An Oracle White Paper
June 2013

Oracle Database 12*c* Application Continuity for Java

1 - Introduction

Have you ever experienced paying twice the same flight ticket, the same article or your taxes? This must be a hard problem to solve! Have you ever wanted the infrastructure to just deal with database failure and not ask you to restart your transaction from the beginning? If it was easy, it would have been done!

Upon database outages (hardware, software, network, or storage failure), four problems confront applications: hangs, errors handling, determining the outcome of in-flight work (i.e., last COMMIT), and the resubmission in-flight work.

In previous releases, as explained in "*Application Failover with Oracle database 11g*"[1] white paper, Fast Application Notification (FAN) helps deal with hangs however these mechanisms do not address the third and fourth issues. Oracle database 12*c* pushes the envelope further with formalized "Recoverable Errors", Transaction Guard for dealing with the outcome of in-flight work, and Application Continuity for resubmitting in-flight work.

If you are application developer, database and system administrator, integrator or ISV looking to better exploit Oracle RAC and Active Data Guard to achieve maximum application availability, this is the paper for you (although with a Java focus).

[1] http://www.oracle.com/technetwork/database/app-failover-oracle-database-11g-173323.pdf

2 - New and Enhanced Concepts

Recoverable Error

Oracle database 12*c* exposes a new error attribute `is_recoverable` that applications can use to determine if an error is recoverable or not without maintaining their own list of error codes (e.g., `ORA-1033, ORA-1034, ORA-xxx`). JDBC throws `SQLRecoverableException`, which tells the application or the driver that the database connection is no longer valid and that a new connection must be obtained. Constraint violation is an example of unrecoverable error.

Database Request (Unit of Work)

A unit of work submitted by the application which may have zero, one or multiple COMMITS statements. The typical database request starts when a connection is chevked-out of th connection pool then include a combination of SQL calls (queries, DMLs), local calls, remote procedure calls, and a COMMIT statement,

Logical Transaction ID (LTXID)

A logical value issued by the RDBMS and associated with every transaction. LTXIDs are only incremented (by the RDBMS) when the corresponding transaction is committed. The only purpose of LTXIDs is to help make a reliable determination of the outcome of the last COMMIT statement.

Mutable Functions

Non-deterministic functions that can change their results each time they are called e.g., `SYSDATE, SYSTIMESTAMP, SEQUENCES, SYS_GUID`. Applications may or may not be sensitive to the exact value of mutable functions in case of the resubmission of the same unit of work.

FAN - HA Events & Notification

RAC and Data Guard emit HA events such as NODE DOWN, INSTANCE UP/DOWN, SERVICE UP/DOWN, etc; upon emission, these events are sent/notified to subscribers (drivers, applications) using Oracle Notification Services (ONS). Oracle JDBC drivers and the Universal Connection Pool subscribe to all HA events types when Fast Connection Failover is enabled and act upon. Java applications and third party drivers and connections pools may subscribe directly to DOWN events, using use `SimpleFAN.jar` (UP events are not currently supported).

3 - Transaction Guard for Java

Problems to solve

Address the 3rd issues that is: make a reliable determination of the outcome of the in-flight work. Following a break in communication between Java applications and the RDBMS, the outcome of last COMMIT operation is often doubtful and leads to the resubmission of work already committed thereby leading to paying twice or several times the same article or service. This problem is challenging because simply checking the outcome at a given time does not

guarantee a reliable outcome, as the COMMIT statement may eventually go through, after the check. Transaction Guard is an API for checking the outcome of the last COMMIT operation, in a fast, reliable and scalable manner. The secret sauce is that when the application checks the status of the actual transaction using the LTXID, if not COMMITed, the RDBMS will return the status and in addition block it from COMMITing. If Transaction Guard says “Un-COMMITed”, it stays this way; rock-solid!

