10 Using Oracle RAC Features

The following sections are included in this chapter:

10.1 Overview of Oracle RAC Features

UCP JDBC connection pools provide a tight integration with various Oracle Real Application Clusters (Oracle RAC) Database features. The features include Fast Connection Failover (FCF), Run-Time Connection Load Balancing, and Connection Affinity. These features require the use of an Oracle JDBC driver, Oracle RAC database, and the Oracle Notification Service library (`ons.jar`) that is included with the Oracle Client software.

Applications use Oracle RAC features to maximize connection performance and availability and to mitigate down-time due to connection problems. Applications have different availability and performance requirements and should implement Oracle RAC features accordingly.

Note:

Starting from Oracle Database 11*g* Release 1 (11.2), FCF is also supported by Oracle Restart on a single instance database. Oracle Restart was previously known as Single-Instance High Availability (SIHA).

See Also:

- *Oracle Real Application Clusters Administration and Deployment Guide* for more information about these technologies
- *Oracle Database Administrator's Guide* for more information about Oracle Restart

Generic High Availability and Performance Features

The UCP APIs and connection pool properties include many high availability and performance features that do not require an Oracle RAC database. These features work well with both Oracle and non-Oracle connections and are discussed throughout this guide. For example: validating connections on borrow; setting timeout properties; setting maximum reuse properties; and connection pool manager operations are all used to ensure a high-level of connection availability and optimal performance.

Note:

Generic high availability and performance features work slightly better when using Oracle connections because UCP leverages Oracle JDBC internal APIs.

Database Version Compatibility for Oracle RAC

The following table lists supported Database versions for various Oracle RAC features:

Table 10-1 Oracle RAC Version Compatibility

Feature	Supported Database Version
Fast Connection Failover	Oracle Database 10.1.x and later versions
Run-time Connection Load-Balancing	Oracle Database 10.2.x and later versions
Web Session Affinity	Oracle Database 11.1.x and later versions
Transaction-Based Affinity	Oracle Database 10.2.x and later versions (Oracle Database 11.1.x recommended)

Oracle JDBC Driver Version Compatibility for Oracle RAC

Oracle JDBC driver 10.1.x and later versions are supported with Oracle RAC features.

10.2 About Fast Connection Failover

This section contains the following subsections:

- Overview of Fast Connection Failover
- What is Fast Connection Failover
- Fast Connection Failover Prerequisites
- Example of Fast Connection Failover Configuration
- Enabling Fast Connection Failover
- What is ONS
- Configuring the Connection URL

10.2.1 Overview of Fast Connection Failover

The Fast Connection Failover (FCF) feature is a Fast Application Notification (FAN) client implemented through the connection pool. The feature requires the use of an Oracle JDBC driver and high availability (HA) database configurations like Oracle RAC, Single Restart, or Active Data Guard.

Note:

This section describes only the steps that an application must perform when using FCF with Oracle RAC.

See Also:

Oracle Real Application Clusters Administration and Deployment Guide for more information

FCF is useful in the following scenarios:

- **Unplanned Outages:** FCF rapidly detects non-functioning connections, and then terminates and removes them from the pool. Connection removal relies on terminating the sever-socket connections rapidly to prevent the system from becoming non-responsive. Borrowed and in-use connections are interrupted only for unplanned outages.
- **Planned Outages:** FCF does not interrupt and close the borrowed or in-use connections until work is done and control of the connection is returned to the pool.
- **Irrecoverable Connection Errors and Exceptions:** FCF encapsulates irrecoverable connection errors and exceptions into the `isValid` API for robust and efficient retries.
- **Addition of New Nodes to the Cluster:** FCF recognizes the new nodes that join an Oracle RAC cluster and places new connections on that node appropriately for delivering maximum quality of service to applications at run time. This facilitates middle-tier integration of Oracle RAC node joins and work-request routing from the application tier.
- **Run-Time Work Requests:** FCF distributes run-time work requests to all active Oracle RAC instances.

Unplanned Shutdown Scenarios

FCF supports unplanned shutdown scenarios by detecting and removing stale connections to an Oracle RAC cluster. Stale connections include connections that do not have a service available on any instance in an Oracle RAC cluster due to service-down and node-down events. Borrowed connections and available-but-stale connections are detected, and their network connection is severed before removing them from the pool. These removed connections are not replaced by the pool. Instead, the application must retry connections before performing any work with a connection.

