:

One transaction should not interfere with another. It means that each transaction should execute independently of other transactions that may be executing concurrently in the same environment. The effect of executing a set of transactions serially should be the same as that of running them concurrently.

- During the course of a transaction, intermediate (possibly inconsistent) state of the data should not be exposed to all other transactions.
- Two concurrent transactions should not be able to operate on the same data. Database management systems usually implement this feature using locking.

2.4 Durability:

A transaction that has been committed should be durable. For example, the data that has been committed should survive a server crash.

INDEX

3 Isolation Anomalies and Levels:

Depending on the behavior of your application, different isolation anomalies can occur and as a developer or deployer, you can choose appropriate isolation levels. Now the reason you choose different isolation level is because which will choose will have direct impact to overall (throughput) performance of your application..

3.1 Isolation Anomalies:

Three typical anomalies are listed below:

Ø **Dirty read**

Ø **No repeatable reads**

Ø **Phantom reads**

Dirty read:

A dirty read occurs when a transaction is allowed to read data from a row that has been modified by another running transaction and not yet committed.

- Occurs when one transaction (T2) reads data that has been modified by previously started transaction (T1), but not committed
- What happens if the T1 rolls back? T1 has incorrect data, therefore it is called ¿dirty read

No repeatable reads:

A *non-repeatable read* occurs, when during the course of a transaction, a row is retrieved twice and the values within the row differ between reads.- Occurs when one transaction (T1) reads same data twice while another transaction (T2) modifies the same data between the two reads by T1.

- T1 gets different value between the first and the second read, therefore it is called ¿nonrepeatable read¿

Phantom reads:

A *phantom read* occurs when, in the course of a transaction, two identical queries are executed, and the collection of rows returned by the second query is different from the first. For example when one transaction (T1) begins reading data and another transaction (T2) inserts to or deletes data from the table being read.

How to prevent Isolation Anomalies?

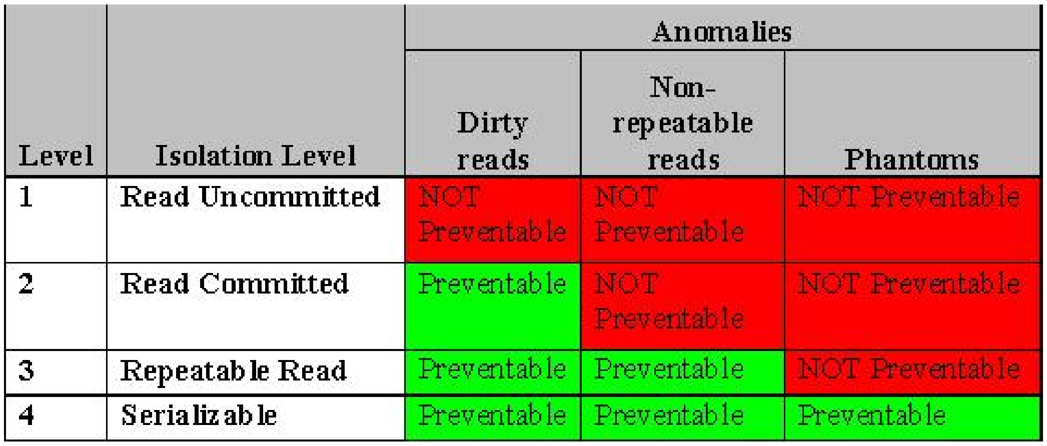
Depending on which level of database locking you impose, that is different isolation levels, different performance result will occur. So the isolation and performance requirement of your application has to be balanced out. Now database vendors have standardized the types of database locking.

3.2 Isolation Levels:

The isolation property is the one most often relaxed when attempting to maintain the highest level of isolation a DBMS usually acquires locks on data. Most DBMS's offer a number of transaction isolation levels, which control the degree of locking that, occurs when selecting data.

