An Oracle White Paper
June 2010

XA and Oracle controlled Distributed Transactions

Introduction

In today's computing environment the use of distributed or "global" transactions is becoming more widespread with the adoption of Java EE component based architectures. Although distributed transactions have been in use for many years, the XA specification being published in 1991, there is a distinct lack of awareness of the design implications of using distributed transactions within an architecture.

This Oracle whitepaper discusses some of the potential areas that need to be considered when using either XA or Oracle-controlled distributed transactions, namely:

- **Data Recoverability**: as there are multiple points of the same truth, then transactional consistency can only be maintained if all data sources can be recovered to the same transactionally consistent point. With Oracle-controlled transactions this is possible using the System Change Number (SCN), but this can increase the level of data loss to encompass all databases involved in the recovery process. With XA distributed transactions there is no easy way to synchronize heterogeneous data sources to the same transactionally consistent point.

- **In-doubt Transactions**: On failure, "in-doubt" transactions can cause locks to be placed on any data that has been changed and "prepared" by a failed distributed transaction. Until the "prepared" transaction can be recovered to either confirm (commit) or deny (rollback) these changes, the affected data may not be accessible to other transactions. A single failure can then escalate to encompass a wider set of functionality within the application(s) using this data.

- **Performance**: Because of the additional communication involved within a distributed two-phase commit (e.g. the 'Prepare to Commit' phases), as well as the need to write transactional log records, use of distributed transactions typically incurs a performance overhead compared to single-phase transactions.

Distributed transactions should only be used where multiple data sources must be updated in the same business transaction or when used as an integral part of a product (e.g. Oracle Application Server BPEL Process Manager).

The scope of the distributed transaction should be small, both to limit the impact of failures, and to simplify any custom recovery procedures that may be needed to restore transactional consistency across all the related databases / data sources.

Alternative approaches should be considered before committing to an architecture that encompasses distributed transactions. For example, if one data source is driving updates to the other, then updates to the secondary data source could be committed before updates to the primary (driving) data source. If the transaction fails between these commits, the data is committed on the secondary data source but will rollback on the primary (driving) data source. On restart (or as part of the normal transaction processing), the driving data source can then check whether the update has been applied to the secondary data source (via a unique key for example) before performing any processing. If a duplicate is detected, then the transaction can be discarded and the primary data source updated to reflect the success of the update on the secondary data source.

Distributed Transactions Overview

A distributed transaction is a set of two or more related transaction branches that have to be managed in a coordinated fashion. The transactions span multiple data sources. For example, a distributed transaction may involve managing a transaction across both a database (e.g. Oracle) and a message queue (e.g. MQ Series) or two independent databases (e.g. Oracle and Oracle). When using distributed transactions there are multiple points of the truth, this can cause quite a few challenges especially in recovery situations, where all points of the truth need to be recovered to the same transactionally consistent point. However, the main challenge is to have a mechanism that maintains transactional consistency during normal running, this is achieved using a two-phase commit mechanism.

A two-phase commit mechanism is broken into two key phases, a “prepare” phase and a “commit” phase. In the prepare phase, the coordinator of the transaction asks the other participating databases / data sources to promise to commit or rollback the transaction. They persistently record their transactions as “prepared” and send a response back to the transaction coordinator. During the commit phase, if all the transaction participants have agreed to commit the transaction, then the transaction coordinator asks all participants to commit the transaction. Otherwise if one or more participants cannot agree to commit the transaction, then the transaction coordinator asks all participants to rollback the transaction. Once all participants have acknowledged they have performed the commit or rollback request, the transaction is complete and the transaction coordinator can then remove or “forget” the existence of the distributed transaction. This is sometimes referred to as the “forget phase” and is implementation specific as to how and when the transaction coordinator removes information relating to competed distributed transactions.

Most implementations of two-phase commit have a "Read Only" optimization, whereby data sources that have not had any changes applied withdraw from the scope of the distributed transaction during the prepare phase.

Likewise implementations can also have a "Single Phase" optimization, whereby if only one data source has been opened or updated then the prepare phase will be omitted with the transaction being committed in a single-phase.

All participants within a distributed transaction should perform the same action: they should either all commit or all perform a rollback of the transaction. The transaction coordinator automatically controls and monitors the commit or rollback of a distributed transaction and maintains transactional integrity using the two-phase commit mechanism.

However if a failure occurs between the prepare phase and the commit phase because the transaction coordinator becomes unavailable, then transactions and their associated data can be left "in-doubt". Data within the data-source marked "in-doubt" **cannot** be changed until the transaction outcome is resolved either by the transaction coordinator or by external intervention (e.g. by a DBA or systems administrator). In the majority of cases, the in-doubt transaction is automatically recovered once the transaction coordinator can communicate to the data source. Manual resolution should only be required when the in-doubt transaction has locks on critical data resources or the cause of the machine, network, or software failure cannot be repaired quickly.

Generally the two-phase commit mechanism is completely transparent, requiring no programming on the part of the user or application developer. However it does require a transaction coordinator, this can be either a database such as Oracle, a transaction monitor such as Tuxedo or an application server such as Oracle WebLogic Server.

Oracle-Controlled Distributed Transactions

The Oracle database can be used to co-ordinate a distributed transaction across two or more Oracle databases participating in a single transaction. For the purposes of this section only transactions that utilize a Database Link are discussed, examples of Oracle based products that use Database Links as part of a distributed transaction are Advanced Replication and Oracle AQ propagation. There are other distributed transaction mechanisms used within the Oracle database such as XA and those used by the procedural gateways, these differ from the standard distributed transaction processing described here.

The Oracle distributed transaction mechanism has both "Read Only" and "Single Phase" optimizations as described previously. If a database is asked to prepare and no updates have occurred on that database, then it will return a read only status code and not take any further part in the transaction. For a distributed transaction to occur within Oracle, changes must be made against two or more databases within the same transaction. If only one database has been updated, then the prepare phase is bypassed and this is considered a "remote" transaction rather than a distributed transaction.

For both distributed and remote transactions the initiating database is the global coordinator of the transaction. Of those databases that have been updated, the database with the greatest

COMMIT_POINT_STRENGTH is nominated as the "commit point site" for the transaction. For remote transactions, this is and can only be the database that was updated. The "commit point site" is the single point where the status of the transaction is recorded and it will commit before any of the other databases involved in the transaction. It is the only database that never enters the "prepare" state and can therefore never have any 'in-doubt' transactions on failure. In effect, the outcome of the distributed or remote transaction at the commit point site determines whether the transaction is committed or rolled back. Other nodes if involved will follow the lead of the commit point site. The global coordinator ensures that all nodes within a distributed transaction complete the transaction in the same manner as the commit point site.

Note that for distributed transactions, the database selected as commit point site should be the database that stores the most critical data, as this database never enters the prepared state and consequently can never become "in-doubt", even if a failure occurs. In failure situations, failed nodes remain in a prepared state, holding locks on affected data until the "in-doubt" transactions are resolved.