Note:

Borrowed connections are immediately terminated and closed during unplanned shutdown scenarios. Any on-going transactions immediately receive an exception.

Planned Shutdown Scenarios

FCF supports planned shutdown scenarios where an Oracle RAC service can be gracefully shutdown. In such scenarios, stale borrowed connections are marked, and are terminated and removed after they are returned to the pool. Any on-going transactions do not see any difference and proceed to complete.

The primary difference between unplanned and planned shutdown scenarios is how borrowed connections are handled. Stale connections that are idle in the pool (not borrowed) are removed in the same manner as the unplanned shutdown scenario.

UCP also supports graceful connection draining from any planned-down Oracle RAC instance. Affected borrowed connections are removed smoothly over a grace period, instead of immediate removal upon their return to the pool. This helps in avoiding throughput impact and logon storms during any service relocation. In the FAN events, UCP uses the value of the `drain_timeout`[1] parameter as the grace period, when doing graceful draining.

Oracle RAC Instance Rejoin and New Instance Scenarios

FCF supports scenarios where an Oracle RAC cluster adds instances that provide a service of interest. The instance may be new to the cluster or may have been restarted after a down event. In both cases, UCP recognizes the new instance and creates connections to the node as required.

Related Topics

- Checking If a Connection Is Valid
- Enabling Fast Connection Failover

10.2.2 What is Fast Connection Failover?

After Fast Connection Failover is enabled, the mechanism is automatic; no application intervention is needed. This section discusses how a connection failover is presented to an application and what steps the application takes to recover, in the following sections:

- What the Application Sees
- How FCF Works

10.2.2.1 What the Application Sees

By the time an Oracle RAC service failure is propagated to the JDBC application, the database already rolls back the local transaction. The cache manager then cleans up all invalid connections. When an application holding an invalid connection tries to do work through that connection, it is possible to receive `SQLException, ORA-17008, Closed Connection`.

When an application receives a `Closed Connection` error message, it should do the following:

1. Retry the connection request. This is essential, because the old connection is no longer open.
2. Replay the transaction. All work done before the connection was closed has been lost.

Note:

The application should not try to roll back the transaction. The transaction was already rolled back in the database by the time the application received the exception.

10.2.2.2 How FCF Works

Under Fast Connection Failover, each connection in the cache maintains a mapping to a service, instance, database, and host name.

When a database generates an Oracle RAC event, that event is forwarded to the JVM in which JDBC is running. A daemon thread inside the JVM receives the Oracle RAC event and passes

[1] The drain_time parameter specifies the time in seconds, during which a service drains.

it on to the Connection Cache Manager. The Connection Cache Manager then throws SQL exceptions to the applications affected by the Oracle RAC event.

A typical failover scenario may work like the following:

1. A database instance fails, leaving several stale connections in the cache.
2. The Oracle RAC mechanism in the database generates an Oracle RAC event which is sent to the JVM containing JDBC.
3. The daemon thread inside the JVM finds all the connections affected by the Oracle RAC event, notifies them of the closed connection through SQL exceptions, and rolls back any open transactions.
4. Each individual connection receives a SQL exception and must retry.

10.2.3 Fast Connection Failover Prerequisites

Fast Connection Failover is available under the following circumstances:

- The Universal Connection Pool is enabled.

  Fast Connection Failover works in conjunction with the JDBC connection caching mechanism. This helps applications manage connections to ensure high availability.
- The application uses service names to connect to the database.

  The application cannot use service identifiers.
- The underlying database has Oracle Database 12*c* Release 1 (12.1) or later Real Application Clusters (Oracle RAC) capability or Oracle Data Guard configured with either single instance Databases or Oracle RAC.

  If failover events are not propagated, then connection failover cannot occur.
- Oracle Notification Service (ONS) is configured and available on the node where JDBC is running.

  JDBC depends on ONS to propagate database events and notify JDBC of them.
- The Java Virtual Machine (JVM) in which your JDBC instance is running must have `oracle.ons.oraclehome` set to point to your *`ORACLE_HOME`*.

10.2.4 Example of Fast Connection Failover Configuration

The following example demonstrates a connection pool that uses the FCF feature. FCF is configured through a pool-enabled data source. The example includes enabling FCF, configuring the Oracle Notification Service (ONS) and configuring a connection URL. These topics are discussed after the example.