For many database applications, the majority of database transactions can be constructed to avoid requiring high isolation levels (e.g. SERIALIZABLE level), thus reducing the locking overhead for the system. The developer must carefully analyze database access code to ensure that any relaxation of isolation does not cause software bugs that are difficult to find. If higher isolation levels are used, the possibility of deadlock is increased, this also requires careful analysis and programming techniques to avoid.

IsolationLevel



Level	Isolation Level	Anomalies		
		Dirty reads	Non-repeatable reads	Phantoms
1	Read Uncommitted	NOT Preventable	NOT Preventable	NOT Preventable
2	Read Committed	Preventable	NOT Preventable	NOT Preventable
3	Repeatable Read	Preventable	Preventable	NOT Preventable
4	Serializable	Preventable	Preventable	Preventable

Level 1, TRANSACTION_READ_UNCOMMITTED is the weakest isolation level. When database is set with this level, all anomalies are possible. If your application does only read-only operations, then this should provide the highest performance.

Level 2 prevents dirty reads; while Level 3 prevents both dirty read aid non-repeatable reads.

Level 4 is the strongest isolation and prevents all three anomalies - dirty read, non-repeatable reads, and even phantom reads.

INDEX

4 Transaction Components:

4.1 Applications Components:

Application components are clients o support an application run-time environment which EXcludes transaction state management, for the transactional resources. These are the programs with which the application developer implements business transactions

4.2 Resource Manager:

A resource manager is a component that manages persistent and stable data storage system, and participates in the two phase commit with the transaction manager.

4.3 Transaction Manager:

The transaction manager is the core component of a transaction processing environment.

Its primary responsibilities are below:

- Management functions required to support **transaction demarcation**.
- **Resource Management**: To create transactions when requested by application components and to conduct the two-phase commit with the resource managers.
- Makes **synchronization** calls to the application components before beginning and after end of two-phase commit

INDEX

5 Transaction Processing:

5.1 Transaction Demarcation:

- A transaction has to be demarcated, meaning it has to have definite beginning and ending points initiated by someone.
- At the beginning point, a new transaction has to be created and ending of a transaction can be done either Committing or aborting the transaction (Rollback).

5.2 Resource Enlistment:

Resource enlistment is the process by which resource managers inform the transaction manager of their participation in a transaction. This process enables the transaction manager to keep track of all the resources participating in a transaction. The transaction manager uses this information to coordinate transactional work performed by the resource managers and to drivetwo-phase commitand recovery protocol.

5.3 Two-Phase Commit:

The commit process takes two phases to complete. In the first phase, checks are made to see that the transaction ran without error. If there were no errors, the results are committed in the second phase. If there were errors in the first-phase check, the transaction is rolled back. This protocol guarantees that the work is either successfully completed or not performed at all. This common transaction strategy is appropriately called the two-phase commit protocol.

Successful 2 Phase Commit:

Successful Two-phase Commit

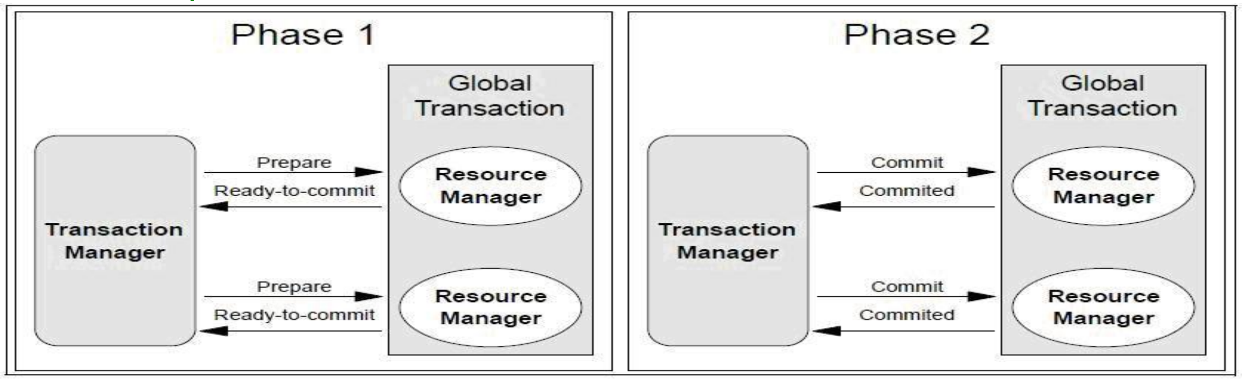


Fig 1 Successful two-phase commit
Above **Fig1** demonstrates when the transaction manager handles two resource managers in a two-phase commit transaction, where both resource managers are ready-to-commit and can commit at the end of the transaction.

