Java Programming with Oracle Database 12c RAC and Active Data Guard

High Availability and Scalability Concepts

ORACLE WHITE PAPER FEBRUARY 2015

Table of Contents

Introduction

This paper is a supporting document for designing and deploying web applications with Java EE containers (Tomcat, WebSphere and so on) in the face of planned and unplanned database outages[2]. It is an update, a follow up to "**Application Failover with Oracle Database 11g".**[1]

To support high-availability and load balancing solutions, Oracle Database 12c furnish the following features, mechanisms, and concepts.

- Universal Connection Pool (UCP)
- Fast Application Notification (FAN)
- Oracle Notification Service (ONS)
- Fast Connection Failover (FCF)
- Logical Transaction ID (LTXID)
- Database Request
- Recoverable Errors
- Mutable Functions
- Transaction Guard (TG)
- Application Continuity (AC)

If you are a Java developer or architect looking to exploit high-availability and scalability mechanisms in Real Application Cluster (RAC), Exadata, Active Data Guard (ADG), and/or Global Data Services (GDS) environments, this is the paper for you.

Universal Connection Pool (UCP)

Introduced with Oracle Database 11g, Oracle Universal Connection Pool for Java (UCP) is a features-rich Oracle Database component; a replacement for the Implicit Connection Cache (ICC) which has been de-supported in Oracle Database 12c. UCP has been designed to work with RAC and Active Data Guard and Global Data Services[3] thereby furnishing connections availability, failover and workload balancing. The latest UCP in Oracle Database 12c has been profoundly enhanced for faster connection management (heap, enqueue, dequeue), workload balancing, high-availability and can be used against older database releases (features with Oracle Database 12c support won't be available).
To use UCP, Java applications simply need **ucp.jar** in their class path, along with **ojdbc7.jar** (for JDK 7 and JDK 8) or **ojdbc6.jar** (for JDK 6). Refer to UCP Developer's guide[4] for more details.

UCP has numerous properties for tuning the behavior of the connection pool. The base properties are summarized in the table hereafter.

Property Name	Property Type	Property Details
setMinPoolSize	java.lang.String	Required. Set the appropriate minimum pool size
setMaxPoolSize	java.lang.String	Required. Set the appropriate maximum pool size

2 See the JDBC/UCP portal @ http://www.oracle.com/technetwork/database/application-development/index-099369.html

1 http://www.oracle.com/technetwork/database/app-failover-oracle-database-11g-173323.pdf
3 For GDS, please specify global service name and region in connect URL as illustrated hereafter

```
(DESCRIPTION= (ADDRESS_LIST= (LOAD_BALANCE=ON) (FAILOVER=ON) (ADDRESS=(GDS_protocol_address_information))
(ADDRESS=(GDS_protocol_address_information))) (CONNECT_DATA= (SERVICE_NAME=global_service_name) (REGION=region_name)))
```

4 http://docs.oracle.com/cd/E16655_01/java.121/e17659.pdf

setInitialPoolSize	java.lang.String	Required. Should be closer to minPoolSize
setFastConnectionFailoverEnabled	java.lang.Boolean	Required. Set it to TRUE
setONSConfiguration	java.lang.String	Optional. Required for pre 12c Oracle Database version

Other useful UCP timeout properties are mentioned here. Refer to PoolDataSource class for the complete list of UCP properties.

Property Name	**Property Type**	**Property Details**
setMaxConnectionReuseTime	java.lang.Long	Allows connections to be gracefully closed and removed from the pool after a connection has been in use for a specific amount of time
setMaxConnectionReuseCount	java.lang.String	Allows connections to be gracefully closed and removed from the pool after a connection has been borrowed a specific number of times.
setAbandonedConnectionTimeout	java.lang.String	Allows borrowed connections to be reclaimed back into the pool after a connection has not been used for a specific amount of time
setTimeToLiveConnectionTimeout	java.lang.String	Sets the maximum time (in seconds) a connection may remain in-use.
setConnectionWaitTimeout	java.lang.String	Sets the amount of time to wait (in seconds) for a used connection to be released by a client.
setInactiveConnectionTimeout	java.lang.String	Specifies how long an available connection can remain idle before it is closed and removed from the pool
setTimoutCheckInterval	java.lang.String	Controls how frequently the timeout properties (mentioned in this table) are enforced.