The Oracle two-phase commit mechanism has the following distinct phases, which the database performs automatically whenever a user commits a distributed transaction:

- **Prepare phase**: The global coordinator, asks participating nodes other than the commit point site to promise to commit or rollback the transaction, even if there is a failure. If any node cannot prepare, the transaction is rolled back.
- **Commit phase**: If all participants respond to the coordinator that they are prepared, then the coordinator asks the commit point site to commit. After it commits, the coordinator asks all other nodes to commit the transaction.
- **Forget phase**: The global coordinator forgets about the transaction.

Within Oracle each committed transaction has an associated System Change Number (SCN) to uniquely identify the changes made by the SQL statements within each transaction. Oracle utilizes the SCN to maintain read consistency for queries. As each database has its own SCN that is maintained independently, there is the opportunity for one database to be out of date with respect to the other databases within the distributed transaction. To minimize the impact of this gap, the SCNs in a distributed transaction are synchronized at the end of each remote SQL statement and at the start and end of each transaction. Because of this SCN gap, a query can be executed that uses a slightly old snapshot, such that the most recent changes to the remote database are not seen. In accordance with read consistency, a query can therefore retrieve consistent, but out-of-date data. Note that all data retrieved by the query will be from the old SCN, so that if a locally executed update transaction updates two tables at a remote node, then data selected from both tables in the next remote access contain data prior to the update. If SCN synchronization is required for specific queries then a common workaround is to precede each remote query with a dummy remote query to the same site, for example, SELECT * FROM DUAL@REMOTE.

One of the benefits of the coordination and synchronization of SCNs is that incomplete recovery of all databases to a single SCN (transaction) can be achieved; this enables transactional consistency to be maintained even when only one of the databases can be partially recovered (e.g. due to a disaster). However, incomplete recovery does mean data loss across all databases unless there is a custom mechanism to replay all missing transactions.

To simulate different failure scenarios within an Oracle-controlled distributed transaction see My Oracle Support Note 126069.1 Manually Resolving In-Doubt Transactions: Different Scenarios.

XA Based Distributed Transactions

The X/Open Distributed Transaction Processing Model

The XA standard is an X/Open specification for distributed transaction processing (DTP) across heterogeneous data sources (e.g. Oracle Database and DB2) that was published in 1991.

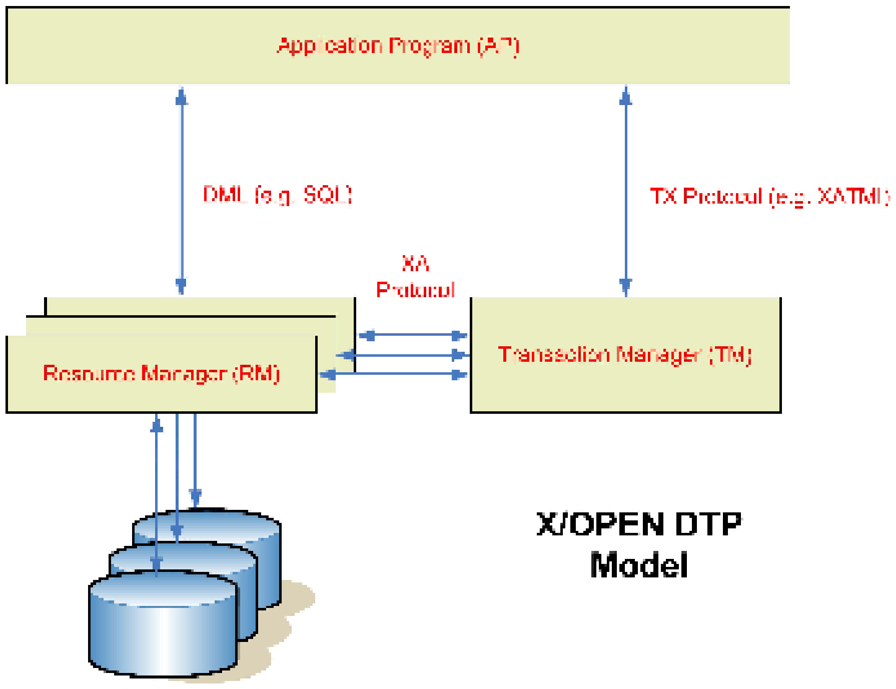
It describes the interface between the transaction coordinator and the data sources that participate in the distributed transaction. Within XA the transaction coordinator is termed the Transaction Manager and the participating data sources are termed the Resource Managers. Transaction Managers and Resource managers that follow this specification are said to be XA compliant.

Some products such as the Oracle database and Oracle WebLogic Server can act as either Transaction Managers or Resource Managers or both within the same XA transaction.

Examples of an XA Transaction Manager are: Tuxedo, Oracle WebLogic Server, the Oracle database and IBM WebSphere Application Server. Examples of an XA Resource Manager are: Oracle Database, IBM DB2, MS-SQL, IBM MQ-Series and JMS.

When referring to XA, most organizations refer to version 1 of the XA specification. However a snapshot (draft) of version 2 was published in 1994 and is known as the XA+ specification, this was never ratified and formally published as a technical standard and is therefore not used by any of the key XA vendors. The XA+ specification was an enhanced specification that would enable distributed transactions to be coordinated across multiple heterogeneous Transaction Managers (e.g. Tuxedo and IBM Encina) as well as dictating how Transaction Managers should communicate with multiple RM instances (e.g. RAC). A version of the XA Specification can also be found in My Oracle Support Note 180207.1 The XA specification for Distributed Transaction Processing.

The X/Open DTP model is depicted in the following diagram. An Application Program (AP) updates multiple XA compliant Resource Managers (RM). All of the updates are coordinated by an XA compliant Transaction Manager (TM) to ensure transactional consistency across all of the RMs. Each RM will have at least one branch of the distributed transaction coordinated by the TM. All branches of the distributed transaction must be either committed or rolled-back as a single unit of work.



Within the X/Open DTP model the AP will communicate to the RM using the RM's native interface (e.g. SQL) for all DML. The AP will also communicate to the TM using the TX protocol to demarcate transaction boundaries. The TX protocol has a number of "standard" implementations, namely XATMI (Tuxedo based), TxRPC (DCE based) and XCPI-C (IBM Encina based). Not all of these implementations are supported by all TMs e.g. Tuxedo only supports the XATMI and TxRPC standards, whereas IBM Encina only supports the TxRPC and XCPI-C standards. The TM will communicate to all the RMs using the XA protocol to co-ordinate a two-phase commit.

JTA and JTS

Although the use of the X/OPEN DTP model is mainly restricted to Transaction Managers such as Tuxedo and Encina, the X/Open DTP concepts have been widely deployed. For example, the XA specification has now been implemented as a standard set of Java APIs within Java EE known as the Java Transaction API (JTA) specification. JTA enables a Java based Transaction Manager to perform distributed transactions. The JTA is basically the X/Open DTP model, but replaces the TX protocol with a set of Java APIs that application programs use to demarcate transaction boundaries. An example of a JTA Transaction Manager is Oracle WebLogic Server.

