ORACLE
DATABASE

# Application Continuity with Oracle Database12c Release 2

ORACLE WHITE PAPER | OCTOBER 2019

## Executive Overview

It is complex for application developers to mask outages of a database session (instance, node, storage or network or any other related component) and as a result errors and timeouts are often exposed to the end users leading to user frustration, lost productivity, and lost opportunities. Application Continuity masks outages from end users and applications by recovering the in-flight work for impacted database sessions following outages. Application Continuity performs this recovery beneath the application so that the outage appears to the application as a slightly delayed execution.

Application Continuity is invoked for outages that result in recoverable errors, typically related to underlying software, foreground, hardware, communications, network, or storage layers. Application Continuity is used to improve the user experience when handling both unplanned outages and planned maintenance. Application Continuity strengthens the fault tolerance of systems and applications that use an Oracle database.

With Oracle Database 12c Release 2, Application Continuity is available for applications using Java, OCI and ODP.NET unmanaged provider

- Oracle WebLogic Server for non-XA and XA data sources
- Oracle Universal Connection Pool, used standalone or as a data source for third party Application Servers including IBM WebSphere and Apache Tomcat
- JDBC applications using the JDBC `PooledConnection` interface
- Oracle JDBC-Thin Replay Driver
- Oracle Tuxedo for non-XA data sources
- Oracle OCI Session Pool
- Oracle ODP.NET unmanaged provider
- Oracle SQL*Plus

"What application continuity brings to applications now is that they can run in a clustered environment with the security knowing that the application continuity capabilities in Oracle Database 12*c* are going to handle a lot of failure scenarios automatically."

— Marc Fielding, ATCG Principal Consultant Oracle, Pythian

## Introduction

Application Continuity enables replay, in a non-disruptive and rapid manner, of a database request when a recoverable error makes the database session unavailable. The request can contain transactional and non-transactional calls to the database and calls that are executed locally at the client or middle tier. After a successful replay, the application can continue where that database session left off. Users are no longer left in doubt, not knowing what happened to their funds transfers, flight bookings, purchases and so on. Administrators no longer need to re-boot mid-tier machines and recover from logon storms caused by failed sessions. With Application Continuity, the end user experience is improved by masking many outages, planned and unplanned, without requiring the application developer to attempt to recover the request.

Without Application Continuity, it can be almost impossible for an application to mask outages in a safe and performing way, for reasons that include the following:

- The state at the client remains at present time, with entered data, returned data, and variables cached while the state changes reflected in the database session are lost.
- If a commit has been issued, the commit message is not durable. Furthermore, checking the status of a lost request is no guarantee that it will not commit in the future.
- Non-transactional database session states that the application needs to operate are lost.
- If the request can continue, the database and the database session must be in sync.

Don't be misled by naïve approaches that attempt to resubmit using a single key leading to hangs, using the wrong database out of sync, downtime managing indexes, and inability to scale at runtime.

Application Continuity does the work for the application developer. Application Continuity improves developer productivity by attempting to mask database-related outages that can be safely masked.

Applications should continue to include error handling for these cases:

- Non-recoverable errors, such as invalid input data. Application Continuity applies to recoverable errors.
- Recoverable errors when replay encounters one of the listed restrictions, such as usage of `oracle.sql` concrete classes in the application, or when replay is not able to restore the original user-visible results on which the application may have made decisions so far.

## Experience before Application Continuity

Without Application Continuity, database recovery does not mask outages that are caused by network outages, instance failures, hardware failures, network failures, repairs, configuration changes, patches, and so on.

### Earlier Experience for Unplanned Outages

Figure 1 illustrates the earlier experience where errors can be returned to the end user, even when the request completed.

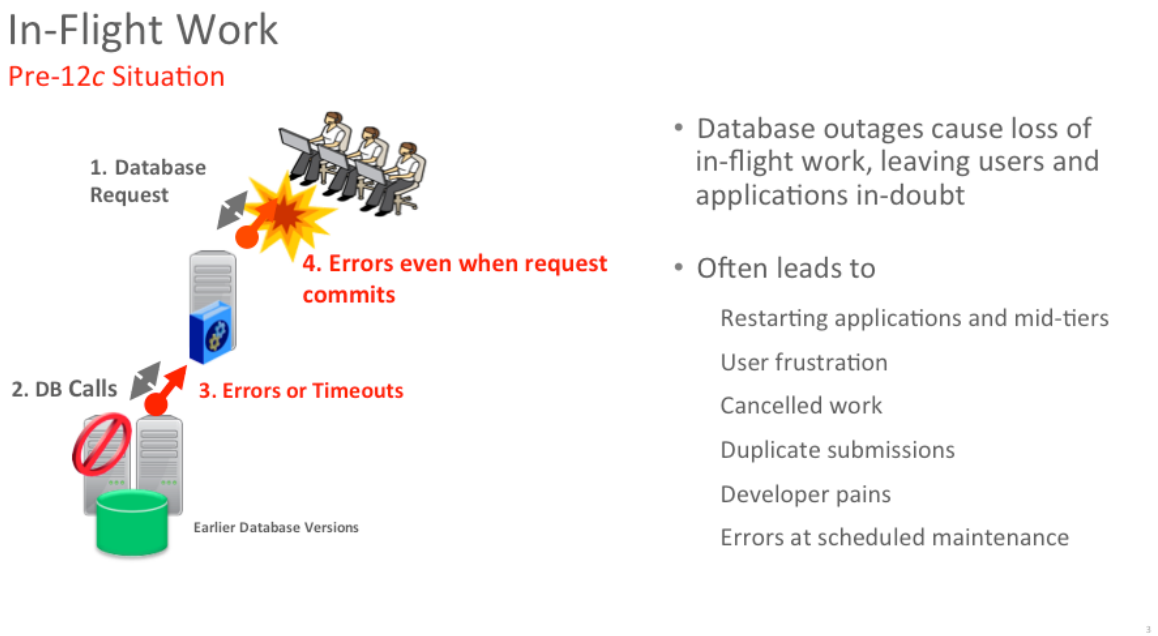


Figure 1 Earlier experience for unplanned outages

For end users database session outages can affect them by:

- Confusion: Users do not know what happened to their application's requests such as funds transfers, orders, payments, bookings, deposits and so on.
- Decreased Usability: Users may see an error, lose uncommitted data, and need to log in again and re-enter or resubmit their data potentially causing duplicate transactions, more frustration and support costs.
- Disruption: DBAs sometimes need to reboot mid-tier machines to handle the load balance and incoming logon storm and again to rebalance on repair. These are situations that should never occur.

### Earlier Experience for Planned Maintenance

Figure 2 illustrates the earlier experience for planned maintenance whereby errors would be returned to the users when a cluster node or instance was stopped to allow the maintenance to continue.

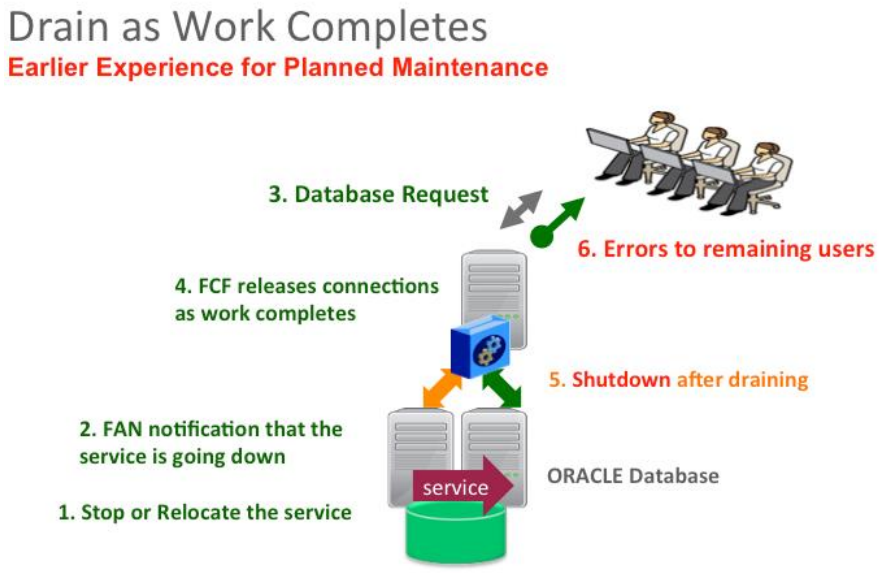


Figure 2. Earlier experience for planned maintenance

**Planned maintenance is more frequent than unplanned outages.**

For applications using RAC, RAC One, Oracle Restart, Oracle Data Guard, Oracle Active Data Guard, and Oracle Guard with FAN in conjunction with Oracle connection pools and drivers – Oracle WebLogic Server Connection Pool, Oracle Universal Connection Pool, ODP.NET managed and unmanaged providers, OCI Session Pool and Tuxedo, and in 12cR2 the JDBC driver – planned maintenance has been masked since Oracle Database 10g. If you have a third party application server, consider using the Oracle Universal Connection Pool as the data source, for example with IBM WebSphere or Apache Tomcat, and also for your standalone Java application.