The `isValid` method of the `oracle.ucp.jdbc.ValidConnection` interface is typically 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. For example:

```
try {  conn = pds.getConnection;  ...}catch (SQLException sqlexc)
{
   if (conn == null || !((ValidConnection) conn).isValid())

   // take the appropriate action

...
conn.close();
}
```

Example 10-1 Fast Connection Failover Configuration Example

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionPoolName("FCFSamplePool");
pds.setFastConnectionFailoverEnabled(true);
pds.setONSConfiguration("nodes=racnode1:4200,racnode2:4200\nwalletfile=
/oracle11/onswalletfile");
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
pds.setURL("jdbc:oracle:thin@(DESCRIPTION= "+
   "(LOAD_BALANCE=on)"+
   "(ADDRESS=(PROTOCOL=TCP)(HOST=racnode1) (PORT=1521))"+
   "(ADDRESS=(PROTOCOL=TCP)(HOST=racnode2) (PORT=1521))"+
   "(CONNECT_DATA=(SERVICE_NAME=service_name)))");
...
```

Related Topics

- Checking If a Connection Is Valid

10.2.5 Enabling Fast Connection Failover

The FCF pool property is used to enable and disable FCF. FCF is disabled by default. The following example demonstrates enabling FCF as shown in Example 10-1.