Also within Java EE specification there is the Java Transaction Service (JTS). JTS is a Java wrapper that supports the JTA specification at the high level whilst implementing the Java mapping of the OMG's Object Transaction Service (OTS) at a lower level. OTS is a component of the OMG's Common Object Request Broker Architecture (CORBA) which was never widely adopted. The OTS specification was built with the help of X/Open and is based on the concepts proposed within XA+ to provide a means of enabling distributed transactions to be coordinated across heterogeneous or multiple Transaction Managers. The OTS model also replaced the functional XA and TX interfaces defined within the X/Open DTP model with CORBA specific interfaces. The OTS model was fully interoperable with the X/Open DTP model. Again the XA interface underpinned all communications between the TM and the RM. JTS does NOT implement the OTS model, but provides a specification for interfaces into an OTS / COBRA compliant transaction manager.

XA Transaction Concepts

Within XA, the TM co-ordinates all branches of a distributed transaction as well as maintaining state information on all distributed transactions within its control. If this state information is lost or becomes unavailable, then the underlying RMs may require manual intervention to resolve any in-doubt transactions.

As the TM is responsible for coordinating and monitoring the progress of a distributed transaction, the application program MUST NOT contain any RM specific statements that will independently rollback or commit a distributed transaction. Therefore the application program MUST NOT issue any DDL (e.g. TRUNCATE, CREATE TABLE etc) as DDL is itself an implicit self-contained transaction. Always refer to the RM's manuals to find out as to which statements are specifically allowed.

The XA two-phase commit mechanism has the following distinct phases, which the TM and RMs perform automatically whenever an application program commits a distributed transaction:

- **Prepare phase**: The TM asks participating RMs to promise to commit (or rollback) the transaction as requested, even if there is a failure. If any RM cannot prepare, the transaction is rolled back.
- **Commit phase**: If all RMs respond to the TM that they are prepared, then the TM asks all RMs to commit the transaction.

XA supports both the "Read Only" and "Single Phase" optimizations as discussed in the two-phase commit section above.

Within an XA distributed transaction there are a least two or more transaction branches that will perform work against one or more RMs. Within a single RM, these branches can be related whereby the relationship between any pair of participating branches is either tightly-coupled or loosely-coupled. If the transaction branches are tightly coupled, then they share locks. Consequently for tightly coupled transactions, pre-commit updates made in one transaction branch will be visible in other branches that belong to the same distributed transaction. In loosely coupled transaction branches, the branches do not share locks and do not see updates in other branches.

Tightly coupled transactions require the RM to guarantee that resource deadlock will not occur between individual branches of an XA transaction. The way that this is implemented is at the discretion of the RM, but it can have the effect of limiting concurrent data access across all tightly coupled branches. (e.g. only one branch can be actively using DML at a time).

The XA standard allows for heuristic completion of one or more transaction branches within a distributed transaction. If a communication failure occurs, some RMs may employ heuristic decision making to decide whether a prepared transaction is committed or rolled back independently of the TM. For example this could be to unlock shared resources, however this may leave the data in an inconsistent state with regards to the rest of the distributed transaction, e.g. if an RM commits it's branch but all other branches are asked to rollback, then the result will be inconsistent across all RMs. Alternatively there may have been manual intervention (e.g. by a DBA or system administrator) to release locks, which again results in Heuristic completion of a distributed transaction. Note that if an RM that reports heuristic completion to the TM, the RM must not discard its knowledge of the transaction branch, this often means that although the outcome of the distributed transaction has been resolved manually, references to the heuristic transaction may still remain within the RM until manually removed.

The XA standard also allows for both synchronous and asynchronous modes of working. The asynchronous mode is optional and NOT supported by the Oracle database.

To simulate the different failure scenarios within a XA controlled distributed transaction see My Oracle Support notes:

- 50743.1 Session Switching & Global Transaction Mgt, which provides a 'C' source program;
- 227352.1 Demonstration of Integrating OJMS operations within an XA Transaction, which provides a Java source program.

Unlike Oracle-controlled distributed transactions, the XA specification does not allow a 'point in time' marker to be agreed between participating databases (e.g. an equivalent to the SCN synchronization discussed in Oracle-controlled distributed transactions above). This means that if an incomplete recovery of one or more RMs is required, there is NO mechanism of bringing all RMs to the same transactionally consistent point in time. For incomplete recovery there are two options:

- Design an application specific recovery mechanism to ensure transactional consistency across all RMs is maintained in the event of an incomplete recovery being performed;
- A point in time recovery is performed to a point where there was little or no activity on the RMs (e.g. from a cold backup or recovery to a point where the application was quiesed). My Oracle Support Note 52776.1 Determining a Point In Time for Incomplete Recovery under XA, describes a mechanism for identifying when to perform incomplete recovery within Oracle.

XA and Point in Time data consistency across Resource Managers

It should be noted that within in an OLTP system there could be a lot of transactions occurring at any particular point in time, the state of individual transactions will vary across RMs and there is no guarantee that all branches of a transaction complete at the same point in time. This affects both backup / recovery as described previously as well as application designs that expect transactional consistency across all RMs at a specific point in time.

The XA specification doesn't state what order a TM requests RMs to commit or in what time frame the RMs must commit. It only states that all RMs who have agreed to commit must commit the data. Therefore there is the possibility of a race condition occurring, especially if processing is dependent on the order of writes to one or more RMs.

This is a significant point, since an application program that is independent of the one performing the distributed transaction may be able to read changes made by a distributed transaction on one RM before they are available on other RMs within the same distributed transaction. Note: the sequence in which the TM requests RMs to commit can be in any order, so the first RM requested to commit may have its changes visible before the last RM to be requested to commit within the same distributed transaction. Conversely, although the RM has acknowledged the commit to the TM, these changes may need to be processed before they are made available to other users of the RM. When multiple or different RMs are used within the same distributed transaction, the level of processing to perform the commit and make these changes visible to others will vary. Therefore there is the possibility that the last RM to be asked to commit within the distributed transaction could makes its changes visible before the first RM requested to commit by the TM. Either way transactional consistency is not guaranteed until all RMs have committed and made their changes visible, whilst the commit phase is in progress there is the opportunity for by an independent reader to obtain an inconsistent view of the data across multiple RMs.

This problem of data consistency at a particular point in time tends to affect queue based architectures the most. For example, a transaction updates one or more databases and a file based message queue. If an independent process monitors the queue and reads the message from the queue during the commit phase, there is the possibility that the corresponding updates will not available to readers on the databases being updated. In these use cases the independent process reading the queue may fail if it cannot find the changes it expects or end up making changes to data based on values that were valid prior to the distributed transaction starting.

Using XA and Oracle-Controlled Distributed Transactions

XA Timeouts and the Oracle Database

There are a number of timeouts used within XA implementations to mainly avoid locking issues between the prepare and commit phases. E.g. if a TM disappears after the prepare stage, the RM needs to be able to release locks for other transactions that may not be managed by the failing TM instance. The rule of thumb for these timeouts is that those used by the RM must be greater than those used by the TM, such that the TM always remains in control of the distributed transaction.

For instance, if the RM's timeouts are less than the TM's timeouts, then the RM may discard a transaction without reference to the controlling TM. A common result in this case is a response of "ORA-24756: Transaction does not exist" from the Oracle database.