For more details, please refer to the UCP documentation[5], on Oracle Technology Network (OTN).

Fast Application Notification (FAN)

Traditionally, upon database events (service, instance, node: DOWN or UP) applications hang until the timeouts kick-in then Java applications and Servlets receive the relevant SQL exception. Introduced with Oracle Database 11g, Fast Application Notification (FAN) events are notified to subscribers (drivers, containers) in a fast and reliable manner using the Oracle Notification System (ONS):

- Down – Service/Node down in 50ms or less to invoke failover
- Planned Down – Drains sessions with no user interruption
- Up – Relocates work when services resume
- RLB % - Hint to balance sessions locally for RAC or globally with GDS
- Affinity – Hint to service connection requests from the same instance/node maintain conversation locality

Drivers, containers, and frameworks subscribe and receive FAN events via ONS (described hereafter) either by setting the `FastConnectionFailoverEnabled` datasource property to true (as illustrated hereafter), or by using the Java FAN APIs through the `SimpleFan.jar`.

In addition, FAN callout scripts can be configured on the database tier and allow server-side actions when FAN events happen.

5 http://docs.oracle.com/database/121/JJUCP/toc.htm

Oracle Notification Service (ONS)

As already indicated, FAN events are notified to subscribers using Oracle Notification System (ONS). ONS is enabled on the RDBMS server via a configuration file usually located at **$ORACLE_HOME/opmn/conf/ons.config** or **$ORACLE_CONFIG_HOME/opmn/conf/ons.config.<hostid>**. For RAC, the configuration file is under **GRID_HOME/opmn/conf/config.ons**. The Oracle installer creates this configuration file by default.
Here is an example.

```
# First three values are required
localport=4100
remoteport=4200
nodes=racnode1:4200, racnode2:4200, racnode3:4200
```

Note that last line "`nodes=racnode:4200, racnode2:4200, racnode3:4200`" is used as value for one of the UCP properties `ONSConfiguration` for pre-12c database versions.

Monitor if ONS daemon is running on the RDBMS server using the following command.

```
$srvctl status nodeapps |grep ONS
  ONS is enabled
  ONS daemon is running on node: <racnode1>
  ONS daemon is running on node: <racnode2>
  ONS daemon is not running on node: <racnode3>
$crsctl status res ora.ons
  NAME=ora.ons
  TYPE=ora.ons.type
  TARGET=ONLINE            , ONLINE
  STATE=ONLINE on <racnode1>, ONLINE on <racnode2>
$srvctl config nodeapps | grep ONS
  ONS exists: Local port 6100, remote port 6200, EM port 0, Uses SSL false
  ONS is enabled
  ONS is explicitly enabled on nodes: <racnode1>
  ONS is explicitly disabled on nodes: <racnode2>
$setenv ORACLE_CONFIG_HOME $T_WORK/ons0
$onsctl ping or onsctl debug
```

Fast Connection Failover (FCF)

Upon database outage, a FAN DOWN event it issued by one of the good standing RAC instance or AD site. UCP and Java Servlets/applications collaborate to implement connection failover as follows:

a) UCP quickly marks un-checked connections belonging to the dead instance as invalid then removes these from the pool thereby ensuring these will not be dispensed.

b) Java Servlet using connections belonging to the dead instance will receive a SQLException; if the exception indicates a Recoverable Error i.e., invalid connection, then the application closes the connection and requests a new connection which will (transparently) come from one of the good standing instances (or disaster recovery sites with ADG or GDS).