Failure 2 Phase Commit:

Failure Two-phase Commit

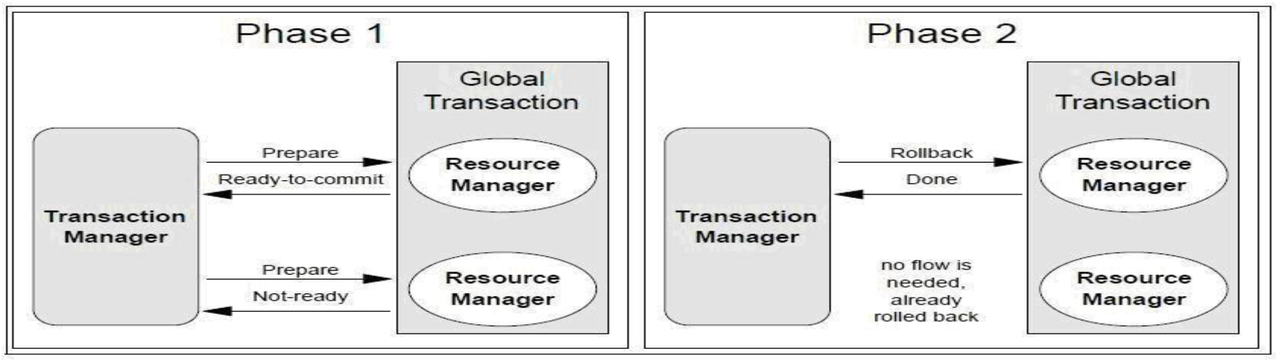


Fig 2 Failure two-phase commit
Above **Fig 2** demonstrates the case when the transaction manager handles two resource managers in a two-phase commit transaction, where one of the resource managers failed to prepare to commit the transaction.

INDEX

6 EJB Transactions:

The EJB framework does not specify any specific transaction service (such as the JTS) or protocol for transaction management. The specification requires that the javax.transaction.UserTransaction interface of the JTS be exposed to enterprise beans. This interface is required for programmatic transaction demarcation as discussed in the next section. The container performs automatic demarcation depending on the transaction attributes specified at the time of deploying an enterprise bean in a container.

The following determine how transactions are created.

1. **`NotSupported:`** The container invokes the bean without a global transaction context.
2. **`Required:`** The container invokes the bean within a global transaction context. If the invoking thread already has a transaction context associated, the container invokes the bean in the same context. Otherwise, the container creates a new transaction and invokes the bean within the transaction context.
3. **`Supports:`** The bean is transaction-ready. If the client invokes the bean within a transaction, the bean is also invoked within the same transaction. Otherwise, the bean is invoked without a transaction context.
4. **`RequiresNew:`** The container invokes the bean within a new transaction irrespective of whether the client is associated with a transaction or not.
5. **`Mandatory:`** The container must invoke the bean within a transaction. The caller should always start a transaction before invoking any method on the bean.
6. Never - If the client is running within a transaction and invokes the enterprise bean's method, the container throws a RemoteException. If the client is not associated with a transaction, the container does not start a new transaction before running the method.

7 Java Transaction Model: JTA & JTS:

The JTA API defines local Java interfaces required for the transaction manager to support distributed transaction management. This consists of three interfaces between a Transaction Manager and other Distributed Transaction Processing (DTP) participants:

1. UserTransaction Interface: A high-level application transaction interface, implemented in the javax.transaction.UserTransaction

interface.