Typical usage

1) Upon database instance crash: (i) death of sessions belonging to that instance; (ii) Fast Application Notification immediately sends the event to subscribers; (iii) application gets an error quickly; (iv) the connection pool (UCP) removes orphan connections from the pool

2) server-side package and procedure to help determine the outcome of the last COMMIT

a) New DBMS_APP_CONT package
b) Here is a sketch of the RDBMS and application interaction

```
If “recoverable error”
then
 Get last LTXID from dead session or from your JDBC callback
 Obtain a new database session
 Call DBMS_APP_CONT.GET_LTXID_OUTCOME with last LTXID to obtain
                    COMMITTED  and USER_CALL_COMPLETED status
 If COMMITTED and USER_CALL_COMPLETED
       Then return result
 ELSEIF COMMITTED and NOT USER_CALL_COMPLETED
       Then return result with a warning
 ELSEIF NOT COMMITTED
      Cleanup and resubmit request
       Note: the RDBMS prevents the transaction from committing (RETENTION_TIMEOUT)
 END
```

And here is the definition of GET_LTXID_OUTCOME

```
CREATE OR REPLACE PROCEDURE get_ltxid_outcome(
                  client_ltxid        IN  RAW,
                  committed           OUT INT,
                  user_call_completed OUT INT)
AS
   committed_b            BOOLEAN;
   user_call_completed_b  BOOLEAN;
BEGIN
 dbms_app_cont.get_ltxid_outcome(client_ltxid, committed_b,
                                              user_call_completed_b);
 IF committed_b=TRUE THEN committed := 1;
 ELSE
    committed := 0;
 END IF;
 IF user_call_completed_b=TRUE THEN
      user_call_completed := 1;
 ELSE
      user_call_completed := 0;
 END IF;
END;
/
```

Ensure that execute permission on the DBMS_APP_CONT package has been granted to the database users that will call GET_LTXID_OUTCOME:

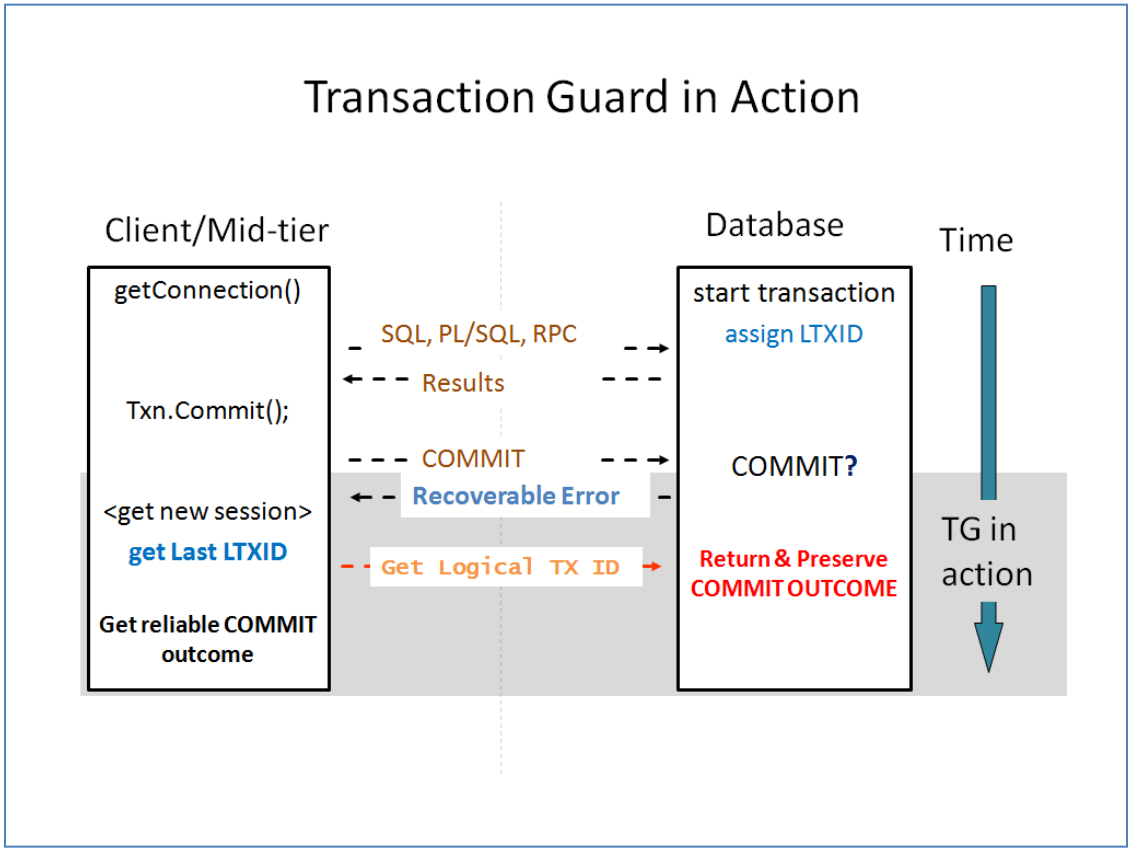
```
GRANT EXECUTE ON DBMS_APP_CONT TO <user-name>;
```