Web application steps

Set UCP property `setFastConnectionFailoverEnabled` to `true` to enable FCF programmatically as shown below. By default FCF is disabled. Please make sure to have ons.jar in the classpath. Also, it is recommended that ojdbc7.jar, ucp.jar and ons.jar used should be from the same database version.

```
PoolDataSource pds = new PoolDataSourceFactory.getPoolDataSource();
String ONS_CONFIG = "nodes=racnode1:4200, racnode2:4200, racnode3:4200";
// not required with auto-ONS in 12c
pds.setONSConfiguration(ONS_CONFIG);

pds.setFastConnectionFailoverEnabled(true);
```

In addition to enabling FCF as a UCP property, **t**he following Java code snippet shows a typical FCF implementation. The `isValid()` method of the `oracle.ucp.jdbc.ValidConnection` interface used in conjunction with the FCF feature and is used to check if a borrowed connection is still usable after an SQL exception has been thrown due to an Oracle RAC down event.

```
try {
  connection = pds.getConnection();
  // do some work
} catch (SQLException ea) {
  // Fast Connection Failover
  if (connection == null || !((ValidConnection)connection).isValid(){
     ea.printStackTrace();
     String fcfInfo = ((OracleJDBCConnectionPoolStatistics)
                        pds.getStatistics()).getFCFProcessingInfoProcessedOnly();
     System.out.println("FCF information: " + fcfInfo);
   }
   // Close the connection
   conn.close();
}
```

FCF is primarily destined for unplanned outages but may also help during planned outage to failover applications which cannot relinquish connections (and have to be interrupted brutally).
Note: With FCF, in-flight transactions are lost (see Application Continuity hereafter for failing over in-flight work).

Logical Transaction ID (LTXID)

With Oracle Database 12c, each transaction is associated with a logical transaction ID (LTXID). The main and only purpose of LTXIDs is to help make a reliable determination of the outcome of the in-flight COMMIT statement. LTXIDs are assigned by the RDBMS at the beginning of each transaction and changed (i.e., incremented), only by the RDBMS when the corresponding transaction is either committed or rolled back. See the Transaction Guard section to find out how exactly LTXID is used.

Database Request (a.k.a. database unit of work)

In order to demarcate a replay-able unit of work, Oracle Database 12c introduces the notion of "*database request*".
UCP transparently demarcates the beginning of a database request when a connection is checked out, and the end of the request when the connection is checked back into the pool, as illustrated by the following code snippet.

```
Connection conn = pooldatasource.getConnection();
// Begin of Database Request
PreparedStatement pstmt = …
      …
 SQL, PL/SQL, local calls, RPC
      …
conn.commit();
// End of Database Request
conn.close();
```

With UCP, third party Java EE containers as well as custom Java frameworks get transparent demarcation of database requests. Without UCP, Java containers must explicitly demarcate database requests using **beginRequest** and **endRequest**[6].

Recoverable Errors

Oracle Database 12c classifies all SQL exceptions under two broad categories: (i) recoverable error (i.e., connection invalid upon RDBMS node/instance down) and (ii) unrecoverable (i.e., constraint violation). All error messages have an additional property named ***OracleException.IsRecoverable*.** JDBC throws a `SQLRecoverableException` however, Oracle JDBC 12c folds recoverable exception check under JDBC 4.0 `isValid()` method.
The following Java code snippet avoids applications the burden of maintaining their own list of error codes (e.g., ora-1033, `ora-1034, ora-xxx).`

```
try {  conn = pds.getConnection();
  ...
} catch (SQLException ea) {
    if (!((ValidConnection)conn).isValid()) {
      ea.printStackTrace();
      conn.close();
      System.out.println("Retry for a new conn" + ea.getMessage());
    }
}
```

Mutable Functions

Mutable functions are functions such as `SYSDATE, SYSTIMESTAMP, SEQUENCES` and `SYS_GUID` which result change on each invocation. In other words, replaying a unit of work containing a mutable function will not give the same result as the initial invocation. Your Web applications may or may not be sensitive to mutable functions during the replay of the same unit of work. Web application developers must assess the sensitivity of their applications and work with their Oracle DBA for granting the appropriate database permission to the user schema, as shown hereafter.

```
grant KEEP DATE TIME to user2;
grant KEEP SEQUENCE on sales.seq1 to user2;
```