It defines methods for explicit transactiondemarcation management by an application program

2. TransactionManager Interface: A high-level application server interface, implemented in thejavax.transaction.TransactionManager and javax.transaction.Transaction interfaces.It defines methods to manage the transaction and control the transaction boundaries.

3. XAResource: The mappings of the XA interface, implemented in the javax.transaction.xa.XAResource interface. This interface is implemented in resource adapters such as a JDBC driver (for the relational

database resource manager) or JMS provider (for the message queue server resource manager).

The JTA API is included in the Java 2 Enterprise Edition (J2EE) Standard Services specification.

INDEX

7.1 UserTransaction Interface

The javax.transaction.UserTransaction interface provides the application the ability to control transaction boundaries programmatically. This interface may be used by Java client programs or EJB beans.

The UserTransaction.begin method starts a global transaction and associates the transaction with the calling thread. The transaction-to-thread association is managedtransparently by the Transaction Manager.

Exception case:
The UserTransaction.begin method throws the NotSupportedException when the calling thread is already associated with a transaction and the transaction manager implementation does not support nested transactions.

7.1.1 UserTransaction Support in EJB Server

EJB servers are required to support the UserTransaction interface for use by EJB beans with the TX_BEAN_MANAGED transaction attribute. The UserTransaction interface is exposed to EJB components through the EJBContext interface using the getUserTransaction method. Thus, an EJB application does not interface with the Transaction Manager directly for transaction demarcation; instead, the EJB bean relies on the EJB Server to provide support for all of its transaction work as defined in the Enterprise JavaBeans Specification

The code sample below illustrates the usage of UserTransaction by a TX_BEAN_MANAGED EJB session bean:

Example.java

```
// In the session bean¿s setSessionContext method,
// store the bean context in an instance variable
SessionContext ctx = sessionContext;
// somewhere else in the bean¿s business logic
UserTransaction utx = ctx.getUserTransaction();
// start a transaction
utx.begin();
   .. do work
// commit the work
utx.commit();
```

7.1.2 UserTransaction Support for Transactional Clients

The UserTransaction interface may be used by Java client programs either through support from the application server or support from the transaction manager on the client host.

The application server vendor is expected to provide tools for an administrator to configure the UserTransaction object binding in the JNDI namespace. The implementation of the UserTransaction object must be both

- javax.naming.Referenceable
- java.io.Serializable

so that the object can be stored in all JNDI naming contexts.

An example of such an implementation is through the use of a system property. The following sample code is provided for illustrative purposes:

ExampleTx.java

```
// get the system property value configured by administrator
String utxPropVal = System.getProperty(¿jta.UserTransaction¿);
// use JNDI to locate the UserTransaction object
Context ctx = new InitialContext();
UserTransaction utx = (UserTransaction)ctx.lookup(utxPropVal);
// start transaction work..
utx.begin();
.. do work
utx.commit();
```

INDEX

7.2 TransactionManager Interface

The *javax.transaction.TransactionManager* interface allows the application server to control transaction boundaries on behalf of the application being managed.

For example, the EJB container manages the transaction states for transactional EJB components; the container uses the TransactionManager interface mainly to Java Transaction API demarcate transaction boundaries where operations affect the calling thread¿s transaction context. The Transaction Manager maintains the transaction context
association with threads as part of its internal data structure. A thread¿s transaction context is either null or it refers to a specific global transaction. Multiple threads may concurrently be associated with the same global transaction.

Each transaction context is encapsulated by a Transaction object, which can be used to perform operations which are specific to the target transaction, regardless of the calling thread¿s transaction context. The following sections provide more details.

7.2.1 Starting a Transaction:

TransactionManager.begin:

The *TransactionManager.begin* method starts a global transaction and associates the transaction context with the calling thread.

Exception case:
If the Transaction Manager implementation does not support nested transactions, the
TransactionManager.begin method throws the *NotSupportedException* when the calling thread is already
associated with a transaction.