3) Application Usage (Java)

```
addLogicalTransactionIdEventListener()//register a listener to
                                  // Logical Transaction Id events

LogicalTransactionId firstLtxid = oconn.getLogicalTransactionId();
     //sent by the server in a piggy back message and hence this
     //method call doesn't make a roundtrip.
  …
CallableStatement cstmt = oconn.prepareCall(GET_LTXID_OUTCOME);
     // procedure defined above
  …
  committed = cstmt.getBoolean(1);
```

The following picture shows Transaction Guard in action



Supported Transaction Types

Transaction Guard supports the following transaction types: local transactions, DDL and DCL transactions, distributed and Remote transactions, parallel transactions, commit on success (auto-commit), and PL/SQL with embedded COMMIT.

Exclusions

In this release, Transaction Guard excludes the following transaction types

- Intentionally recursive transactions and autonomous transactions intentionally so that these can be re-executed.
- XA transactions
- Active Data Guard with read/write DB Links for forwarding transactions
- Golden Gate and Logical Standby

Configuration

RDBMS
On Service

- `COMMIT_OUTCOME`: values {`TRUE or FALSE`}, default is `FALSE`; applies to new sessions
- `RETENTION_TIMEOUT`: Units in seconds, default is 86400 (24 hours); maximum value is 2592000 (30 days)

```
SQL>
declare
params dbms_service.svc_parameter_array;
begin
  params('COMMIT_OUTCOME'):='true';
  params('RETENTION_TIMEOUT'):=604800;
  dbms_service.modify_service('[your service]',params);
end;
/
```

4 - Application Continuity for Java

Problem to Solve

Address the 4th issue that confronts applications upon RDBMS instance failure, in other words, the resubmission of in-flight work resulting in masking database instance outage (hardware, software, network, and storage) to applications.

Solution

Application Continuity is an out of the box solution with the following building blocks: the unit of work (a.k.a. "*database request*"), the JDBC replay data source, Transaction Guard for Java, and RDBMS High Availability configurations (RAC, Data Guard).

Application Continuity works as follows:

1. Transparently captures in flight work a.k.a. “database request”, during normal runtime
2. Reconnect phase: upon RDBMS instance outage or site failure, if recoverable errors then Transaction Guardis used under the covers then the driver reconnects to a good RDBMs instance (RAC) or disaster recovery `site` (ADG). In order for the replay datasource to transparently replace the invalid/dead connection with a new one belonging to the surviving good instance or site, the application must be using Java interfaces and not concrete classes.
3. Replay phase: the driver and RDBMS cooperate to replay the in-flight work captured during normal runtime (until the point of failure).

When successful, Application Continuity masks hardware, software, network, and storage outages to applications; end-users will only observe/experience a slight delay in response time.
When not successful, the original error is re-thrown by the driver to the application.

Table 1 below summarizes how Application Continuity works

TABLE 1. PROCESSING PHASES OF APPLICATION CONTINUITY

NORMAL RUNTIME	RECONNECT	REPLAY
• Identifies database requests • Decides what is replayable and what is not • Builds proxy objects • Holds original calls with validation	• Ensures request has replay enabled • Handles timeouts • Creates a new connection • Validates target database • Uses Transaction Guard to enforce last outcome	• Replays held calls • During replay, ensures that user visible results match original • Continues the request if replay is successful • Throws the original exception if replay is unsuccessful

RDBMS and Application Configuration

In Oracle database 12*c* Release 1, *Application Continuity* is available with JDBC-Thin and UCP. These Oracle clients transparently demarcate units of work on connection check-out/check-in whereas third party drivers and connection pools must explicitly demarcate the units of work (a.k.a. “*database requests*”) using `beginRequest()/endRequest()` calls.