6 See UCP ReplayableConnection http://docs.oracle.com/cd/E16655_01/appdev.121/e17663/oracle/jdbc/replay/ReplayableConnection.html

Transaction Guard for Java (TG)

Transaction Guard is an Oracle Database 12c feature exposed by all Oracle drivers for implementing "*at-most-once COMMIT execution*" during database outages (planned, unplanned). To summarize, each database transaction gets assigned a Logical Transaction Identifier (LTXID) as described above, which may be used by Web applications to make a reliable determination of the outcome of a COMMIT call failure thereby avoiding duplicate submission (i.e., buying/paying twice the same product or service). How would Java Servlets use Transaction Guard? How would DBAs configure the database service for TG?

Usage scenario, RDBMS and Web application interaction

At high-level, here is the interaction between the Oracle RDBMS and the Web application during Transaction Guard.

1) Transaction starts (typically upon connection check-out): the RDBMS associates a LTXID and sends it back to the driver.

2) The Web application performs its normal transaction (queries + DML) and issues a COMMIT statement

3) Typical use case: If a break in communication between the application and the RDBMS during the COMMIT statement execution occurs

   a) the RDBMS aborts the session subsequently, the associated connection dies

   b) UCP receives a FAN DOWN event from one of the good standing database instance

   c) The Java/Web application

      (i) receives a Recoverable Error (i.e., SQL exception)
      (ii) retrieves the LTXID of the in-flight transaction from the "dead connection" object
      (iii) requests a new connection
      (iv) uses the new connection and the LTXID to check the outcome of the COMMIT statement

   d) If the outcome is "`COMMITTED`" or "`ROLLBACKED`" then the application proceeds to the next transaction otherwise:
      (i) the RDBMS blocks the in-doubt transaction from eventually COMMITting
      (ii) the application may then resubmit the same transaction

DBA Steps[7]

The DBA must perform the following two tasks: configure the service to be used during TG and grant execute privilege to the user schema.

1) Configure the service for TG

```
$srvctl modify service -db <db_name> -service tgservice -commit_outcome true
$GDSCTL modify service -db <db_name> -service tgservice -commit_outcome true
```

2) Grant execute privilege to the user schema (e.g., SCOTT)

```
SQL> GRANT EXECUTE ON DBMS_APP_CONT TO SCOTT;
```

7 See more details in the TG paper @ http://www.oracle.com/technetwork/database/database-cloud/private/transaction-guard-wp-12c-1966209.pdf

Web Applications Steps

The Web application must: (i) define a PL/SQL procedure for retrieving the outcome of the transaction (i.e., the COMMIT call) associated with the LTXID; (ii) then use Java TG APIs

(1) Define a wrapper for PL/SQL dbms_app_cont.get_ltxid_outcome() procedure

```
private static final String GET_LTXID_OUTCOME_WRAPPER =
"DECLARE PROCEDURE GET_LTXID_OUTCOME_WRAPPER("+
"  ltxid IN RAW,is_committed OUT NUMBER ) IS " +
      "user_call_completed BOOLEAN; committed BOOLEAN; "+
      "BEGIN "+
      "  DBMS_APP_CONT.GET_LTXID_OUTCOME(ltxid, committed, user_call_completed); "+
      "  if committed then is_committed := 1;  else is_committed := 0; end if; "+
      "END; "+
      "BEGIN GET_LTXID_OUTCOME_WRAPPER(?,?); END;";
```

(ii) Define a boolean Java function for determining the output of the COMMIT call. Note that PL/SQL procedure GET_LTXID_OUTCOME returns another boolean USER_CALL_COMPLETED. Most Java applications do not need to check USER_CALL_COMPLETED.

```
boolean getTransactionOutcome(Connection conn, LogicalTransactionId ltxid) throws
SQLException {
  boolean committed = false;
  CallableStatement cstmt = null;
  try {
    cstmt = conn.prepareCall(GET_LTXID_OUTCOME_WRAPPER);
    cstmt.setObject(1, ltxid); // use this starting in 12.1.0.2
    cstmt.registerOutParameter(2, OracleTypes.BIT);
    cstmt.execute();
    committed = cstmt.getBoolean(2);
  }
  catch (SQLException sqlexc) {
    throw sqlexc;
  }
  finally {
    if(cstmt != null)
      cstmt.close();
  }
  return committed;
}
```

(iii) A code snippet showing how to use the TG for Java API[8] to handle errors coming from the driver, more specifically SQLRecoverableException type of exceptions.