TransactionManager.getTransaction:

The *TransactionManager.getTransaction()* method returns the Transaction object that represents the transaction context currently associated with the calling thread. This Transaction object can be used to perform various operations on the target transaction. Examples of Transaction object operations are resource enlistment and synchronization registration.

7.2.2 Transaction Completion

Commit:

The *TransactionManager.commit* method completes the transaction currently associated with the calling thread. After the commit method returns, the calling thread is not associated with a transaction.

If the commit method is called when the thread is not associated with any transaction context, the TxManager throws an exception. In some implementations, the commit operation is restricted to the transaction originator only.

Exception case:
If the calling thread is not allowed to commit the transaction, the TxManager throws an exception.

RollBack:

The *TransactionManager.rollback* method rolls back the transaction associated with the current thread. After the rollback method completes, the thread is associated with no transaction.

7.2.3 Suspending and Resuming a Transaction

Suspend:

A call to the TransactionManager.suspend method temporarily suspends the transaction that is currently associated with the calling thread.

If the thread is not associated with any transaction, a null object reference is returned; otherwise, a valid Transaction object is returned. The Transaction object can later be passed to the resume method to reinstate the transaction context association with the calling thread.

+application server: +
The application server is responsible for ensuring that the resources in use by the application are properly delisted from the suspended transaction. A resource delist operation triggers the Transaction Manager to inform the resource manager to disassociate the transaction from the specified resource object (XAResource.end(TMSUSPEND)).

Resume:

The *TransactionManager.resume* method re-associates the specified transaction context with the calling thread. If the transaction specified is a valid transaction, the transaction context is associated with the calling thread; otherwise, the thread is associated with no transaction.

TmSuspend.java
```
Transaction tobj = TransactionManager.suspend();
..
TransactionManager.resume(tobj);
```

Exception case:
If TransactionManager.resume is invoked when the calling thread is already associated with another transaction, the Transaction Manager throws the *IllegalStateException* exception.

Some transaction manager implementations allow a suspended transaction to be resumed by a different thread. This feature is not required by JTA.

+application server: +
The application¿s transaction context is resumed, the application server ensures that the resource in use by the application is again enlisted with the transaction. Enlisting a resource as a result of resuming a transaction triggers the Transaction Manager to inform the resource manager to re-associate the resource object with the resumed transaction (XAResource.start(TMRESUME)).

INDEX

7.3 XAResource Interface

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).

The XAResource interface defines the contract between a Resource Manager and a Transaction Manager in a distributed transaction processing (DTP) environment. A resource adapter for a resource manager implements the XAResource interface to support association of a global transaction to a transaction resource, such as a connection to a relational database.

A global transaction is a unit of work that is performed by one or more resource managers (RM) in a DTP system. Such a system relies on an external transaction manager, such as Java Transaction Service (JTS), to coordinate transactions global transaction is a unit of work that is performed by one or more resource managers (RM) in a DTP system. Such a system relies on an external transaction manager, such as Java Transaction Service (JTS), to coordinate transactions.

XAResource:

XAResource ›

7.3.1 Opening a Resource Manager

A resource manager is closed by the resource adapter as a result of destroying the transactional resource. A transaction resource at the resource adapter level is comprised of two separate objects:

¿ An *XAResource* object that allows the transaction manager to start and end the transaction association with the resource in use and to coordinate transactioncompletion process.

¿ A *connection* object that allows the application to perform operations on the underlying resource (for example, JDBC operations on an RDBMS).

The resource manager, once opened, is kept open until the resource is released (closed) explicitly. When the application invokes the connection¿s close method, the resource adapter invalidates the connection object reference that was held by the application

and notifies the application server about the close. The transaction manager should invoke the XAResource.end method to disassociate the transaction from that connection. The close notification allows the application server to perform any necessary cleanup work and to mark the physical XA connection as free for
reuse, if connection pooling is in place.

The X/Open XA interface specifies that the transaction manager must initialize a resource manager (xa_open) prior to any other xa_ calls.