```
pds.setFastConnectionFailoverEnabled(true);
```

Note:

Starting from Oracle Database 12*c* Release 1 (12.1.0.2), UCP supports the `oracle.ucp.PlannedDrainingPeriod` system property. It specifies the grace time period (in integer seconds) over which the pool smoothly drains the borrowed connections affected by a planned shut down. Draining starts when the same Database service becomes available on another instance different from the one that is going down.

When this property is not set, or set to `0`, then the pool closes any affected borrowed connection immediately when it is returned to the pool.

Querying Fast Connection Failover Status

An application determines if Fast Connection Failover is enabled by calling `OracleDataSource.getFastConnectionFailoverEnabled`, which returns `true` if failover is enabled, `false` otherwise.

Note:

FCF must also be enabled to use run-time connection load balancing and connection affinity. These features are discussed later in this chapter.

10.2.6 What is ONS?

FCF relies on the Oracle Notification Service (ONS) to propagate database events between the connection pool and the Oracle RAC database. At run time, the connection pool must be able to setup an ONS environment. ONS (`ons.jar`) is included as part of the Oracle Client software. ONS can be configured using either remote configuration or client-side ONS daemon configuration. Remote configuration is the preferred configuration for standalone client applications. This section discusses the following topics:

- Overview of ONS Configuration File
- Remote Configuration of ONS
- Configuration of Client-Side ONS Daemon

10.2.6.1 Overview of ONS Configuration File

ONS configuration is controlled by the ONS configuration file, `*ORACLE_HOME*/opmn/conf/ons.config`. This file tells the ONS daemon how it should behave. Configuration information within `ons.config` is defined in simple name and value pairs.

Some parameters in the `ons.config` file are required and some are optional. Table 10-2 lists the required ONS configuration parameters and Table 10-3 lists the optional ONS configuration parameters. ONS must be refreshed after updating the `ons.config` file.

Table 10-2 Required ONS Configuration Parameters

Parameter	Explanation
`localport`	Specifies the port that ONS binds to on the local host interface to talk to local clients. For example, `localport=4100`
`remoteport`	Specifies the port that ONS binds to on all interfaces for talking to other ONS daemons. For example, `remoteport=4200`
`nodes`	Specifies a list of other ONS daemons to talk to. Node values are given as a comma-delimited list of either host names or IP addresses plus ports. The port value that is given is the remote port that each ONS instance is listening on. In order to maintain an identical file on all nodes, the `host:port` of the current ONS node can also be listed in the nodes list. It will be ignored when reading the list. For example, `nodes=myhost.example.com:4200,123.123.123.123:4200` The nodes listed in the nodes line correspond to the individual nodes in the Oracle RAC instance. Listing the nodes ensures that the middle-tier node can communicate with the Oracle RAC nodes. At least one middle-tier node and one node in the Oracle RAC instance must be configured to see one another. As long as one node on each side is aware of the other, all nodes are visible. You need not list every single cluster and middle-tier node in the ONS configuration file of each Oracle RAC node. In particular, if one ONS configuration file cluster node is aware of the middle tier, then all nodes in the cluster are aware of it.

Table 10-3 Optional ONS Configuration Parameters

Parameter	Description

`logcomp`	Specifies the ONS components to log. The format is as follows:
	`<component>[<subcomponent>,...];<component>[<subcomponent>,...];...`
	If no subcomponents need to be specified, then do not include the brackets ([]) after the component name. To exclude messages from a subcomponent, precede the subcomponent name with an exclamation mark (!). For example, to exclude messages from the `topology` subcomponent, you use the following format:
	`[all,!topology]`
	Note that before specifying a subcomponent from which you want to exclude messages, you must first ensure that the subcomponent includes the messages.
	Following are the valid values for components:
	• `internal`
	• `ons`
	If you specify the component as `internal`, then there are no valid values for subcomponent. If you specify the component as `ons`, then you can specify the following values for subcomponent:
	• `all`: Specifies all messages
	• `ons`: ONS local information
	• `listener`: ONS listener information
	• `discover`: ONS discover (server or multicast) information
	• `servers`: ONS remote servers currently up and connected to the cluster
	• `topology`: ONS current cluster wide server connection topology
	• `server`: ONS remote server connection information
	• `client`: ONS client connection information
	• `connect`: ONS generic connection information
	• `subscribe`: ONS client subscription information
	• `message`: ONS notification receiving and processing information
	• `deliver`: ONS notification delivery information
	• `special`: ONS special notification processing
	• `internal`: ONS internal resource information
	• `secure`: ONS SSL operation information
	• `workers`: ONS worker threads
	The following example shows that you want to log messages for all the subcomponents under `ons`, except the `secure` subcomponent:
	`logcomp=ons[all,!secure]`

`logfile`	Specifies a log file that ONS should use for logging messages. The default value for log file is `$ORACLE_HOME/opmn/logs/ons.log`. For example, `logfile=/private/oraclehome/opmn/logs/myons.log`

Table 10-3 (Cont.) Optional ONS Configuration Parameters

Parameter	Description
`walletfile`	Specifies the wallet file used by the Oracle Secure Sockets Layer (SSL) to store SSL certificates. If a wallet file is specified to ONS, then it uses SSL when communicating with other ONS instances and require SSL certificate authentication from all ONS instances that try to connect to it. This means that if you want to turn on SSL for one ONS instance, then you must turn it on for all instances that are connected. This value should point to the directory where your `ewallet.p12` file is located. For example, `walletfile=/private/oraclehome/opmn/conf/ssl.wlt/default`
`useocr`	Specifies the value, reserved for use on the server-side, to indicate ONS whether it should store all Oracle RAC nodes and port numbers in Oracle Cluster Registry (OCR) instead of the ONS configuration file or not. A value of `useocr=on` is used to store all Oracle RAC nodes and port numbers in Oracle Cluster Registry (OCR). Do not use this option on the client-side.
`allowgroup`	Specifies the ONS setting to indicate the user group connecting to the `localport`. When set to `true`, ONS allows users within the same OS group to connect to its local port. When set to `false`, ONS only allows the same user running the ONS daemon to access its local port. The default value of this parameter is `false`. When using remote ONS configuration, there is no need to set this parameter.

The `ons.config` file allows blank lines and comments on lines that begin with the number sign (`#`).

10.2.6.2 Remote Configuration of ONS

UCP supports remote configuration of ONS through the `ONSConfiguration` pool property. The `ONSConfiguration` pool property value is a string that closely resembles the content of the `ons.config` file. The string contains a list of `name=value` pairs separated by a new line character (`\n`). You can set this pool property in the following two ways:

- The name can be one of the following: `nodes`, `walletfile`, or `walletpassword`. The parameter string should at least specify the ONS configuration `nodes` attribute as a list of `host:port` pairs separated by a comma. SSL is used when the `walletfile` attribute is specified as an Oracle wallet file.

  The following example demonstrates an ONS configuration string as shown in Example 10-1:

  ```
  ...
  pds.setONSConfiguration("nodes=racnode1:4200,racnode2:4200\nwalletfile=/oracle11/
  onswalletfile");
  ...
  ```

- The name can be only `propertiesfile`. The value is the location of an ONS-specific Java properties file. This file must contain the `oracle.ons.nodes` property, and one or both of the following ONS Java properties:

  – `oracle.ons.walletfile`

  – `oracle.ons.walletpassword`

    The following example illustrates such an `ONSConfiguration` string:

    ```
    pds.setONSConfiguration("propertiesfile=/usr/ons/ons.properties");
    ```

The following is an example of the content of the Java properties `ons.properties` file:

```
oracle.ons.nodes=racnode1:4200,racnode2:4200
oracle.ons.walletfile=/oracle11/onswalletfile
```

Note:

The parameters in the configuration string must match those for the Oracle RAC Database. In addition, if you are using Oracle Application Server, then you must configure ONS using procedures that are applicable to the server.

For standalone Java applications, you must configure ONS using the `setONSConfiguration` method. However, if your application meets the following requirements, then you no longer need to call the `setONSConfiguration` method for enabling FCF:

- Your application is using Oracle Database 12*c* Release 1 (12.1) or later UCP and Oracle RAC Database 12*c* Release 1 (12.1) or later
- Your application does not require ONS wallet or keystore

10.2.6.3 Configuration of Client-Side ONS Daemon

Client-side ONS daemon configuration is typical of applications that run on a middle-tier server such as the Oracle Application Server. Clients in this scenario directly configure ONS by updating the `ons.config` file. The location of the file may be different depending on the platform. Example 10-2 demonstrates an `ons.config` file for Example 10-1:

Note:

For client-side ONS daemon configuration, if the operating system (OS) user that starts the connection pool and the OS user that starts the client-side daemon are different, then they both *must* belong to the same OS group. Also, the value of the `allowgroup` parameter must be set to `true` in the `ons.config` file.

After configuring ONS, you start the ONS daemon with the `onsctl` command. You *must* make sure that an ONS daemon is running at all times.

Using the onsctl Command

After configuring, use *`ORACLE_HOME`*`/opmn/bin/onsctl` to start, stop, reconfigure, and monitor the ONS daemon. Table 10-4 is a summary of the commands that `onsctl` supports.

Table 10-4 onsctl Commands

Command	Effect	Output
`start`	Starts the ONS daemon	`onsctl: ons started`
`stop`	Stops the ONS daemon	`onsctl: shutting down ons daemon...`
`ping`	Verifies whether or not the ONS daemon is running	`ons is running ...`

Table 10-4 (Cont.) onsctl Commands

Command	Effect	Output
`reconfig`	Triggers a reload of the ONS configuration without shutting down the ONS daemon	
`help`	Prints a help summary message for onsctl	
`detailed`	Prints a detailed help message for onsctl	

See Also:

Oracle Real Application Clusters Administration and Deployment Guide

Note:

- The Java Virtual Machine (JVM), in which your JDBC instance is running, must have the `oracle.ons.oraclehome` system property set to the location of `ORACLE_HOME` before starting the application. For example:

  ```
  java -Doracle.ons.oraclehome=$ORACLE_HOME ...
  ```

- Oracle recommends remote configuration of ONS for UCP.

Note:

In Oracle RAC 12.1.0.2.0, by default, server installation requires the value of the `walletfile` ONS parameter to be set, and enforces the use of SSL for all ONS connections.

If you have a UCP application that is already using the `walletfile` parameter in the ONS remote configuration string or local configuration file, then the only requirement is that, for the same topology, the wallet file on the client side must have the same content as the wallet file on the server side. You can make a copy of the server-side file and make it available on the client side.

For UCP applications that are using Oracle RAC features without setting the `walletfile` parameter, you must perform one of the following:

- Add the walletfile parameter setting to the ONS remote configuration string or local configuration file, as shown in Example 10-1. Keep in mind that, for the same topology, the wallet file on the client side must have the same content as the wallet file on the Oracle RAC server side.
- Run the following command to remove the `walletfile` parameter setting from both client and server ONS configuration string and the local configuration file:

  ```
  srvctl modify nodeapps -clientdata
  ```

For secure communication, the ONS auto-configuration in Oracle RAC 12.1.x no longer works when Oracle RAC 12.1.0.2.0 is first installed or patched. Applications have to use explicit ONS configuration (remote or local) instead, and make one of the changes previously discussed.

Example 10-2 Example of a Sample ons.config File