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();
boolean isJobDone = false;
try (Connection connection = pds.getConnection()) {
  while (!isJobDone) {
    try {  // Auto-closable statement
        try (Statement statement = connection.createStatement()) {
          // your application logic goes here (DML and COMMIT)
          // Example: give 500$ raise to an employee
          statement.executeUpdate("update employees set
                     salary=salary+500 where employee_id=200;");
```

8 The complete TG for Java code sample will be available under https://github.com/oracle/jdbc-ucp

```
          connection.commit();
        }
        isJobDone = true;
        System.out.println("Job was successfully completed");
     } catch(SQLRecoverableException recoverableException) {
       // If there is a SQLRecoverableException, we can just retry the entire
       // transaction based on whether it is committed or not.

       // First step is recovery by closing the connection
       try { connection.close();
       } catch(Exception ex) {}

       // To use TG, get LogicalTranasactionId from the original connection
       LogicalTransactionId ltxid =
((OracleConnection)connection).getLogicalTransactionId();

       // Get a new connection
       Connection newConnection = pds.getConnection();

       // Check if the transaction is committed or not
       isJobDone = getTransactionOutcome(newConnection, ltxid);
       // If committed, we don't need to retry the transaction, else,
       // it gets into while loop for a retry
       if(isJobDone) {
         System.out.println("No retry required, TX is already committed.");
       }
       // Transaction was not commited, so retry
       connection = newConnection;
     }
   }
}
```

Application Continuity for Java (AC)

Application Continuity (AC) for Java is an out of the box solution which straddles Oracle Database 12c and UCP. AC is a runtime collaboration between the RDBMS and UCP/JDBC, using the following building blocks: database request (or unit of work), Recoverable Error, the new JDBC replay data source, Transaction Guard for Java, and database High Availability configurations (RAC, Data Guard).

To summarize and assuming DBAs and Web applications designers have taken the proper steps outlined hereafter: during a database request, AC records every interaction with the RDBMS and replays it upon outages. When successful, AC masks hardware, software, network, and storage failure to applications during planned and unplanned outages. End-users only experience a slight delay in response time. When not successful, the original error is re-thrown by the driver, applications must still provision code to deal with such errors but such error handling code will be rarely used..

What are the DBA and Web applications steps or configurations?

Web Application Steps

For AC to be successful, Web applications designers need to perform the following steps:

a) [Optional] implement and register a UCP callback for setting custom connection states before replaying the unit of work

b) Specify the Replay Data source (`oracle.jdbc.replay.OracleDataSourceImpl`)

   Example: In Apache Tomcat, `ConnectionFactoryClassName` is defined in “context.xml” to use replay data source.

```
<Resource
name="tomcat/UCPPool"
auth="Container"
factory="oracle.ucp.jdbc.PoolDataSourceImpl"
type="oracle.ucp.jdbc.PoolDataSource"
description="UCP Pool in Tomcat"
connectionFactoryClassName="oracle.jdbc.replay.OracleDataSourceImpl"
… />
```

In addition, Web application designers must make the following assessments:

a) Explicit demarcation of "*database request*" boundaries; not needed with UCP.

b) Side effects: replaying applications sections may incur side effects. As an example, replaying the section of your Web applications which makes external calls for auditing, mailing, printing checks and so on, might not be desirable. To avoid such side effects, you may disable "*replay*", using the `disableReplay()` method of the `ReplayableConnection` interface.

c) Are your Web applications sensitive or not to "mutable functions" during replay? If so, work with the DBA o grant specific privileges to the schema in question as discussed in the Mutable Functions section above.

DBA steps

The default database service is not recommended and suitable. DBA must configure a service with the following properties

FAILOVER_TYPE='TRANSACTION' -- For Application Continuity
REPLAY_INITIATION_TIMEOUT=1800 -- After which replay is cancelled
FAILOVER_DELAY=10; -- Delay in seconds between connection retries
FAILOVER_RETRIES=30 -- Number of connection retries per replay
commit_outcome=true
aq_ha_notifications=true