In 12cR2, Fast Connection Failover (FCF) is on by default for most clients when using the recommended Net connection string. Fast Application Notification (FAN) is pre-configured at the server at install and upgrade for Real Application Clusters

Use SRVCTL or GDSCTL to relocate the service from an instance or, if you are using a UNIFORM service pr all preferred, then stop the service on an instance. From 12$^{c}$R2, set drain-timeout to allow time for the sessions to drain, and stop_option to abort immediately after draining.

FAN sends a notification that the service is down for planned reasons.

On receipt of the FAN event by the connection pool or driver, idle connections to the down service are removed, and no further connections to that instance are allowed. For those applications that return their connections to the pool or in 12cR2 check their Java connections, Fast Connection Failover automatically terminates the connection at a safe place. The application and users see no errors. Existing connections on other instances remain usable, and new connections are opened to these instances as needed. With the drain_timeout service setting in 12cR2, the draining is graceful so as not to disrupt work already at the target instances.

All applications should return their connections to the pool or check their connections, or both, but in reality this does not always happen. Those that do not can encounter errors when the instance is stopped, leading to a poor user experience.

## Oracle Database 12c Experience using Application Continuity

Figure 3 illustrates the improved user experience possible with Oracle Database 12c for applications using Application Continuity.

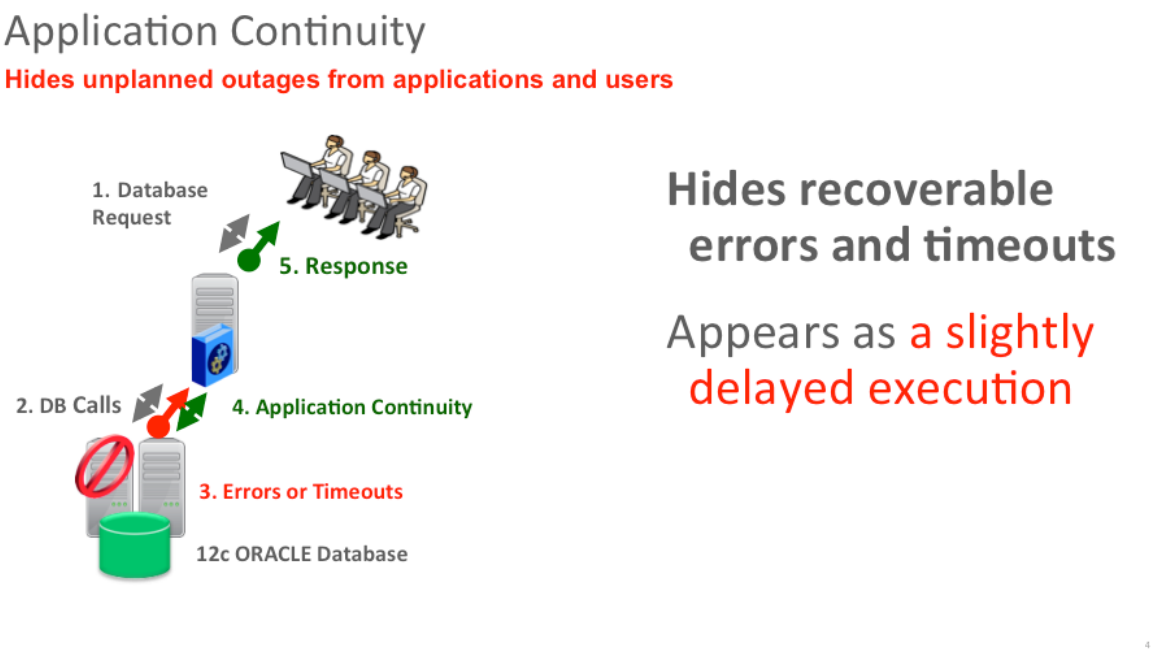


Figure 3. User experience when using Application Continuity

When replay is successful, Application Continuity masks many recoverable database outages from the applications and the users. It achieves the masking by restoring the database session, the full session (including session states, cursors, variables), and the last in-flight transaction (if there is one).

If the database session becomes unavailable due to a recoverable error, Application Continuity attempts to rebuild the session and restore any open transactions to the correct states.

If the transaction is successfully committed, the successful status is returned to the application.

If the replay is successful, the request continues safely and transparently, with no risk of duplication.

If the replay is not successful, the database rejects the replay and the application receives the original error. To be successful, the replay must return to the client the exact same data that the client received previously in the request, which the application potentially made a decision on.

# Application Continuity with Oracle Database 12c

## Application Continuity Coverage

Application Continuity for Oracle Database 12c supports the following client and server features:

### Oracle Database 12c Client

- Oracle JDBC Replay Driver 12c or later. This is a JDBC driver feature provided with Oracle Database 12c for Application Continuity, referred to as the "replay driver" onwards.
- Oracle Universal Connection Pool (UCP)
- Oracle WebLogic Server 12c, and third-party
- Java connection pools or standalone Java applications using Oracle JDBC - Replay Driver 12c or later.
- Oracle Tuxedo (version?)
- OCI Session Pool 12c Release 2 or later
- SQL*Plus 12c release 2 or later
- ODP.NET Unmanaged Provider 12c Release 2 or later

If using a third party, Java-based application server, there are three ways to obtain Application Continuity functionality out of the box:

- The most effective method is to replace the data source with UCP. This approach is supported by many application servers including IBM WebSphere and Apache Tomcat. Using UCP as the data source allows UCP features such as Fast Connection Failover, Runtime Load Balancing and Application Continuity to be used with full certification.
- Starting with Oracle JDBC 12102, request boundaries are embedded in the JDBC `PooledConnection` interface. This makes Application Continuity available to Third-party Java connection pools and standalone Java applications that use this interface in their native pools.
- Applications and third parties may also add request boundaries. Database request identification is demarcated at the database using JDBC APIs `beginRequest` and `endRequest`.

### Oracle Database 12c Server

Application Continuity is driven by the database. The database decides which calls are replayable, how replay should be done, builds validation, saves mutable values, and sends instructions to the clients on how to handle capture and replay.

All common database calls are replayed - SELECT, PL/SQL, ALTER SESSION, DML, DDL, COMMIT, ROLLBACK, SAVEPOINT, JDBC and OCI RPCs, and local JDBC and OCI calls.

All common database transaction types are replayed - local, parallel, remote, distributed, and transactions embedded within PL/SQL.

Mutable Oracle functions are kept for replay including sequences, date/time and GUIDs. see Appendix - New Database Concepts for Application Continuity).

Hardware acceleration is in embedded using Software in Silicon for hardware using Intel & Sparc chips.

## Processing Phases in Application Continuity

There are three distinct processing phases used by Application Continuity, shown in table 1. This processing occurs beneath the application in the driver and database, and is transparent to the application.

**TABLE 1. PROCESSING PHASES OF APPLICATION CONTINUITY**

| NORMAL RUNTIME | RECONNECT | REPLAY |
|---|---|---|
| • Tags database requests<br>• Server and client decide what is replayable and what is not<br>• Under instruction, client holds original calls with validation | • Ensures request has replay enabled<br>• Verifies timeliness<br>• Creates a new connection<br>• Validates target database<br>• Uses Transaction Guard to enforce last commit outcome | • Replays held calls<br>• During replay, ensures that user visible results match original<br>• Continues the request if replay is successful<br>• Throws the original exception if replay is unsuccessful |

**Normal Runtime -** At normal runtime, each database request is tagged with a request begin and end marker, either by checking out of or back into the WebLogic Server Connection Pool, or Universal Connection Pool, OCI Session Pool, Tuxedo, or ODP.NET, by using SQL*PLUS, or by adding begin and end request tags to your own application or to your own Java connection pool at connection check-out and check-in. The Universal Connection Pool is supported standalone and with many 3rd party Application Servers including IBM WebSphere and Apache Tomcat.