Within the Oracle XA implementation there are a number of timeouts that are utilized to determine failures within a distributed transaction. Two of these timeouts are passed by the TM to Oracle when initially opening a connection using the xa_open api, these are:

- **SesTm** (session timeout) specifies the maximum length of time a transaction can be inactive before it is automatically aborted by the system. For example, if the TM uses a remote procedure call between the client and the server, then SesTM applies to the time between the completion of one RPC and the initiation of the next RPC, or the commit / rollback;
- **SesWt** (session wait) specifies the maximum timeout limit when waiting for a transaction branch that is being used by another session. The default value is 60 seconds.

Both the SesTM and SesWt timeouts are set in TM specific configuration files via the database connection string. SesTM is mandatory, whilst SesWt is often left to its default value. An example connection string is as follows:

```
ORACLE_XA+SqlNet=SID3+ACC=P/user/pwd  +SesTM=10+LogDir=/usr/local/xalog
```

The SesTM and SesWt timeouts do not apply to XA transactions managed through a JDBC connection, in these cases the setTransactionTimeout method is used instead.

Similarly, for PL/SQL (from Oracle Database 11gR1), the XA_SETTIMEOUT function within the DBMS_XA package can be used to set the transaction timeout, the default value is 60 seconds.

Additionally there are the following timeouts that need to be considered, these are:

- **The TM's global transaction timeout** (for Java EE based Application Servers this is the JTA timeout);
- The Oracle database initialization parameter **DISTRIBUTED_LOCK_TIMEOUT** which limits how long distributed transactions will wait for a lock;
- The **CONNECT_TIME** & **IDLE_TIME** parameters within an Oracle database users PROFILE (these are both usually defaulted to UNLIMITED for the DEFAULT profile)

Generally, these timeouts should be set in the following order:

1. TM's global transaction timeout (or the JTA timeout) <
2. SesTm and SesWt (Transaction Timeout for JDBC or PL/SQL) <
3. DISTRIBUTED_LOCK_TIMEOUT <
4. Either of CONNECT_TIME or IDLE_TIME

Note that if a session times out within Oracle, then the Oracle database's Process Monitor (PMON) will delete the transaction branch if not prepared or cause the transaction to become in-doubt if prepared. As Process Monitor (PMON) activates every 60 seconds if not already active, there can be a delay of up to 60 seconds before the session is tidied up after the session timeout expiring. Therefore the time between the last action taking place e.g. "prepare" and the transaction subsequently being rolled back or appearing as an "in-doubt" transaction is the SesTm timeout plus up to 60 seconds.

If a transaction times out and becomes "in-doubt", it will appear in the DBA_2PC_PENDING and DBA_PENDING_TRANSACTIONS views. The in "in-doubt" transaction can then cause any subsequent transactions that wish to access this data to fail with "ORA-01591 lock held by in-doubt distributed transaction <tran id >". Within the majority of cases, the in-doubt transaction is automatically recovered once the TM can communicate to the RM. Manual resolution should only be required when the in-doubt transaction has locks on critical data, resources or the cause of the machine, network, or software failure cannot be repaired quickly. However, manual resolution of a transaction may result in Heuristic completion of the distributed transaction, which itself may need resolving later on.

Alternatively, if the DISTRIBUTED_LOCK_TIMEOUT is set to low, then transactions may timeout waiting for other sessions to complete. This can result in "Error - ORA-02049 timeout: distributed transaction waiting for lock".

Distributed Transactions and Database Locking

All tightly coupled distributed transactions acquire a DX (Distributed Transaction) enqueue while they are actively working on the transaction. This enqueue (memory lock) is used to serialize access to the database across multiple branches of a distributed transaction, such that only one branch of the transaction can have SQL active at any point in time.

Within a single instance database deployment all distributed transactions are tightly coupled by default. The XA transaction manager can override this at the start of a distributed transaction.

Similarly within a multiple instance RAC environment all distributed transactions are tightly coupled by default. However prior to Oracle Database 11gR1 RAC, DX enqueues were maintained independently within each instance. Since Oracle Database 11gR1 RAC this has changed and a cluster wide DX enqueue is acquired for each tightly coupled distributed transaction. This cluster wide DX enqueue is managed by the Global Enqueue Service (GES) within RAC, and control of this enqueue is transferred between database instances when distributed transactions span more than one database instance in the cluster. However, only one tightly coupled transaction branch can be active at any one time.

To complement the changes within DX enqueue processing, the Oracle Database read consistency mechanism of 11gR1 RAC enables all pre-commit updates made in one transaction branch of a tightly coupled transaction to be visible to other tightly coupled branches in different instances of the database. Prior to 11gR1, changes made on one database instance were not visible to branches located on another instance of the same database.

Loosely coupled distributed transactions are independent of each other and do not acquire a DX enqueue and do not need to serialize transactional activity across different transaction branches.

The use of multiple branches within a distributed transactions forces all transaction branches to go through a two-phase commit, that is, the system cannot use the XA "Single Phase" optimization when multiple branches of the same distributed transaction occur against the same database.

Generally, Oracle maintains read consistency based on the SCN (System Change Number) when the read statement was initiated, only changes that were committed before the read statement was executed will be included in the returned result set. However there is an exception when it comes to distributed transactions, where the reader can be blocked and block level locks can be acquired between the prepare and commit phases.

Prior to 10gR2, externally coordinated distributed transactions (XA) followed the general Oracle distributed transaction locking strategy and locked changed data at block level whilst blocking readers between prepare and commit stages. E.g. readers could not view the before or after images until the changes were committed. Since 10gR2, this behavior has been changed such that externally coordinated distributed transactions will lock at row level and do not block readers between the prepare and commit phases. This behavior is available at previous versions of the database via patch 3566945, which was incorporated into the 9.2.0.6 and 10.1.0.3 patchsets.

However for Oracle-controlled distributed transactions there is no change in functionality, where the reader is blocked and block level locks are acquired between the prepare and commit phases. This is because within an Oracle-controlled distributed transaction the changes can have a commit SCN lower than the current system SCN, it is therefore not known at read time if a prepared transaction should be visible or not to the reader. With Oracle-controlled distributed transactions, all databases involved in the transaction will coordinate such that they align their SCNs during the commit phase.

XA and Oracle distributed transactions that change rows that have been altered by another transaction will wait until the locking transaction completes. Ideally the locking transaction will successfully complete before the waiting distributed transaction times out either due to the TM transaction timeout or the database DISTRIBUTED_LOCK_TIMEOUT. If the locking transaction becomes "in-doubt", then the waiting transactions will fail with "ORA-01591 lock held by in-doubt distributed transaction <tran id>". If the locking transaction runs for longer than the database DISTRIBUTED_LOCK_TIMEOUT (e.g. a single phase transaction with no timeout) a deadlock type situation occurs and the XA / Oracle distributed transactions can fail with "ORA-02049: timeout: distributed transaction waiting for lock". If the ordering of timeouts is correct, then this error should never be received within a XA distributed transaction since the TM timeout should be exceeded before the DISTRIBUTED_LOCK_TIMEOUT is reached.