```
$srvctl modify service -db <db_name> -service <service_name>  -failovertype
TRANSACTION  -replay_init_time 600 -failoverretry 30 -failoverdelay 10
-commit_outcome TRUE
```

Connection Time Load Balancing

A feature of Oracle Net Services that balances incoming connections requests across all instances providing the same database service. Connection time load balancing improves scalability by balancing the number of active connections among multiple instances.

Refer to the sample URL hereafter for a RAC/GDS environment

```
url = "jdbc:oracle:thin:@(DESCRIPTION_LIST=
   (LOAD_BALANCE=off) - Load balance between data centers  -- should be turned OFF
   (FAILOVER=on) - Failover between data centers -- should be turned ON
   (DESCRIPTION=(CONNECT_TIMEOUT=90) (RETRY_COUNT=10)(RETRY_DELAY=3)
     (ADDRESS_LIST=
       (LOAD_BALANCE=on) - Connection time load balancing
       (ADDRESS=(PROTOCOL=TCP)(HOST=NY-scan)(PORT=1521)))
     (CONNECT_DATA=(SERVICE_NAME=gold)))
   (DESCRIPTION=(CONNECT_TIMEOUT=90) (RETRY_COUNT=10)(RETRY_DELAY=10)
     (ADDRESS_LIST=
       (LOAD_BALANCE=on)  - Connection time load balancing
       (ADDRESS=(PROTOCOL=TCP)(HOST= UK-scan)(PORT=1521)))
     (CONNECT_DATA=(SERVICE_NAME=gold))))
```

Run Time Load Balancing

Run time load balancing is the ability of Oracle Database clients frameworks (i.e., drivers and connection pools) to allocate connections based on the current workload of the database instance. Runtime Load Balancing comes also into play when new node(s)/instance(s) are added/removed to/from the service; the work load gets balanced in both situations without manual intervention.

When enabled, RAC and GDS post runtime load balancing advisories every 30 seconds. Load balancing advisories furnish workload metrics which are used to distribute connection requests across all available Oracle RAC instances. UCP uses the Oracle RAC load balancing advisories as hints to dispense new connections from the least loaded instances thereby achieving best scalability. Load balancing advisories require subscribing to FAN events, which is achieved by turning on FCF. Without the load balancing advisories, UCP will not receive workload metrics and connections will be checked out randomly.

There are two types of service-level goals for run-time load balancing:

SERVICE_TIME: Attempts to direct work requests to instances according to response time. Load balancing advisory data is based on elapsed time for work done in the service plus available bandwidth to the service. An example for the use of SERVICE_TIME is for workloads such as internet shopping where the rate of demand changes. The following example shows how to set the goal to SERVICE_TIME at the service level:

```
$ srvctl modify service -db <db_name> -service <service_name> -rlbgoal SERVICE_TIME
-clbgoal SHORT
```

THROUGHPUT: Attempts to direct work requests according to throughput. The load balancing advisory is based on the rate that work is completed in the service plus available bandwidth to the service. An example for the use of THROUGHPUT is for workloads such as batch processes, where the next job starts when the last job completes. The following example shows how to set the goal to THROUGHPUT at the service level:

```
$ srvctl modify service -db <db_name>  -service <service_name> -rlbgoal THROUGHPUT -clbgoal LONG
```

Conclusion

This paper explains Oracle Database high availability and scalability concepts as well as the configurations or application changes required to enable these features. These features may be used either in isolation with Java frameworks or through Java containers such as Weblogic, WebSphere, Tomcat and others -- which expose these capabilities -- thereby allowing Web applications to sustain planned and unplanned outages.

The features described in this paper are valid for all Oracle Database 12c high availability and scalability configurations including RAC, Active Data Guard and Global Data Services.