In collaboration between the Oracle 12c client driver and the Oracle Database 12c, it is decided which calls in a request are replayable. Replayable calls are held longer by the 12c driver, together with validation received from the database. The driver holds these calls and validation information until the end of the database request or until replay is disabled. Replay can be disabled by a restricted call, a commit (in the default mode (see special modes), the request ending, or explicitly by the application.

**Reconnect -** The reconnect phase of Application Continuity is triggered by the driver when a recoverable error occurs. In this phase, the request is checked to see if replay is still enabled, and the timeout after which replay is not permitted (replay initiation timeout) is checked to ensure that it has not expired. Assuming these checks pass, a new connection to the database is established. The reconnection to the database can take time if the service needs to be re-established, so the DBA should check the NET configuration values for REPLAY_COUNT and REPLAY_DELAY, and attributes of the service, FAILOVER_DELAY and FAILOVER_TIMEOUT. The NET settings are essential as these are used for both failing over and for new incoming connections.

After the driver has established a connection to the database, the driver checks if the connection is to a valid database target, and whether or not the last transaction (if there was one) committed successfully. Replay will not occur if the connection is to a logically different database or is to the same database but that database has lost transactions. For example, the database has been restored back in time. The driver will not resubmit committed transactions. At-most-once submission is enforced using Transaction Guard.

**Replay -** The replay phase starts once a new connection to the database is established. All calls held by the driver are replayed under the direction of the database. Replay is disabled if there are any user visible changes in

results observed during the replay as determined by the validation. Replay does not allow commit during the replay phase but does allow the last call to commit. That is, the call that encountered the recoverable error. Following a successful replay, the request continues from the point where it had failed. To the application is appears as a slightly longer execution.

## Restrictions using Application Continuity

The following restrictions apply to using Application Continuity at three levels global, local, and database.

**TABLE 2. RESTRICTIONS FOR APPLICATION CONTINUITY**

| GLOBAL | REQUEST | TARGET DATABASE |
|---|---|---|
| Does not support :<br>• Default database service and default PDB service<br>• Deprecated JDBC concrete classes<br>• 3rd party statement cache | • For Java streams, replay is on a "best effort" basis<br>• Active Data Guard with app-level database links<br>• Two phase XA<br>• Request-level disable for<br>— Alter System<br>— Alter Database<br>— Some alter session | Does not support:<br>• Logical Standby<br>• Golden Gate<br>• 3rd Party Replication |

**Global** - This restriction prevents Application Continuity from being enabled or used on any request.

Replay is not supported for connections using the database service, i.e. the default service corresponding to the DB_NAME or DB_UNIQUE_NAME. The database service is not intended for use by high availability applications because this service cannot be enabled, disabled, or failed over.

For applications using Oracle JDBC Driver, there is no support for deprecated oracle.sql concrete classes like BLOB, CLOB, BFILE, OPAQUE, ARRAY, STRUCT or ORADATA. (See MOS note 1364193.1). Use ORAchk acchk "Clean Up Concrete Classes for Application Continuity" to know if an application passes.

For OCI and ODP.NET, in Oracle Database 12c release 2 (12.2.0.1), Application Continuity on OCI driver excludes ADTs, advance queues, dynamic binds, and some LOB APIs.

If a statement cache at the application server level is enabled (for example, a third-party application server statement cache), this must be disabled when the replay is used. Instead, configure the JDBC or OCI statement cache which support Application Continuity, and perform well because they are optimized for JDBC and Oracle. E.g. Use `oracle.jdbc.implicitstatementcachesize=nnn`.

**Request** - This restriction disables Application Continuity for part of a database request. Replay is enabled automatically on the next request.

For JDBC stream arguments, replay is on a "best effort" basis. For example, if the application is using physical addresses, the address is no longer valid with the outage and cannot be repositioned.

For AC for OCI, 12.2.0.1 disables for the remainder of the request on seeing AQ, ADT's and some lob operations.

Starting with Oracle Database 12 Release 2 (12.2), replay is supported for the XA data source for Java and ODP.NET Unmanaged Driver. Replay supports local transactions only. Replay silently disables when two-phase is used and re-enables automatically on the next request. This allows Application Continuity to work very well with promotable XA and with applications using the XA data source and mostly not using two-phase commit.

Replay is not supported if you are using Active Data Guard with database links to another database at the application level. Database links are supported for Data Guard itself and for AWR using a different service from your application.

Replay is disabled if the request executes an ALTER SYSTEM or ALTER DATABASE statement. For example, replay will not add tablespaces or startup or shutdown the database. Replay is also disabled for some ALTER SESSION statements such as those that change commit behavior or set events.

**Target Database** – Currently, Application Continuity does not support failover to logically different databases – including Oracle Logical Standby, Oracle Golden Gate and third party replication solutions. Replay has a strict requirement that it applies to the same databases with verified no transaction loss and no divergence in ancestry. The database at each end point should have a different DBID. Use V$DATABASE to obtain the DBID for each database.

### When is a request not covered for replay

The following events can disable replay for a database request. If disabled, replay is automatically re-enabled at the beginning of the next request. Disabling replay does not impact normal runtime or other requests. Validation and recording simply stop and held calls are released by the driver. That request is not protected for replay once disabled.

**TABLE 3. WHEN IS AN APPLICATION NOT COVERED BY REPLAY**

| NORMAL RUNTIME | RECONNECT | REPLAY |
|---|---|---|
| Any calls in same request after –<br>• successful commit in dynamic mode (the default)<br>• a restricted call<br>• Replay Disable API<br>• Promote to XA | • Error is not recoverable<br>• Timeouts<br>— Replay initiation timeout<br>— Max connection retries<br>— Max retries per incident<br>• Target database is not valid for replay<br>• Last call committed in dynamic mode | • Validation detects different results |

**Normal Runtime** - Call capture is enabled at the `begin Request` tag. Capture happens on all original calls until a successful COMMIT (in dynamic mode), unless a restricted call is encountered, or the application makes an explicit `disableReplay` call for Java or OCIRequestDisableReplay for OCI. Capture otherwise continues until the `endRequest` tag is encountered.

**Reconnect** - Replay does not occur if the error is not recoverable, if timeouts are exceeded (see Configure Services for Application Continuity), if the target database is not the same or an ancestor of the original database, or if the request committed and Application Continuity is using dynamic mode.

**Replay** - is aborted and the original error is returned if the validation for the replayed call does not pass. The replay must return the same user visible results that the application has seen and potentially made a decision on.

### Potential Side-Effects

Autonomous transactions and external PL/SQL and server-side Java callouts create side effects that are separate from the main database request. Some of the database calls that create side effects include email and notifications using DBMS_ALERT calls, copying files using DBMS_PIPE and RPC calls, writing text files using UTL_FILE calls, making HTTP callouts using UTL_HTTP calls, sending email using UTL_MAIL calls, sending SMTP messages using UTL_SMTP calls, sending TCP messages using UTL_TCP calls and accessing URLs using UTL_URL calls.

Calls with side-effects can leave persistent results behind. For example, if a user walks away part way through some work without committing and the session times out or the user issues Ctrl+C, the foreground or a component fails; the main transaction rolls back while the side effects may have been applied.

Side effects are replayed unless the application specifies otherwise. Applications that use external actions should be reviewed to decide if requests with side effects make sense to replay or not. For example, does the application want to send email again or to audit again or to transfer a file again? Frequently it is desirable to replay the side effect. However, sometimes it may be better not to. If a request has an external action that should not be replayed, that request can use a connection that does not have Application Continuity enabled, or replay can be disabled for that request using the `disableReplay` API for Java or OCIRequestDisableReplay for OCI. All other requests continue to be replayed.

## Assessment Steps to Use Application Continuity

### Assessment steps for your application

Before enabling Application Continuity for an application, complete the assessment steps shown in table 4.