When a distributed transaction branch fails between prepare and commit, then the transaction becomes "in-doubt" until it can be resolved by the transaction coordinator. In this case, all rows changed by the failing transaction will be locked until the transaction is successfully resolved. All transactions attempting to access this locked data will immediately fail with "ORA-01591 lock held by in-doubt distributed transaction <tran id >".

An application that uses distributed transactions must be designed to avoid contention at both row and block level. The inherent latency added by performing the prepare and commit phases over performing a single phase commit can significantly degrade performance and throughput if transactions require access to the same data or indexes. e.g. heavily inserting into the same part of a non-unique index will cause contention on the index header blocks as and when the underlying leaf splits. Likewise the impact of "in-doubt" transactions can have a catastrophic cascading impact on the applications that requires access to the same rows or blocks of data.

Distributed Transactions and Read Only Optimization

Prior to Oracle RAC 11gR1, the distributed transaction read-only optimization was only done at database instance level within RAC. If a distributed transaction used more than one RAC database instance, then Read Only optimization would not occur.

Since 11gR1, the distributed transaction read-only optimization can now take place when a distributed transaction uses more than one RAC database instance.

Distributed Transactions and Redo

Both XA & Oracle distributed transactions write a redo record during the prepare phase and a redo record during the commit phase. At 11gR1, an additional sibling record is written at the prepare stage if there are more than two database instances involved in the same distributed transaction. The record written to the redo during prepare is an internal redo record that is not visible through Oracle Log Miner. Incomplete recovery of a database using redo logs (e.g. some redo has been lost either by point in time recovery from a backup or by using asynchronous Data Guard) may result in transactions being left "in-doubt".

Managing distributed transactions within the Oracle database

The Oracle database allows access to the following information on distributed transactions within the database:

- Information on unresolved distributed transactions (e.g. prepared and awaiting commit/rollback) is available via the DBA_PENDING_TRANSACTIONS system view.
- Information on distributed transactions awaiting recovery (e.g. "in-doubt" transactions) is available via the DBA_2PC_PENDING system view.
- Information on the currently active distributed transactions is available via the V$GLOBAL_TRANSACTION system view.

- Information on incoming and outgoing connections for pending transactions within an Oracle distributed transaction is available via the DBA_2PC_NEIGHBORS system view.

"In-doubt" transactions can be manually committed (commit force 'tran_id';) or manually rolled back (rollback force 'tran_id') if the TM is no longer able to recover the transaction. Where possible the TM should be allowed to recover the distributed transaction to avoid heuristic completion (e.g. different outcomes on each of the RMs involved in the transaction).

Purging old distributed transactions that have been manually resolved can be achieved using EXECUTE DBMS_TRANSACTION.PURGE_LOST_DB_ENTRY('<tran_id>'), details of how this procedure is used is described in My Oracle Support Note: 159377.1 How to Purge a Distributed Transaction from a Database.

XA and Oracle Real Application Clusters (RAC)

RAC has a number of restrictions / best practices, the main ones are:

- Prior to 11gR1, the entire transaction branch must be performed on the same database instance. On failover, special conditions apply depending on which version of RAC is deployed.
- Prior to 11gR1, an XA Transaction must not be load balanced across instances. The XA specification allows for a transaction branch to be suspended and resumed by the TM, if connection load balancing is utilized then any resumed connection could land on an alternate Oracle instance to the one that the transaction branch started on. From 11gR1, this restriction is removed by default; however there is a performance implication if a single distributed transaction spans multiple database instances.
- There is no failsafe capability for guaranteeing global uniqueness of the transaction id (XID) within RAC. RAC cannot check that the given XID is unique among all the instances within a RAC cluster. The TM needs to maintain the global uniqueness of XIDs, although that said, according to the XA specification the RM must always accept the XIDs provided by the TM.
- Prior to 11gR1, to avoid the prospect of tightly coupled transactions not seeing changes made by each other, all transaction branches of a tightly coupled distributed transaction must be located on the same instance. Likewise it was recommended although not mandatory, to locate all loosely coupled transactions on the same database instance, this was to avoid different outcomes from multiple transaction branches within the same distributed transaction (e.g. one is rolled back whilst the other is committed). From 11gR1, these restrictions are removed, all distributed transactions within a cluster default to being tightly coupled; however there is a performance implication if a single distributed transaction spans multiple database instances.

Special conditions apply when connections failover to an alternate instance within the RAC cluster after either instance loss or loss of network connectivity between the TM and RM (database) instance. If the TM tries to obtain the status of global transactions from an alternate RAC instance within the cluster, then it can obtain incorrect information on the status of distributed transactions until either all undo segments of the failed RM instance have been recovered or until all transactions of the originating RAC instance have been timed out. If a TM requests the status of a distributed transaction branch from a different instance to the one that started that distributed transaction branch, the likelihood is that the response returned will be "ORA-24756: Transaction does not exist" unless the transaction has either been timed out (e.g. become in-doubt) or been recovered as part of an database instance recovery.

Oracle database versions 9.0 and below

For Oracle database versions 9.0 and below, the XA connection must be opened with OPS_FAILOVER=T within the xa_open api.

On loss of a database instance, the TM must then use the xa_recover api to return a list of prepared transaction branches before the TM can proceed to resolve in-doubt transactions. Transactions that were not prepared will have been rolled back, transactions that have been completed will have been committed. The OPS_FAILOVER=T setting instructed xa_recover to call the internal Oracle procedure SYS.DBMS_SYSTEM.DIST_TXN_SYNC which blocked the TM until all of the undo segments of the failed database instance were recovered.

Careful design is needed to ensure that the TM will only failover to an alternate RM instance on loss of the original RM instance rather than due to errors such as loss of network connectivity.

Oracle 9iR2 RAC

For 9iR2, the same rules as for previous releases applied. But additionally the setting of the OPS_FAILOVER flag was made redundant by the imposition of an enhancement to the JDBC driver and database server from 9.2.0.6 such that the internal Oracle procedure SYS.DBMS_SYSTEM.DIST_TXN_SYNC was always called on xa_recover, xa_commit and xa_rollback.

A custom 9iR2 solution for XA and RAC for handling database instance failover is described within the "Best Practices for Using XA with RAC" Oracle whitepaper. Although database services could be used to easily maintain instance affinity, careful solution design was required to cater for the loss of network connectivity rather than loss of a database instance. On network loss, distributed transactions will exist until they time out, once timed out they must then become "in-doubt" transactions before they are recoverable from an alternate database instance.

9iR2 RAC also introduced the MAX_COMMIT_PROPAGATION_DELAY initialization parameter (default=700), which specifies the maximum amount of time (in hundredths of seconds) allowed before the system change number (SCN) held in the SGA of a database instance is refreshed by the log writer process (LGWR). It determines whether the local SCN should be refreshed from the lock value when getting the snapshot SCN for a query. Setting the parameter to zero causes the SCN to be refreshed immediately after a commit. This may be needed if recently committed data is required to be read from a database instance other than the one that committed the data.