The transaction manager does not need to know how to initialize a resource manager. The TM is only responsible for informing the resource manager about when to start and end work associated with a global transaction and when to complete the transaction.

The resource adapter is responsible for opening (initializing) the resource manager when the connection to the resource manager is established.

7.3.2 Resource Sharing

When the same transactional resource is used to multiple transactions, it is the responsibility of the application server to ensure that only one transaction is enlisted with the resource at any given time. To initiate the transaction commit process, the transaction manager is allowed to use any of the resource objects connected to the same resource manager instance. The resource object used for the two-phase commit protocol need not have been involved with the transaction being completed.

The resource adapter must be able to handle multiple threads invoking the XAResource methods concurrently for transaction commit processing. For example, suppose we have a transactional resource r1. Global transaction xid1 was started and ended with r1. Then a different global transaction xid2 is associated with r1. Meanwhile, the transaction manager may start the two phase commit process for xid1 using r1 or any other transactional resource connected to the same resource manager. The resource adapter needs to allow the commit process to be executed while the resource is currently associated with a different global transaction.

The sample code below illustrates the above scenario:

ResrcSharing.java

```
// Suppose have some transactional connection-based esource r1
// that is connected to an enterprise information service system.
XAResource xares = r1.getXAResource();
xares.start(xid1); // associate xid1 to the connection
..
xares.end(xid1); // dissociate xid1 to the connection
..
xares.start(xid2); // associate xid2 to the connection
..
// While the connection is associated with xid2,
// the TM starts the commit process for xid1
status = xares.prepare(xid1);
..
xares.commit(xid1, false);
```

7.3.3 Local and Global Transactions

The resource adapter is encouraged to support the usage of both local and global Txs within the same transactional connection. Local Txs are transactions that are started and coordinated by the resource manager internally. The
XAResource interface is not used for local transactions.

When using the same connection to perform both local and global transactions, the following rules apply:

- The local transaction must be committed (or rolled back) before starting a global transaction in the connection.
- The global transaction must be disassociated from the connection before any local transaction is started.

<u>Exception case:</u>
If a resource adapter does not support mixing local and global txs within the same connection, the resource
adapter should throw the resource specific exception.

For example, java.sql.SQLException is thrown to the application if the resource manager for the underlying
RDBMS does not support mixing local and global txs within the same JDBC connection.

7.3.4 Failures Recovery

During recovery, the Transaction Manager must be able to communicate to all resource managers that are in use by the applications in the system. For each resource manager, the Transaction Manager uses the *XAResource.recover* method to retrieve the list of txs that are currently in a prepared or heuristically completed state.

Typically, the system administrator configures all transactional resource factories that are used by the applications deployed on the system. An example of such a resource factory is the JDBC XADataSource object, which is a factory for the JDBC XAConnection objects. The implementation of these transactional resource factory
objects are both *javax.naming.Referenceable* and *java.io.Serializable* so that they can be stored in all JNDI naming contexts. Because XAResource objects are not persistent across system failures, the Transaction Manager needs to have some way to acquire the XAResource objects that represent the resource managers which might have participated in the txs prior to the system failure. For example, a Transaction Manager might, through the use of the JNDI lookup mechanism and cooperation from the application server, acquire an XAResource object representing each of the Resource Manager configured in the system. The Transaction Manager then invokes the *XAResource.recover* method to ask each resource manager to return any transactions that are currently in a prepared o heuristically completed state. It is the responsibility of the Transaction Manager to ignore transactions that do not belong to it.

Example

INDEX

a Reference:

Assessment:

- Write a Program using JTA to handle with Multiple transactions

a Reference:

1. http://community.jboss.org/wiki/SessionsAndTransactions
2. http://www.subbu.org/articles/nuts-and-bolts-of-transaction-processing
3. http://www.javapassion.com/
4. http://docs.redhat.com/docs/en-US/JBoss_Enterprise_Web_Platform/5/
5. http://static.springsource.org/spring/docs/2.5.x/reference/transaction.html

INDEX