**TABLE 4. ASSESSMENT STEPS**

| DECIDE | WHAT TO DO |
|---|---|
| Request Boundaries | Return connections to the Oracle pool between requests. If using 3rd party Java connection pools, use UCP, or *PooledConnection*, or add request boundaries |
| JDBC Concrete Classes | Replace deprecated JDBC concrete classes with Java interfaces. Use OraChk acchk to know if the application passes, and if not, where. |
| Decide to Disable | Use a different connection or a disable replay API if a request has any request that should not be replayed. |
| Initial State | Set FAILOVER_RESTORE=LEVEL1 if the application sets initial states.<br>Register a callback if using initial states not restored by FAILOVER_RESTORE<br>Do nothing if using WebLogic Server Active GridLink or UCP labeling. |
| Grant Mutable Functions | Grant keeping original mutable values |
| Measure protection level | Use OraChk acchk to report the level of protection using AC, and for any requests not protected, which and also why. |

## Check Request Boundaries

Request boundaries are tags that mark the beginning and end of a database request. As of Oracle 12c Release 2 Clients, connection pools that embed boundaries include Oracle Universal Connection Pool, all WebLogic Server data sources, Tuxedo, OCI, ODP.NET Unmanaged Provider and standard 3rd party Application Servers and standalone Java pools that use the Oracle 12 JDBC drivers' `PooledConnection` interface, and also SQL*PLUS.

Follow these steps to decide if request boundaries are in place.

1. First determine whether the application borrows and returns connections from one of the pools listed. If it does, and is returning the connection between requests, request boundaries are embedded.

   (For SQL*Plus use the –ac switch.)

2. If the application is using one of these pools, and is not releasing connections, there is often an application property to set to release the connections. Releasing connections scales much better than not releasing connections, and marks the request boundaries with no other change. Make this configuration change and no other change is needed for request boundaries.

3. If the application uses a third party Java connection pool that does not use the Oracle JDBC `PooledConnection` interface, there are three choices:

   - Replace the data source with UCP. See the white papers for this easy data source replacement on OTN under Oracle JDBC for the various 3rd party application servers. Using this method, allows the 3rd party application server to use all UCP features including FAN, Runtime Load Balancing, XA affinity, and Application Continuity. This method is a highly recommended solution for IBM WebSphere, Apache TomCat and RedHat Spring.
   - Use the Oracle JDBC `PooledConnection` interface for creating and managing the connection pool.
   - For Java applications, add `beginRequest` and `endRequest` APIs to identify database request boundaries. These API's have no performance cost and is isolated to check-out and check-in for a custom JDBC pool. They must be added at borrow and return. Placing the API's anywhere other than start and end of request results in partial requests and is not supported. It is always better to use one of the other methods.

## Ensure No Deprecated Oracle JDBC Concrete Classes

For Java applications only, determine whether the application uses deprecated Oracle JDBC concrete classes.

To use Application Continuity with Java, deprecated Oracle JDBC concrete classes must be replaced. For information about the deprecation of concrete classes, including actions to take if an application uses them, see My Oracle Support Note 1364193.1 https://support.oracle.com/CSP/main/article?cmd=show&type=NOT&id=1364193.1

To know if the application is using concrete classes, use AC checking (called acchk in ORAchk ). There is no need to use 12c driver or database or to have source code for this check for concrete classes. The application can be verified in advance while planning for high availability for your application.

There are three values that control the AC checking for concrete classes. They can be set either on the command line or via shell environment variables (or mixed). They are the following.

**TABLE 5. USING AC CHECKING FOR CONCRETE CLASSES**

| Command Line Argument | Shell Environment Variable | Usage |
|---|---|---|
| –asmhome jarfilename | RAT_AC_ASMJAR | This must point to a version of asm-all-5.0.3.jar or higher that you download from http://asm.ow2.org/. |
| -javahome JDK8dirname | RAT_JAVA_HOME | This must point to the JAVA_HOME directory for a JDK8 installation. |
| -appjar dirname | RAT_AC_JARDIR | To analyze the application code for references to Oracle concrete classes, this must point to the parent directory name for the code. The program analyzes .class, .jar, .ear, .war, and .rar files through the directory tree recursively. |

Example usage:

$ ./orachk -asmhome /tmp/asm-all-5.0.3.jar -javahome /tmp/jdk1.8.0_40 -appjar /tmp/appdir

Example output:

| Outage Type | Status | Message |
|---|---|---|
| Concrete class checks | | **Total : 19845 Passed : 19610 Warning : 0 Failed : 235 (Failed check count is one per file)** |
| | FAILED | [oracle/jdbc/driver/ArrayDataResultSet][[CAST] desc= oracle/sql/STRUCT method name=getObject, lineno=1477] |
| | FAILED | [oracle/jdbc/driver/ArrayDataResultSet][[CAST] desc= oracle/sql/NCLOB method name=getNClob, lineno=1776] |
| | FAILED | [oracle/jdbc/driver/BfileAccessor][[Method] name=getBFILE, desc=(I)Loracle/sql/BFILE;, lineno=99] |
| | FAILED | [oracle/jdbc/driver/BlobAccessor][[Method] name=getBLOB, desc=(I)Loracle/sql/BLOB;, lineno=129] |
| | FAILED | [oracle/jdbc/driver/ClobAccessor][[Method] name=getCLOB_, desc=(I[B)Loracle/sql/CLOB;, lineno=230] |
| | FAILED | [oracle/jdbc/driver/ClobAccessor][[Method] name=getCLOB, desc=(I)Loracle/sql/CLOB;, lineno=226] |
| | FAILED | [oracle/jdbc/driver/ClobAccessor][[Method] name=getNCLOB, desc=(I)Loracle/sql/NCLOB;, lineno=390] |
| | FAILED | [oracle/jdbc/driver/ClobAccessor][[CAST] desc= oracle/sql/NCLOB method name=getNCLOB, lineno=398] |
| | FAILED | [oracle/jdbc/driver/ClosedConnection][[Method] name=createTemporaryBlob, desc=(Ljava/sql/Connection;ZI)Loracle/sql/BLOB;, lineno=974] |

For more examples using AC Checking for concrete classes, see:

https://blogs.oracle.com/WebLogicServer/entry/using_orachk_to_clean_up

### Decide if any requests should not be replayed

Before deploying decide whether the application has any requests that should not be replayed. If a request makes an external call using one of the external PL/SQL messaging actions or via server-side OJVM, or uses autonomous transactions, check this request. Decide if it should be replayed or not, and disable if it should or should not be replayed. Whenever a screen is reloaded or a request is resubmitted actions such as sending email or auditing are repeated. Many applications want this. But there are cases when reloading or replaying should not be permitted for a request. (See Potential Side Effects replaying in 12cR2 documentation.)

If a function in the application should not be replayed, use a connection to a service without Application Continuity for this function. Alternatively, embed a disable replay API (`disableReplay for Java or OCIRequestDisableReplay for OCI`) within a request that should not be replayed. All other requests are replayed.

Disabling is especially important for requests that use the UTL_HTTP package or autonomous procedures that enforce serialization, or that rely on mid-tier wall clock time. Replay recovers sessions concurrently and independently. Disable replay if the application logic assumes that otherwise independent database sessions are synchronized. For example, do not use replay if the application synchronizes sessions using volatile resources such as user locks or external devices that are held until commit, rollback, or session loss.

Disable replay if the request relies on time at the mid-tier in the execution logic. If a request assumes that a statement executed at Time T1 is not re-executed at Time T2 then disable replay for this request. The replay driver does not repeat the mid-tier logic. It repeats the database calls that execute as part of this logic.

### Decide if the application needs initial state re-established before failover

Many applications are stateless and the request manages its own state. Stateless applications do not expect state on a connection when the connection is borrowed from a pool. These applications may use state in requests. They do not expect that state when the connection is next borrowed. Choose -

- FAILOVER_RESTORE=NONE set on the service. (This is the default.)

Some applications and mid-tiers color connection pools so all connections are for example in a preset language or time zone. If session state is set intentionally on connections outside requests, and requests expect this state, replay needs to re-create this state before replaying. Choose one of the following options

- Universal Connection Pool or WebLogic Server Connection Labeling
- FAILOVER_RESTORE=LEVEL1 set on the service
- Connection Initialization Callback for Java or TAF Callback for OCI

When using Oracle WebLogic Server or the Universal Connection Pool, connection labeling is suggested. If using connection labeling, these labels are used automatically for Application Continuity. No change is needed.

In all other cases, most common states are restored automatically by setting FAILOVER_RESTORE to LEVEL1 on the service. See Table 6.