Oracle 10gR1 RAC

For Oracle 10gR1, the use of Workload Management features made the design of an XA and RAC solution easier to manage without the need for custom scripting, this is described in the "Best Practices for Using XA with RAC" Oracle whitepaper.

From 10gR1 the XA connection had to be opened with the xa_open api using the RAC_FAILOVER=T setting rather than OPS_FAILOVER=T setting as in previous releases. The enhancement to the JDBC driver and database server such that the internal Oracle procedure SYS.DBMS_SYSTEM.DIST_TXN_SYNC was always called on xa_recover, xa_commit and xa_rollback was also added from the 10.1.0.3 patchset. But again the only supported mechanism for performing transaction recovery was for the TM to issue a call to xa_recover on the loss of a database instance.

With the use of singleton database services, issues due to loss of network connectivity disappeared, as the TM would always be directed to the instance where the database service was running.

Like 9iR2, 10gR1 also had the MAX_COMMIT_PROPAGATION_DELAY initialization parameter defaulted to 700.

Oracle 10gR2 RAC

For 10gR2, a new database service feature was introduced to support Distributed Transaction Processing (DTP), this optimized XA transaction recovery within the database. During normal running in-flight transaction prepare information is written to the undo area. If the transaction fails, then the prepared transaction information is pushed to the appropriate system tables to allow external recovery to take place. During normal operations, this push is done lazily. However, when using a DTP service all the in-flight "prepared" transactions of a failed instance are always pushed to database system tables before the DTP service is allowed to re-start on the any of the surviving database instances.

Within 10gR2 the requirement to always perform xa_recover on instance failover was withdrawn in cases where the TM had enough information about in-flight transactions to perform transaction recovery using xa_commit or xa_rollback.

Along with the DTP service feature becoming the supported, the internal Oracle procedure SYS.DBMS_SYSTEM.DIST_TXN_SYNC was made a null operation (no-op). Like previous versions, the 10gR2 method for failover recovery was documented in the "Best Practices for Using XA with RAC" Oracle whitepaper.

However, not all applications used the DTP Service feature and from 10.2.0.3 SYS.DBMS_SYSTEM.DIST_TXN_SYNC was re-enabled (via Patch 5968965).

In 10.2.0.4 the SYS.DBMS_SYSTEM.DIST_TXN_SYN procedure was subsequently renamed to DBMS_XA.DIST_TXN_SYNC as a security enhancement, thereby limiting user access to the powerful DBMS_SYSTEM procedure. Previously XA clients required execute privileges on DBMS_SYSTEM to invoke DIST_TXN_SYNC. This security enhancement required a change to both the server and client software (e.g. client OCI libraries and JDBC), such that the package DBMS_XA was called instead of DBMS_SYSTEM. This meant that to avoid user permission errors on database failover, all XA Client users now required execute privileges on DBMS_XA rather than DBMS_SYSTEM. Similarly to avoid compatibility issues, both the client and server software versions needed to refer to the same PL/SQL package either DBMS_XA or DBMS_SYSTEM.

Also at 10gR2 the MAX_COMMIT_PROPAGATION_DELAY parameter was deprecated and was only retained for backward compatibility. It is defaulted to 0 causing the SCN to be refreshed immediately after a commit ("broadcast on commit") such that recently committed data will be immediately available on a database instance other than the one that committed the data.

Oracle 11gR1 RAC

11gR1 introduced cluster-wide global transactions, enabling distributed transactions to be processed on any database instance within a RAC cluster. This introduced a number of new features:

- A global DX (Distributed Transaction) Enqueue for tightly coupled transactions;
- Enabling read consistency for tightly coupled transactions, such that pre-committed changes made in one database instance are available in another;
- A new GTXn background process and corresponding initialization parameter (GLOBAL_TXN_PROCESSES) for managing the DX enqueues and performing cross-instance two phase commits.

The DX (Distributed Transaction) Enqueue and new read consistency features are discussed in the previous section on Database Locking.

The GLOBAL_TXN_PROCESSES parameter specifies the initial number of GTXn background processes for each Oracle RAC instance. The number of GLOBAL_TXN_PROCESSES is automatically managed by the database.

However, there is a known performance issue with starting and stopping GTX processes which can give rise to excessive wait events on "Global transaction acquire ins" (see Note: 804426.1), the workaround is to set underscore parameter _DISABLE_AUTOTUNE_GTX=TRUE to disable the GTX auto tuning functionality.

Cluster-wide global transactions is the default at 11gR1 RAC when COMPATIBLE >= 11.0 and can only be disabled by setting the underscore parameter _CLUSTERWIDE_GLOBAL_TRANSACTIONS=FALSE. If this parameter is set to FALSE, then the default 10gR2 distributed transaction processing mechanism will be used.

Oracle 11gR2 RAC

There were no new features or constraints added to 11gR2 regarding XA.

Oracle Account Privileges

Any database account performing distributed transactions must have the following privileges:

- Have permissions (GRANT) to read the DBA_PENDING_TRANSACTIONS system view.
- Have permissions (GRANT) to the "FORCE ANY TRANSACTION" privilege, for database users who need to manipulate distributed transactions created by other database users.

Additionally, when using RAC, user accounts must have the following privileges:

- Have permissions (GRANT) to execute the DBMS_SYSTEM system package (for 9.2.0.8 or 10.2.0.3 and below, unless patches for 5945463 – Server, 5965349 – OCI /C Library, 5892995 JDBC have been applied) or the DBMS_XA system package (10.2.0.4 or higher for database server, client OCI libraries or JDBC drivers).
- Use Client and Server versions of the same patchset or with the appropriate patches applied to avoid mixing the use of the DBMS_SYSTEM package and DBMS_XA package within the same deployment.

XA and Database Links

The use of XA and database links mixes both XA and Oracle-controlled distributed transactions. There are version dependent constraints on using database links with XA, the manuals for all database versions within the scope of the transaction need to be checked for any restrictions. Generally, most versions have the following restrictions with regard to the use of database links:

- All database connections must use the shared server configuration.
- The other database being accessed must be another Oracle Database.

If these restrictions are not satisfied, then when a database link is used within an XA transaction it will create an O/S network connection in the Oracle Server that is directly connected to the TM server process. When the TM completes a particular service or RPC it may need to detach itself from the database such that the connection can be re-used by other services or RPCs. As O/S network connections cannot be moved from one process to another this prevents the TM service or RPC from detaching from the database. This can result in a response of "ORA-24777 use of non-migratable database link not allowed" from the database.

Oracle Real Application Testing and Distributed Transactions

Real Application Testing (RAT) was introduced in 11gR1 and consists of several components used to capture, analyze, and replay production database transactions. RAT allows the full impact of upgrades and system changes to be tested on a non-production system by replaying production database workload. However, RAT is deployed to one database at a time and as such can only perform transactions on a single database, any distributed transactions that are captured by RAT are therefore replayed as local transactions. Therefore the full benefits of RAT cannot be realized within an architecture that uses distributed transactions, since the workload mix and data contention possibilities are inherently different to those that might occur in the production workload.