Application Continuity for Java requires standard JDBC interfaces instead of deprecated `oracle.sql.*` concrete classes: `BLOB, CLOB, BFILE, OPAQUE, ARRAY, STRUCT,` or `ORADATA` (see My Oracle Support Note 1364193.1[2] for the deprecation notice).

[2] See JDBC interfaces for Oracle types:
https://support.oracle.com/CSP/main/article?cmd=show&type=NOT&id=1364193.1

1- Java Application
sets the new replay data source either in property file, as follows, or inline.
```
datasource=oracle.jdbc.replay.OracleDataSourceImpl
```

2- Enable Application Continuity on the database service
The following settings must be performed on the specific service; the default service is not recommended and will not work.

```
FAILOVER_TYPE = TRANSACTION
REPLAY_INITIATION_TIMEOUT = 1800
FAILOVER_DELAY = 3 seconds
FAILOVER_RETRIES = 60 retries
SESSION_STATE_CONSISTENCY = DYNAMIC
COMMIT_OUTCOME = TRUE
```

Exclusions, Restrictions, Side Effects and Design Considerations

Although Application Continuity may be enabled through properties files (datasource) and servers-side configuration, architects and application designers need to make some assessment in terms of exclusions, restrictions, side effects and other design considerations.

Exclusions and Restrictions

Application Continuity is not supported under the certain conditions summarized in table 2 , below.

TABLE 2. THREE LEVELS OF RESTRICTIONS APPLY FOR APPLICATION CONTINUITY

GLOBAL	REQUEST	TARGET DATABASE
Does not support : • XA • Deprecated Java concrete classes • Default database service	• For Java streams, replay is on a “best effort” basis • Request-level disable for Active Data Guard with read/write database links • Request-level disable for — Alter System — Alter Database	Does not support: • Logical Standby • Golden Gate

Beyond exclusions, under some circumstances, Application Continuity replay may be temporarily disabled either explicitly to avoid replaying critical code segment (e.g., check printing).

Table 3 below summarizes the explicit or implicit restrictions (replay temporarily disabled).

TABLE 3. WHEN IS APPLICATION CONTINUITY DEACTIVATED (NOT SUPPORTED)

GLOBAL	REQUEST	TARGET DATABASE
Any calls in same request after – • successful commit in dynamic mode (the default) • a restricted call • disableReplay API	• Error is not recoverable • Timeouts — Replay initiation timeout — Max connection retries — Max retries per incident • Target database is not valid for replay • Last call committed in dynamic mode	• Validation detects different results

Side Effects

Some RDBMS calls have side effects that might not be welcomed when the unit of work is replayed. Examples of such calls include:

- Autonomous transactions
- UTL_HTTP , UTL_URL
- UTL_FILE, UTL_FILE_TRANSFER - files operations
- UTL_SMPT, UTL_TCP, UTL_MAIL - sending messages
- DBMS_PIPE, RPCs - to external sources
- DBMS_ALERT - email or other notifications

Applications may use `disableReplay()/enableReplay()` APIs to temporarily disable and re-enable replay to prevent undesirable side effects.

Callbacks

Applications may set specific session states (NLS, transaction isolation, etc) before issuing units of work (a.k.a. database requests), or outside of it. Oracle Universal Connection Pool (UCP) and Oracle Weblogic Server furnish connection labeling for such use cases; the application designer may also register its own callback with UCP Connection Initialization Callback or Weblogic Administration Console.

Conclusion

This paper walked you through Transaction Guard and Application Continuity for Java with Oracle Database 12c. Using these features, architects, developers, DBAs and ISVs may now implement robust, reliable and replay-able applications which will avoid the inconvenience of paying twice the same flight ticket, book, taxes and attempts to safely mask database outages.

Oracle Database 12*c* Application Continuity for Java
June 2013
Author: Kuassi Mensah
#kmensah

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

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

Hardware and Software, Engineered to Work Together