**TABLE 6. STATES RESTORED BY FAILOVER_RESTORE=LEVEL1**

| | | |
|---|---|---|
| NLS_CALENDAR | NLS_NUMERIC_CHARACTER | MODULE |
| NLS_CURRENCY | NLS_SORT | ACTION |
| NLS_DATE_FORMAT | NLS_TERRITORY | CLIENT_ID |
| NLS_DATE_LANGUAGE | NLS_TIME_FORMAT | ECONTEXT_ID |
| NLS_DUAL_CURRENCY | NLS_TIME_TZ_FORMAT | ECONTEXT_SEQ |
| NLS_ISO_CURRENCY | NLS_TIMESTAMP_FORMAT | DB_OP |
| NLS_LANGUAGE | NLS_TIMESTAMP_TZ_FORMAT | AUTOCOMMIT (Java and SQLPLUS) |
| NLS_LENGTH_SEMANTICS | TIME_ZONE (OCI, ODP.NET 12201) | CONTAINER and SERVICE (OCI, ODP.NET 12201) |
| NLS_NCHAR_CONV_EXCP | CURRENT_SCHEMA | |

If initial states are needed and they are not included with FAILOVER_RESTORE or labels, a callback is needed. Register this at the WebLogic Administration Console, at Universal Connection Pool, or at the JDBC replay driver, or for OCI and ODP.NET using the TAF callback. Application Continuity will re-execute the callback at replay. Use a callback only when the application needs state that is not established in the request, the application has not implemented connection labeling, and the state cannot be restored using FAILOVER_RESTORE set to LEVEL1.

### Grant support for keeping mutable values

Mutable functions are functions that can return a new value each time they are executed. Support for keeping the original results of mutable functions is provided for SYSDATE, SYSTIMESTAMP, SYS_GUID, and sequence.NEXTVAL. If the original values are not kept and if different values are returned to the application at replay, replay is rejected.

Decide whether keeping original mutable values is compatible with the application and if so keep mutable values for replay. To help decide, think of Application Continuity as a slightly delayed execution. If your application receives a date/time or sequence, for example, and then the system snoozes for a few seconds and comes back, the application has these values and continues. Application Continuity is similar. The application has the mutable already so at replay the server uses the same and continues, as if an outage had not occurred, just a snooze.

Configure mutable objects using GRANT KEEP for application users, and the KEEP clause for a sequence owner. When KEEP privilege is granted, replay applies the original function result at replay.

For example

```
SQL> GRANT [KEEP DATE TIME | KEEP SYSGUID].. [to  USER]

SQL> GRANT KEEP SEQUENCE.. [to  USER] on [sequence object];
```

SQL> ALTER SEQUENCE.. [sequence object]  [KEEP|NOKEEP];

Sequences in the application can use the KEEP attribute, which keeps the original values of sequence.NEXTVAL for the sequence owner, so that the keys match during replay. Most applications need sequence values to be kept at replay. The following example sets the KEEP attribute for a sequence (in this case, one owned by the user executing the statement; for others, use GRANT KEEP SEQUENCE):

SQL> CREATE SEQUENCE my_seq KEEP;

SQL> -- Or, if the sequence already exists but without KEEP:

SQL> ALTER SEQUENCE my_seq KEEP;

Mutable application applies to the local database. It does not traverse database links. It also does not apply for SYS_GUID if pushed down to parallel query slaves.

## Measure Coverage

Destructive testing is a good thing to do and definitely should be run. However, introducing failures is non-deterministic. The application can failover beautifully in all the tests, and then in production a failure occurs elsewhere and unexpectedly some requests do not failover.

Using Application Continuity's Coverage Analysis averts this situation by reporting, in advance, the percentage of requests that are fully protected by Application Continuity, and for those are not fully protected, which they are and where. Use the coverage check in advance of deployment, and the after application changes. Developers and management will know how well protected an application release is from failures of the underlying infrastructure.  If there is a problem it can be fixed before the application is released or waived knowing the level of coverage.

Executing the coverage check is rather like using SQL_TRACE. First run the application in a representative test environment with Application Continuity trace turned on at the server side. The trace is collected in the standard RDBMS user trace directory in user trace files. Then, pass this directory as input to Oracle ORAchk to report the coverage for the application functions that were run. As this check uses Application Continuity, the database and client must be using Oracle 12c. The application does not have to be released with Application Continuity. The purpose of the check is to help to release.

The following is a summary of the coverage analysis.

- If a round-trip is made to the database server and returns while Application Continuity' enabled during capture phase, it is counted as a protected call.
- If a round-trip is made to the database server while Application Continuity' capture disabled (not in a request, or following a restricted call or a disable replay API was called), it is counted as unprotected.
- Round trips that are ignored for the purpose of capture and replay are ignored in the protection-level statistics.

At the end of processing each trace file, a level of protection for the calls sent to the database is computed

For each trace: PASS (>= 75), WARNING ( 25 <= value <75) and FAIL (< 25).

**Steps to Obtain a Coverage Report**

1) Turn on trace for Application Continuity at either a session level or RDBMS level

Enable trace for a single application function you want to check – (Set this before beginRequest or in a callback so replay is not disabled by setting an event.)

```
alter session set events 'trace [progint_appcont_rdbms]' ;
```

Enable trace for all sessions run during the test -

```
alter system set event='trace[progint_appcont_rdbms]' scope = spfile ;
```

2) Run through the application functions. To report on an application function, it needs to have executed. The more application functions run, the better the information that the coverage report provides.

3) Use Oracle ORAchk to analyze the collected database traces and report the level of protection, and where not protected, reports why a request is not protected.

There are four values that control the ORAchk checking for coverage. They can be set either on the command line or via shell environment variables (or mixed). They are the following.

**TABLE 7. USING AC CHECKING FOR PROTECTION LEVEL**

| Command Line Argument | Shell Environment Variable | Usage |
|---|---|---|
| asmhome jarfilename | RAT_AC_ASMJAR | This must point to a version of asm-all-5.0.3.jar that you download fromhttp://asm.ow2.org/. |
| -javahome JDK8dirname | RAT_JAVA_HOME | This must point to the JAVA_HOME directory for a JDK8 installation. |
| -apptrc dirname | RAT_AC_TRCDIR | To analyze the coverage, specify a directory name that contains one or more database server trace files. The trace directory is generally $ORACLE_BASE/diag/rdbms/$ORACLE_SID/trace |
| None | RAT_ACTRACEFILE_WINDOW | When scanning the trace directory, this optional value limits the analysis to scanning to files created in the most recent specified number of days. |

**Example usage:**

```
$ ./orachk -asmhome /tmp/asm-all-5.0.3.jar -javahome /tmp/jdk1.8.0_40 -
apptrc $ORACLE_BASE/diag/rdbms/$ORACLE_SID/trace 3
```

4. Reading the Reports

The coverage check produces a directory named orachk_<uname>_<date>_<time>. This report summarizes coverage and lists trace files that have WARNINGS or FAIL status. Also check in the PASS report (acchk_scorecard_pass.html) in the reports directory to ensure all requests PASS (Coverage(%) = 100). To see all of the details, look for reports/acchk_scorecard_pass.html under the `outfile` subdirectory.

The output includes the database service name, the module name (from v$session.program, which can be set on the client side using the connection property on Java for example,

oracle.jdbc.v$session.program), the ACTION and CLIENT_ID (which can be set using setClientInfo with "OCSID.ACTION" and "OCSID.CLIENTID" respectively).

Example output: found in `ORAchk_.....html#acchk_scorecard`

| Outage Type | Status | Message |
|---|---|---|
| Coverage checks | | **TotalRequest = 2139**<br>**PASS = 2133**<br>**WARNING = 0**<br>**FAIL = 6** |
| | FAIL | [FAIL] Trace file name = SAMPLE_ora_1234.trc Row number = 2222<br>SERVICE NAME = (SAMPLE_WEB_SERVICE.OCS.QA) MODULE NAME = (DEBIT)<br>ACTION NAME = null CLIENT ID = null<br>Coverage(%) = 50 ProtectedCalls = 4 UnProtectedCalls = 4 |
| | FAIL | [FAIL] Trace file name = SAMPLE_ora_5678.trc Row number = 7653<br>SERVICE NAME = (SAMPLE_WEB_SERVICE.OCS.QA) MODULE NAME = (PAYMENTS)<br>ACTION NAME = null CLIENT ID = null<br>Coverage(%) = 20 ProtectedCalls = 1 UnProtectedCalls = 4 |
| | FAIL | [FAIL] Trace file name = SAMPLE_ora_90123.trc Row number = 15099<br>SERVICE NAME = (SAMPLE_WEB_SERVICE.OCS.QA) MODULE NAME = (PAYMENTS)<br>ACTION NAME = null CLIENT ID = null<br>Coverage(%) = 60 ProtectedCalls = 3 UnProtectedCalls = 2 |
| | FAIL | [FAIL] Trace file name = SAMPLE_ora_4747.trc Row number = 789<br>SERVICE NAME = (SAMPLE_WEB_SERVICE.OCS.QA) MODULE NAME = (ACCOUNT)<br>ACTION NAME = null CLIENT ID = null<br>Coverage(%) = 50 ProtectedCalls = 2 UnProtectedCalls = 2 |

### Session State Consistency

Session state consistency describes how non-transactional state is changed during a request.