PL/SQL and XA

Since Oracle Database 11gR1, PL/SQL can now utilize the functionality available in the XA interface to support transactions involving multiple resource managers, such as databases and queues. PL/SQL is now able to switch or share transactions across SQL*Plus sessions or processes using the XA/Open interface library contained within the DBMS_XA package (e.g. XA_START to join a transaction, XA_END to leave a transaction).

Using XA and the Oracle Database from the Middle Tier

XA and Oracle WebLogic Server / Fusion Middleware 11gR1

Oracle Weblogic Server implements tightly coupled XA transactions via the Oracle JDBC driver using the standard JTA / JDBC XAResource interface. This cannot be overridden by application developers.

Oracle WebLogic Server Multi-Pools

For RAC, the recommendation at Oracle WebLogic Server 11gR1 (and all versions since 8.1.5) is to use Oracle WebLogic Server multi-datasources. Both an Oracle whitepaper "Oracle WebLogic Server 10.3 and Oracle Real Application Clusters (RAC)" and an OTN "how to guide" have been published on the Oracle Technology Network (OTN) describing the use of Oracle WebLogic Server multi-datasources and Database Services. Further information is also available within the manuals for the relevant release of Oracle WebLogic Server and is available on-line on OTN.

The key architectural constraint Oracle WebLogic Server multi-datasources have on RAC is the removal of any form of "Server Side" load balancing for Oracle database connections. Each RAC database instance is configured such that it's services are only advertised by the local database Listener on the same node/server (local_listener) as the corresponding database instance. The RAC default of configuring all database Listeners to advertise all the database services available across all database instances within the RAC cluster is disabled (remote_listener is set to NULL). This allows individual connection pools within a Oracle WebLogic Server multi-pool to be directed to an individual database instance. As part of the multi-datasources feature Oracle WebLogic Server pins an individual JTA transaction to a particular physical pool.

From Oracle WebLogic Server 11gR1 (10.3.1), the pinning of individual connection pools to a RAC database instance can be achieved by using the "INSTANCE_NAME= " parameter within the JDBC connection string as follows:

url="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)(HOST=VIP1)(PORT=1521)))(CONNECT_DATA=(SERVICE_NAME=SERV1)(INSTANCE_NAME=INST1)))

This enables the Oracle database parameter remote_listener to be set to a non-null value and thereby allow database services to advertised across all database listeners within the RAC cluster, whilst restricting an individual connection pool to a specific database instance. All connections with the same Multi-Pool data source must use the same database service name.

Multi-datasources should be used for the failover and load balancing of database connections across multiple database instances when using XA. Oracle WebLogic Server will load balance requests across data sources but pin individual XA transactions to a single data source.

Oracle WebLogic Server XA Transaction Recovery with Oracle RAC

Within a RAC environment, the recovery operations performed by Oracle WebLogic Server will depend on at what point in the transaction the RAC database instance fails.

If the RAC database instance failure occurs before the application calls for the transaction to be committed or the application has requested a commit but the prepare operation(s) have not yet been completed, then Oracle WebLogic Server rolls back the transaction and sends the application an exception for the failed transaction.

If the RAC database instance failure occurs during the commit phase after the prepare operation(s) are complete, then Oracle WebLogic Server will retry the completion of the transaction using xa_commit. Note that the connection will be blocked when using xa_commit by the database procedure DIST_TXN_SYNC, which will delay the response of xa_commit until all distributed transactions have been recovered.

Oracle Weblogic Server also allows for a RAC brownout period, where any XA operations that could result in a XA_NOTA response from the database (e.g. current transaction is not recognized as a valid transaction by the resource manager) is translated into XA_RMERR (e.g. resource manager error) for the period of the brownout processing to give the database time to recover all outstanding XA transactions. The length of this brownout period is controlled by XARetryDurationSeconds and the transaction retry interval within the brownout period by XARetryIntervalSeconds. Once the brownout period has expired, any subsequent XA_NOTA response from the database will be interpreted as the operation having completed successfully". (Note that XA_NOTA is a valid response if, e.g. the TM re-issues a commit operation but the RM has already committed the branch and forgotten about it).

Oracle WebLogic Server Transaction Log File (TLOG)

After prepare, Oracle WebLogic Server writes the outcome of the prepare phase to its TLOG (transaction log file) before moving into the commit phase. The TLOG is persisted on disk. By default, local commits of multiple resources in Oracle WebLogic Server will be done in parallel using multiple threads, this is controlled by the (non-documented) JTAMBean attribute parallelXAEnabled.

Once committed on all RMs, Oracle WebLogic Server removes the entry from the TLOG as part of it's "forget" processing to remove completed transactions.

If a highly available Oracle WebLogic Server XA configuration is required, consideration must be given to the recovery of the TLOG between physical servers. This can be achieved using various file system / logical volume management methods such as:

- Remounting SAN disks on to an alternate server using server based clustering software (e.g. Oracle Clusterware, IBM HACMP, HP ServiceGuard, Veritas Clustering, etc) or the Whole Server Migration (WSM) feature of Oracle WebLogic Server;
- Attaching a Network Attached Storage (NAS) device to share mount points across multiple servers (e.g. NetApp, EMC Clarion etc) or using a Clustered File System (e.g. Veritas CFS, OCFS2, Oracle's 11gR2 ASM Clustered File System (ACFS), etc) to avoid remounting disks across servers;

However, the possible performance impact of using a shared file system rather than dedicated disk should be considered when deploying a high throughput system requiring fast access to disk.

Oracle WebLogic Server's Whole Server Migration and Automatic Service Migration features are discussed in the Oracle whitepaper "Automatic Service Migration in Oracle WebLogic Server" also published on the Oracle Technology Network (OTN).

For Disaster Recovery, the TLOG should be protected by synchronous disk array based replication (e.g. EMC's SRDF, or Continuous Access used within enterprise disk arrays from HP/Sun/Hitachi). When using synchronous disk array based replication for Disaster Recovery, all participants (both TM and RM) of the XA transaction should be protected within the same disk replication "Consistency Group". This ensures all writes to disks within the "Consistency Group" are sequenced correctly at the remote location, such that all XA data sources and TM logs are in a consistent state relative to each other.

If a multiple replication methods are used for Disaster Recovery, then care must be taken to ensure that the state and consistency of all the participants (both TM and RM) of the XA transaction can be maintained on failover to the Disaster Recovery site.

Oracle WebLogic Server Logging Last Resource (LLR)

An additional option for XA transactions is Oracle WebLogic Server's Logging Last Resource (LLR) transaction optimization through JDBC data sources. LLR is a performance enhancement option that enables one non-XA resource to participate in a distributed transaction with the same ACID guarantee as XA. In a distributed transaction with an LLR participant, the Oracle WebLogic Server transaction manager follows these basic steps:

- Calls prepare on all other (XA-compliant) transaction participants.
- Inserts a commit record to a table on the LLR participant (rather than to the file-based transaction log).
- Commits the LLR participant's local transaction (which includes both the transaction commit record insert and the application's SQL work).
- Calls commit on all other transaction participants.

After the transaction completes successfully, Oracle WebLogic Server lazily deletes the database transaction log entry as part of a future transaction.

LLR can be useful when wishing to avoid using XA on a particular data source, either because it is unsupported or adds complexity to the overall design. However, the LLR participant must be able to maintain the Oracle WebLogic Server commit record, therefore it cannot be used with queue based resource managers.

XA and Oracle 10gAs Application Server / Fusion Middleware 10g

An Oracle whitepaper "Oracle SOA Suite XA and RAC Database Configuration Guide" has been published on the Oracle Technology Network (OTN) describing how to configure SOA Suite for XA. Although the paper describes configuring an XA database user with permissions (GRANT) to execute the DBMS_SYSTEM system package for 10.2.0.3, this may require permissions on the DBMS_XA system package for later versions of the database.

10gAS 10.1.3.1.0 introduced a Middle-Tier Two-Phase Commit (2PC) Coordinator, whereby transaction coordination functionality was now located in the 10gAs engine, replacing the in-database coordination. The in-database coordination was deprecated from 10gAS 10.1.3.1.0+. Prior to 10.1.3.1.0, the only mechanism available was in-database coordination using the database's distributed transaction manager.

The OC4J transaction log file can be persisted to either a database or a file system. The database provides greater flexibility if there is a need to provide a highly available XA configuration.

10gAS's Last Resource Commit (LRC)

An additional option for XA transactions in 10.1.3.1.0 is 10gAS's Last Resource Commit (LRC) transaction optimization through JDBC data sources. LRC is a performance enhancement option that enables one non-XA resource to participate in a global transaction without the same ACID guarantee as XA. In a global transaction with an LRC participant, the OC4J transaction manager follows these basic steps:

- Calls prepare on all other (XA-compliant) transaction participants.
- Commits the LRC participant's local transaction (e.g. the emulated XA resource).

If the one-phase commit on the emulated XA resource completes successfully, then the OC4J transaction manager logs a decision to commit the distributed transaction and calls commit on each of the two-phase-commit resources. Otherwise, if the one-phase commit on the emulated XA resource fails, then the OC4J transaction manager logs a decision to rollback the distributed transaction and calls rollback on each of the two-phase-commit resources.

Although the LRC optimization works correctly for the great majority of the time, there is some risk in using the last resource commit optimization. If a failure occurs during the transfer of control to the non-XA compliant resource, the transaction coordinator has no way of knowing whether the resource committed or rolled back the transaction. This could lead to heuristic completion of the transaction, which could be very hard to detect or rectify.

Typical Architecture and Design Problems with XA

The typical architecture and design problems encountered when using XA are:

- Recovering multiple data sources such as databases and file system based queues to a transactionally consistent point from a backup. File system based queues can be problematic to manage within a backup and recovery strategy. Although databases can be recovered to the same point in time, they may not be transactionally consistent (e.g. a distributed transaction is committed on one, but only prepared on the other).
- Recovering data sources that are replicated asynchronously for Disaster Recovery. The data sources are left in an inconsistent state at different points in time and transactional consistency. This results in the same issue that occurs when recovering from a backup.
- The XA Race Condition - the XA specification doesn't state what order a TM requests RMs to commit or in what time frame the RMs must commit. It only states that all RMs who have agreed to commit, must commit the data. Therefore there is the possibility of a race condition occurring, especially if processing is dependent on the order of writes to one or more RMs. E.g. Message Driven Bean (MDB) type architectures, where an independent process monitors a JMS queue, identifies a new message and tries to process it, only to find the corresponding updates for it to process are not yet available on the database.
- Accidentally using DDL within a distributed transaction e.g. using temporary tables, the CREATE TABLE command is DDL, which involves an implicit transaction.
- Designing in database contention, as Oracle-controlled distributed transactions lock all changed rows for an indeterminate period between prepare and commit, each transaction should be designed to minimise update contention on both indexes and data. Otherwise transactions can block each other causing spikes in load and degraded performance if data with a similar profile is processed simultaneously. E.g. inserting into the same range of a non-unique index.

Recovering Oracle databases

Within an Oracle database environment, an Oracle-controlled distributed transaction will force all databases within the transaction to perform coordination and synchronization of SCNs. The databases then can be recovered to the SCN of the last periodic transaction to occur across all databases to provide transactional consistency. This can be achieved by performing a periodic Oracle distributed transaction that updates a table on each of the databases that need synchronization. However, this means incomplete or partial recovery will occur, as all transactions made since the last periodic update will be lost, unless there is a custom mechanism to replay all missing transactions. Although this mechanism enables Oracle databases to be recovered to a transactional consistent point in time, it doesn't include non-Oracle database resources that may be part of the distributed transaction.

Avoiding the XA Race Condition

To avoid the XA Race condition described above, the design may need to employ one or more of the following techniques (in no particular order):

- A retry mechanism to allow all RMs time to commit.
- Use tightly coupled transaction branches for both the queue and critical data on the same RM instance, this will result in a single commit. (Requires TM to support tightly coupled transaction branches, and this may have a performance impact)
- An in-built time delay to allow all RMs time to commit.
- Use Oracle WebLogic Server's JMS "scheduled message (delivery time)" enhancement to schedule the JMS message for delivery in the future (e.g. achieves an in-built delay).
- For 9iR2 RAC and 10gR1 RAC, set MAX_COMMIT_PROPAGATION_DELAY=0 if recently committed data needs to be read from a database instance other than the one that committed the data.
- The use of Oracle Weblogic Server's Logging Last Resource optimization - enables one non-XA resource to participate in a global transaction with the same ACID guarantee as XA to force the JMS queue to be the last committed RM. (e.g. the database is the LLR participant and is committed first).
- The use of 10gAS's Last Resource Commit (LRC) feature of OC4J transaction manager - enables one non-XA resource to participate in a global transaction. LRC can be used to force the JMS queue to be the last committed RM. (e.g. the database is the LRC participant and is committed first). However, the OC4J transaction manager is only available in 10gAS 10.1.3.1.0+ and this does not provide the same ACID guarantee as XA.

Conclusion

The use of distributed or “global” transactions within any system architecture requires careful consideration to avoid data recoverability issues, cascading failures due to in-doubt transactions and the degradation of system performance.

When using any type of distributed or “global” transactions ensure your architecture:

- Can be recovered in a transactionally consistent state from a backup within the required Recovery Time Objective;
- Does not expect a transaction to be simultaneously applied to all data sources (e.g. XA Race Condition);
- Does not perform actions that require implicit transactions (e.g. DDL) within an XA transaction;
- Avoids building in contention or dependencies for small subsets of data;
- If Highly Available, does not entrap data required by the Transaction Manager (e.g. TLOG) to recover transactions after a failure;
- Avoids the manual resolution of transactions where possible, since this could lead to a heuristic outcome where the individual Resource Managers may contain different versions of the same data. The Transaction Manager where possible should be allowed to complete any outstanding transactions.

ORACLE

Oracle is committed to developing practices and products that help protect the environment

XA and Oracle controlled Distributed Transactions
June 2010
Author: Tony Flint

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com