Examples of session state are NLS settings, optimizer preferences, event settings, PL/SQL global variables, temporary tables, advanced queues, LOBs, and result cache. If non-transactional values change in committed transactions, then use the default value, Dynamic (session_state_consistency is a service level attribute, the default value of which is Dynamic).

Using Dynamic mode, after a COMMIT has executed, if the state was changed in that transaction, then it is not possible to replay the transaction to reestablish that state if the session is lost. Applications can be categorized depending on whether the session state after the initial setup is static or dynamic, and hence whether it is correct to continue past a COMMIT operation.

Dynamic mode is appropriate for almost all applications. If you are unsure, then use Dynamic mode. If your customers or users can modify your application, then you must use Dynamic mode.

For STATELESS applications only, session_state_consistency can be set to STATIC. This allows replay to continue across COMMITs. Application Continuity purges committed transactions. Static mode is strictly for database agnostic applications where session states can be fully restored by either setting FAILOVER_RESTORE to LEVEL1 or by using a callback. This mode is rather similar to TAF. It is unsupported to change state after connection initialization. In this mode requests can be long running, and returned to the pool less frequently.

Do NOT set session_state_consistency if you are unsure whether the application changes state, or if the application can be customized by external consultants. This is a very similar rule, as that for TAF.

Modes of Operation

Application Continuity in its default dynamic mode enables replay at the beginning of a request and disables replay at a successful commit. This is because Application Continuity never replays committed work. The application needs to return connections to the pools. Borrowing and holding connections does not scale.

In long running stateless mode, Application Continuity purges committed transactions and continues. Once disabled by a restricted call re-enable does not occur until the next request begins.

In both modes, capture is disabled by an explicit disable or a restricted call. Capture is re-enabled automatically at the start of the next request. All of this operates automatically.

## Configuring Application Continuity

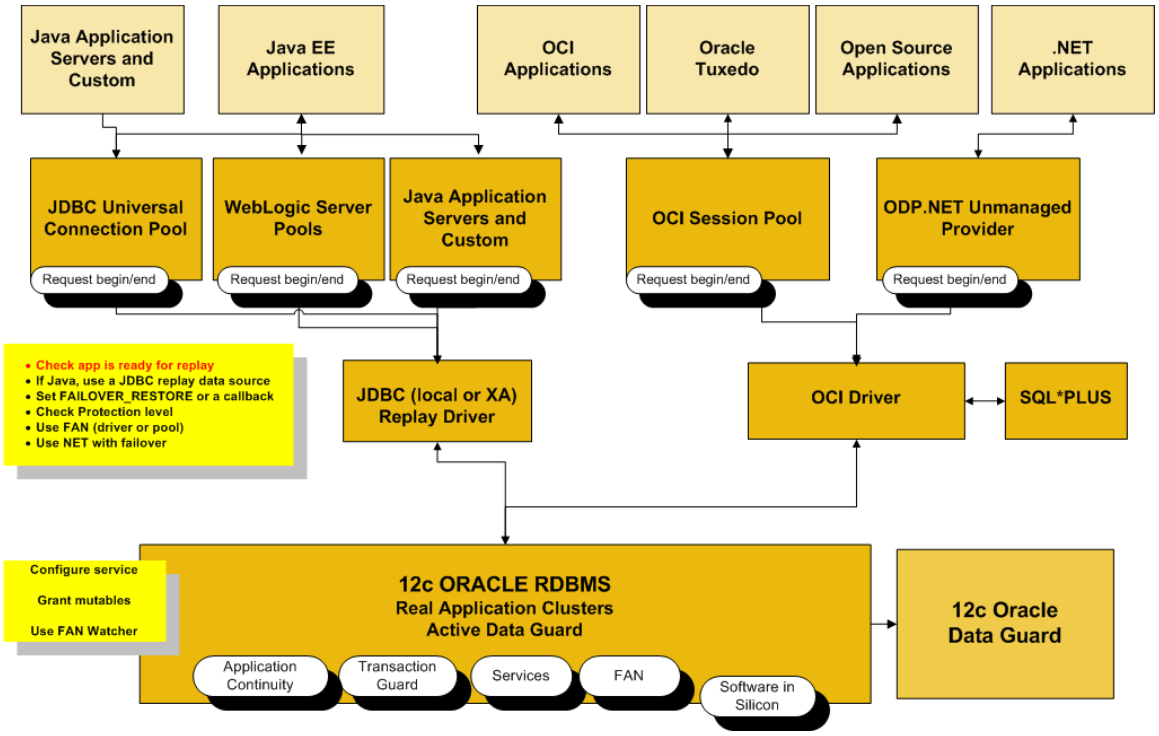


Figure 4. Configuring Application Continuity

### For Java Applications, Configure an Oracle JDBC 12c Replay Driver

Choose from one of the following options depending on your configuration:

**Configure Oracle Universal Connection Pool 12c**

Configure the Oracle JDBC 12c Replay Data Source as a connection factory on UCP `PoolDataSource`:

```
setConnectionFactoryClassName("oracle.jdbc.replay.OracleDataSourceImpl"); or
```

```
setConnectionFactoryClassName("oracle.jdbc.replay.OracleDataSourceImpl");
```

Configure Oracle WebLogic Server 12c

Configure the Oracle 12c JDBC Replay Data Source using the Oracle WebLogic Server Administration Console as shown in the Figure 5 below using the local replay driver or XA replay driver.

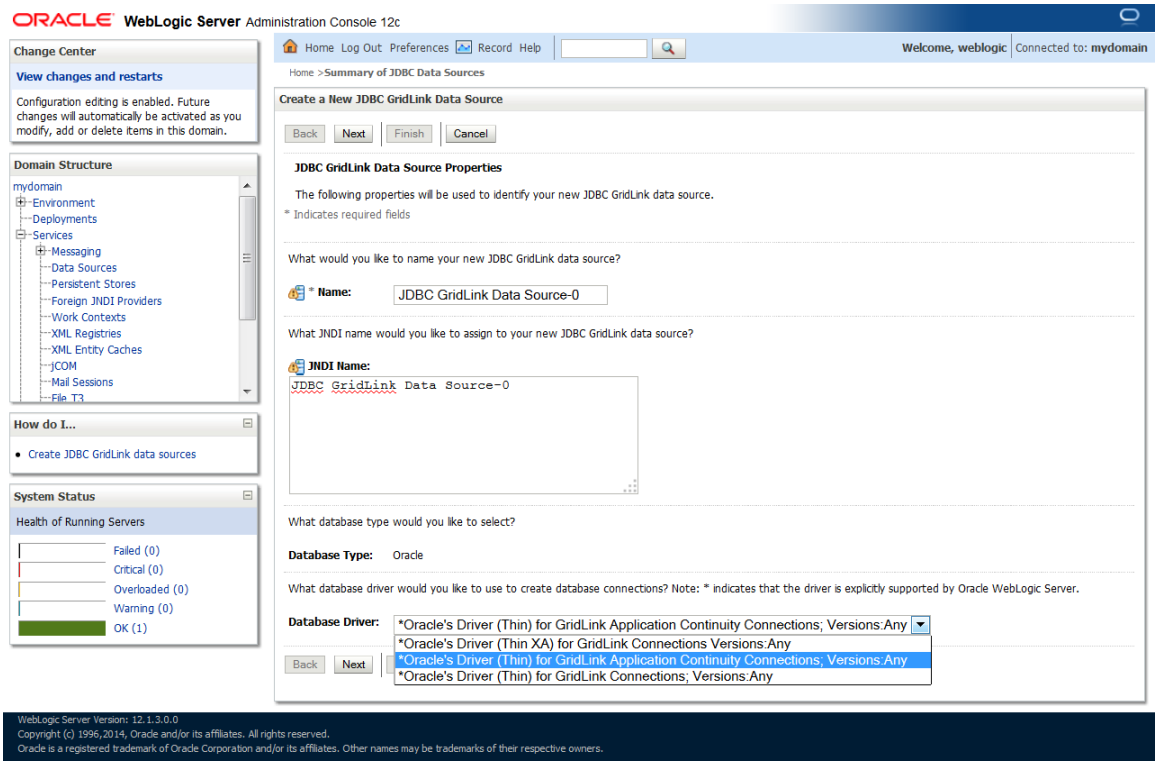


Figure 5. Configuring WebLogic GridLink Data Source with JDBC Replay enabled

### Configure Standalone Java Applications or Third-party Connection Pools

Configure the Oracle JDBC 12c Replay Data Source in the property file or in the thin JDBC application -

```
replay datasource=oracle.jdbc.replay.OracleDataSourceImpl (for non-XA) or
replay datasource= oracle.jdbc.replay.OracleXADataSourceImpl (for XA)
```

## Configure SQL*PLUS

Use the –ac switch for SQL*Plus applications. The –ac flag uses the standard dynamic mode for session state, and disables at commit. Static mode is available only for static applications.

```
sqlplus –ac user/password@ACservice
```

## Configure OCI Session Pool

Application Continuity for OCI requires Database 12cR2 or later. Application Continuity is enabled for the OCI session Pool if Application Continuity is enabled for the service.

Before failover can happen, the outage must be detected. Use the recommended TNS format so the ONS transport for FAN is auto configured. Set events=-true in If oraccess.xml. If oraccess.xml is present with events=-false or events is not specified, this disables the usage of FAN. To maintain FAN with SQL*Plus and PHP when oraccess.xml is in use, also set events=-true.

```
<oraaccess> xmlns="http://xmlns.oracle.com/oci/oraaccess"
    xmlns:oci="http://xmlns.oracle.com/oci/oraaccess"
    schemaLocation="http://xmlns.oracle.com/oci/oraaccess
    http://xmlns.oracle.com/oci/oraaccess.xsd">
  <default_parameters>
    <events>true</events>
  </default_parameters>
</oraaccess>
```

## Configure ODP.NET Unmanaged Provider

Application Continuity for ODP.NET Unmanaged Provider requires Database 12cR2 or later. Application Continuity is enabled if Application Continuity is enabled for the service. FAN is also enabled by default if AQ_HA_NOTIFICATION (-notification) is set to TRUE for the service. Use the recommended TNS format for auto configuring FAN. If not using the recommended TNS, use the recommended TNS.

ODP.NET has its own CONNECT_TIMEOUT. This must be greater than ((RETRY_COUNT+1) * RETRY_DELAY)

## Configure the TNS or URL for High Availability

The following TNS/URL configuration is recommended for successfully connecting at failover, switchover, fallback and basic startup.

Set RETRY_COUNT, RETRY_DELAY, CONNECT_TIMEOUT and TRANSPORT_CONNECT_TIMEOUT parameters in the TNSnames or the URL to allow connection requests to wait for the service and connect successfully.

Set CONNECT_TIMEOUT to a high value to prevent login storms. Low values can result in 'fish feeding frenzies' logging in due to the application or pool cancelling and retrying.

Do not set (RETRY_COUNT+1)*RETRY_DELAY or CONNECT_TIMEOUT larger than your response time SLA. The application should either connect or receive an error within the response time SLA.

These are general recommendations for configuring the connections for high availability. Do not use Easy*Connect as this has no high availability capabilities.

This is the recommended TNS for ALL Oracle drivers for 12.2

```
Alias (or URL) = (DESCRIPTION =
    (CONNECT_TIMEOUT= 120)(RETRY_COUNT=20) RETRY_DELAY=3)
(TRANSPORT_CONNECT_TIMEOUT=3)
      (ADDRESS_LIST =
                  (LOAD_BALANCE=on)
                  ( ADDRESS = (PROTOCOL = TCP)(HOST=primary-scan)(PORT=1521)))
            (ADDRESS_LIST =
                   (LOAD_BALANCE=on)
                  ( ADDRESS = (PROTOCOL = TCP)(HOST=secondary-scan)(PORT=1521)))
           (CONNECT_DATA=(SERVICE_NAME = gold-cloud)))"
```

The REMOTE_LISTENER setting for the database must include the addresses in the ADDRESS_LISTs for all URL used for client connections:

- If any URL uses the SCAN Names, then `REMOTE_LISTENERS` must include the SCAN Name.
- If any URL uses an `ADDRESS_LIST` of host VIPs, then `REMOTE_LISTENERS` must include an ADDRESS list including all SCAN VIPs and all host VIPs.

### Configure Services for Application Continuity

To use Application Continuity, set the service attributes using one of SRVCTL or GDSCTL or DBMS_SERVICE, depending on your system configuration-

`FAILOVER_TYPE` : Set this to `TRANSACTION` to enable Application Continuity

`COMMIT_OUTCOME` : Set this to `TRUE` to enable Transaction Guard

Also review the following service attributes –

`REPLAY_INITIATION_TIMEOUT` : Set this to the duration in seconds after which replay is not started (e.g. 180, 300, 1800 seconds – the override to cancel replay). This timer starts at beginRequest. (default 300 seconds)

`FAILOVER_RETRIES` : Set this to specify the number of connection retries for each replay attempt. (default 18 retries, applied at replay driver)

`FAILOVER_DELAY` : Set this to specify the delay in seconds between connection retries (default 10 seconds, applied at replay driver)

`FAILOVER_RESTORE` : Set LEVEL1 to automatically restore common initial states before replay starts (default NONE) (see table 6 12cR2 and later)

Example

To use SRVCTL to modify the service attributes, use a command similar to the following, where EMEA is the name of the Oracle database, and GOLD is the name of the service:

```
srvctl modify service -db EMEA -service GOLD  -failovertype TRANSACTION
-replay_init_time 300 -failoverretry 30 -failoverdelay 3 -commit_outcome TRUE      –failover_restore LEVEL1 -
drain_timeout 60 –stop_option immediate
```

`Note In 12201 only, for GDSCTL set failover_restore using` DBMS_SERVICE.

To use DBMS_SERVICE package, modify service attributes in the following way:

```
declare
params dbms_service.svc_parameter_array;
begin
params('FAILOVER_TYPE'):='TRANSACTION';
params('REPLAY_INITIATION_TIMEOUT'):=300;
params('FAILOVER_DELAY'):=3;
params('FAILOVER_RETRIES'):=30;
params('FAILOVER_RESTORE'):='LEVEL1';
params('commit_outcome'):='true';
params('drain_timeout'):=60;
params('stop_option'):='immediate';
dbms_service.modify_service('[your service]',params);
end;
```

### Check Resource Allocation

Ensure that the system has the necessary memory and CPU resources.

**Memory**: Replay uses more memory on the client because the calls are retained until the end of a database request. If the number of calls retained is small, then the memory consumption is comparable to AC disabled. At the end of a request, the calls are released. This action differs from no Application Continuity that releases as calls are closed. The sizing is similar to Transparent Application Failover (TAF).

For good performance with Java, if there is sufficient memory, allocate 4 to 8 GB (or more) of memory for the Virtual Machine (VM), for example, by setting – `Xms4096m` for 4 GB.

**CPU**: The driver uses some additional CPU for building proxy objects (Java), managing queues, and for garbage collection. Application Continuity validation is uses Software in Silicon. CPU overhead is reduced on the database side for platforms with current Intel and Sparc chips.

### Timer Alignment

Timer settings are very important. If the application sets a timeout for requests that is lower than the detection and recovery times for the underlying system, there is no time allowed for the underlying recovery and replay before the

request is cancelled. The application timeout needs to allow for the detection and cluster reconfiguration in RAC and Observer and data guard failover. When timers are misaligned, the timer expires and replay starts before the system has recovered. This can result in multiple replays before succeeding, or worse all requests including those not on failed nodes timing out and replaying. Using FAN removes the detection time component.

Systems with very aggressive application timeouts may need to consider using hardware with very low cluster reconfiguration times such as FNDD (2 seconds) on EXADATA for node evictions.

Database recovery must also be tuned by setting FAST_START_MTTR_TARGET. This value can be as low as one. Many systems use in order of 10-30 seconds. Lowering FAST_START_MTTR_TARGET keeps the buffer cache

Lets suppose the application sets READ_TIMEOUT or HTTP_REQUEST_TIMEOUT or a custom timeout. In this example, READ_TIMEOUT is used. The same rules apply for others

**Timer Rules**

- Read_timeout >> EXADATA special node eviction (2 seconds in 12.1.0.2)
- Read_timeout >> Misscount (default 30 sec, modifiable)
- Read_timeout >> Data Guard Observer, FastStartFailoverThreshold (default 30 sec, modifiable)
  And FastStartFailoverThreshold >>> Misscount (at least twice)
- Read_timeout >> FAST_START_MTTR_TARGET
- Read_timeout >> NET level (RETRY_COUNT+1) * RETRY_DELAY

and

- Read_timeout << Replay_Initiation_Timeout (modifiable on the service, default 300 seconds)

To avoid premature cancelling of requests the application timeout should be larger than the maximum of

```
(MISSCOUNT(or FDNN)+ FAST_START_MTTR_TARGET), (FastStartFailoverThreshold +
FAST_START_MTTR_TARGET + OPEN TIME)
```

## Trouble Shooting

Checklist for why a session did not failover

1. Application Continuity is disabled - run orachk coverage report to see where and why
2. The environment changed at failover
   - Mutable values were not kept (see keeping mutable values)
   - Autocommit was not set to the same value as it was at runtime
   - Other states were set outside of requests and a callback or label was not used
3. Results changed at failover
   - The SQL was not run AS OF and returned a different result
   - The DML did something different
4. The test application used V$INSTANCE or V$DATABASE or similar
   - Real applications don't do this – V$ do change
5. The applications tried to commit at replay (a different code path). This is not permitted.

6. The connection did not have enough time to reconnect
    - Use the recommended TNS with RETRY_COUNT and RETRY_DELAY
7. The application timeout is lower than the recovery time
    - The application timed out before it could recover

If all else fails - trace one session at runtime and replay: on the server side using the events

alter system set event='10602 trace name context forever, level 28:trace[progint_appcont_rdbms]:10702 trace name context forever, level 16' scope = spfile ;

# Administration

## Planned Maintenance

For planned maintenance the recommended approach is to drain requests initiated by FAN for Oracle connection pools and Oracle drivers, and third parties using these pools. Use draining in combination with Application Continuity for those requests that do not complete within the allocated time.

Beginning with 12c release 2, you can pre-configure the drain_timeout for how long to allow draining, and the stop option, (usually) immediate that applies after draining. The stop, relocate and switchover operations include a drain timeout and stop mode to override the set values if needed.

Using RELOCATE or STOP commands with drain timeout, the FAN planned down event clears the idle sessions immediately and marks the active sessions to be released when the request completes or, at driver level, when the next connection check occurs. The FAN event causes sessions to drain from the instance without disrupting work for the set drain timeout. If not all sessions have checked in and the drain timeout has been reached, the stop mode applies and the services are stopped (immediate). If configured, Application Continuity will attempt to recover those remaining sessions.

## Killing or Disconnecting a Session without Replay

When Application Continuity is configured and a DBA kills or disconnects a session, Application Continuity attempts to recover that session. However, if you do not want the session to be replayed, use the NOREPLAY keyword:

*Stopping an individual session with no replay*

```
alter system kill session 'sid, serial#, @inst' noreplay;

alter system disconnect session 'sid, serial#, @inst' noreplay;

execute DBMS_SERVICE.DISCONNECT_SESSION('[service name]', DBMS_SERVICE.NOREPLAY) ;
```

*Stopping an individual service with no replay*

With `srvctl stop service` you can specify the instance or the node to stop. Using `–force` and `–noreplay` options will avoid replay if noreplay is required. Examples

```
srvctl stop service -db orcl –instance orcl2 -service orcl_pdb38 –force –stop_option
immediate –noreplay

srvctl stop service -db orcl –node rws3 -service orcl_pdb38 –force –stop_option
immediate –noreplay
```

### Stopping a group of services together

Stop all services that can run at a database level, at an instance level or at a node level as a group. Use `noreplay` options if noreplay is required. Use drain_timeout to drain before aborting the sessions. Examples

```
srvctl stop service -db orcl –drain_timeout 60 –force –stop_option immediate
```

```
srvctl stop service -db orcl –pdb orcl30 –drain_timeout 60 –force –stop_option
immediate
```

```
srvctl stop service -node rws2 –drain_timeout 60 –force –stop_option immediate
```

## Conclusion

Application Continuity masks outages from applications and end users by replaying incomplete database requests following recoverable outages. Many outages and planned maintenance operations are hidden. This prevents occasions when the application raises an error to the user, leaving users not knowing what happened, and forcing the user to re-enter data, or worse that administrators must restart mid-tier servers to cope with the failure. Application Continuity strengthens the fault tolerance of systems and applications that use an Oracle database.

## Appendix - New Database Concepts for Application Continuity

The following terms and concepts are used with Application Continuity

**Recoverable Error**

A recoverable error is an error that arises due to an external system failure, independent of the application session logic that is executing. Recoverable errors occur following planned and unplanned outages of foregrounds, networks, nodes, storage, and databases. The application receives an error code that can leave the application not knowing the status of the last operation submitted. Recoverable errors are enhanced in Oracle Database 12c, to include more errors and to include a public API for OCI. Applications should no longer list error numbers in their code. Application Continuity is invoked following a recoverable error code.

**Reliable Commit Outcome**

From the client perspective, a transaction is committed when an Oracle message (termed Commit Outcome), generated after the transaction redo is written, is received by the client. However, the COMMIT message is not durable. Application Continuity uses Oracle Database 12*c* Transaction Guard to obtain the Commit Outcome reliably when it may have been lost following a recoverable error.

**Database Request**

A database request is a unit of work submitted from the application. It typically corresponds to the SQL and PL/SQL, database RPC calls, and local client-side calls of a single web request on a single database connection. It is generally demarcated by the calls made to check-out and check-in the database connection from a connection pool. For recoverable errors, Application Continuity re-establishes the database session and repeats an uncommitted database request safely.

Typically, database requests that use JDBC follow a standard pattern. Here is a code snippet which shows how many database requests are designed.

1. A database request begins with a `getConnection` call on the `PoolDatasource`.
2. The application's logic is executed. This could include executing SQL, PL/SQL, RPC, or local calls.
3. The transaction is committed.
4. The database request ends when the connection is returned to the connection pool.

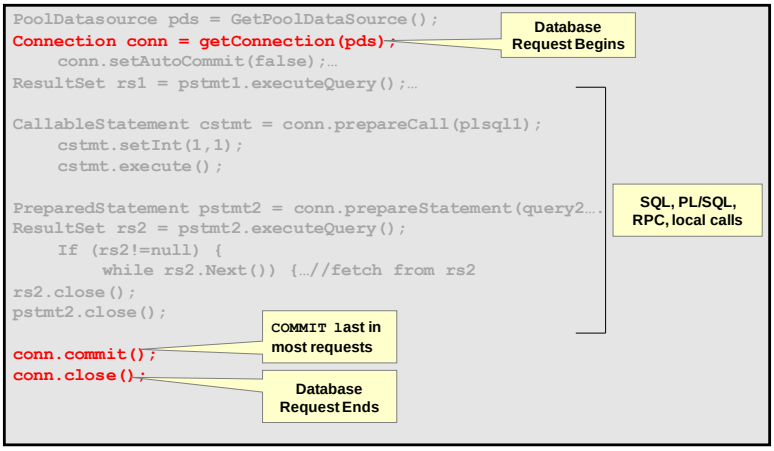


Figure 6. Example of a database request

### Mutable Functions

Mutable functions are functions that can change their results each time that they are called. Mutable functions can cause replay to be rejected because the results visible to the client can change at replay.
Consider sequence.NEXTVAL that is often used in key values. If a primary key is built with a sequence value and this is later used in foreign keys or other binds, the same function result must be returned at replay.

Application Continuity provides mutable value replacement at replay for Oracle function calls if GRANT KEEP or ALTER.. KEEP has been configured. If the call uses database functions that support retaining original mutable values, including sequence.NEXTVAL, SYSDATE, SYSTIMESTAMP, and SYS_GUID, then, the original values returned from the function execution can be saved and reapplied at replay. If an application decides not to grant mutable support and different results are returned to the client at replay, replay for these requests is rejected.

**Oracle Corporation, World Headquarters**
500 Oracle Parkway
Redwood Shores, CA 94065, USA

**Worldwide Inquiries**
Phone: +1.650.506.7000
Fax: +1.650.506.7200

Application Continuity
August 2016
Author: Carol Colrain
Contributing Authors: Kevin Neel, Tong Zhou

CONNECT WITH US

blogs.oracle.com/oracle
facebook.com/oracle
twitter.com/oracle
oracle.com

**Hardware and Software, Engineered to Work Together**