```
# This is an example ons.config file
#
# The first three values are required
localport=4100
remoteport=4200
nodes=racnode1.example.com:4200,racnode2.example.com:4200
```

10.2.7 Configuring the Connection URL

The connection URL of a connection factory must use the service name syntax when using FCF. The service name is used to map the connection pool to the service. In addition, the factory class must be an Oracle factory class. The following example demonstrates configuring the connection URL as shown in Example 10-1:

```
...
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
pds.setURL("jdbc:oracle:thin@//host:port/service_name");
...
```

Note:

An exception is thrown if a service identifier (SID) is specified for the connection URL when FCF is enabled.

The following examples demonstrate valid connection URL syntax when connecting to an Oracle RAC database. Examples for both the Oracle JDBC thin and Oracle OCI driver are included. Notice that the URL can be used to explicitly enable load balancing among Oracle RAC nodes:

Valid Connection URL Usage

```
pds.setURL("jdbc:oracle:thin@//host:port/service_name");

pds.setURL("jdbc:oracle:thin@//cluster-alias:port/service_name");

pds.setURL("jdbc:oracle:thin:@(DESCRIPTION= "+
   "(LOAD_BALANCE=on)"+
   "(ADDRESS=(PROTOCOL=TCP)(HOST=host1)(PORT=1521))"+
   "(ADDRESS=(PROTOCOL=TCP)(HOST=host2)(PORT=1521))"+
   "(CONNECT_DATA=(SERVICE_NAME=service_name)))");

pds.setURL("jdbc:oracle:thin:@(DESCRIPTION= "+
   "(ADDRESS=(PROTOCOL=TCP)(HOST=cluster_alias) (PORT=1521)) "+
   "(CONNECT_DATA=(SERVICE_NAME=service_name)))");

pds.setURL("jdbc:oracle:oci:@TNS_ALIAS");

pds.setURL("jdbc:oracle:oci:@(DESCRIPTION= "+
   "(LOAD_BALANCE=on) "+
   "(ADDRESS=(PROTOCOL=TCP)(HOST=host1) (PORT=1521)) "+
   "(ADDRESS=(PROTOCOL=TCP)(HOST=host2)(PORT=1521)) "+
   "(CONNECT_DATA=(SERVICE_NAME=service_name)))");

pds.setURL("jdbc:oracle:oci:@(DESCRIPTION= "+
   "(ADDRESS=(PROTOCOL=TCP)(HOST=cluster_alias) (PORT=1521)) "+
   "(CONNECT_DATA=(SERVICE_NAME=service_name)))");
```

10.3 About Run-Time Connection Load Balancing

This section contains the following subsections:

- Overview of Run-Time Connection Load Balancing
- Setting Up Run-Time Connection Load Balancing

10.3.1 Overview of Run-Time Connection Load Balancing

In an Oracle Real Application Clusters environment, a connection could belong to any instance that provides the relevant service. In the best case, all instances perform equally well and randomly retrieving a connection from the cache is appropriate.

However, when one instance performs better than others, random selection of a connection is inefficient. The run-time connection load balancing feature enables routing of work requests to an instance that offers the best performance, minimizing the need to relocate work.

UCP JDBC connection pools leverage the load balancing functionality provided by an Oracle RAC database. Run-time connection load balancing requires the use of an Oracle JDBC driver and an Oracle RAC database.

See Also:

Oracle Real Application Clusters Administration and Deployment Guide

Run-time connection load balancing is useful when:

- Traditional balancing of workload is not optimal
- Requests must be routed to make optimal use of resources in a clustered database
- Capacity within the cluster differs and is expected to change over time
- The need to avoid sending work to slow, hung, and non-functioning nodes is required

UCP uses the Oracle RAC Load Balancing Advisory. The advisory is used to balance work across Oracle RAC instances and is used to determine which instances offer the best performance. Applications transparently receive connections from instances that offer the best performance. Connection requests are quickly diverted from instances that have slowed, are not responding, or that have failed.

Run-time connection load balancing provides the following benefits:

- Manages pooled connections for high performance and scalability
- Receives continuous recommendations on the percentage of work to route to database instances
- Adjusts distribution of work based on different back-end node capacities such as CPU capacity or response time
- Reacts quickly to changes in cluster reconfiguration, application workload, overworked nodes, or hangs
- Receives metrics from the Oracle RAC Load Balance Advisory. Connections to well performing instances are used most often. New and unused connections to under-performing instances will gravitate away over time. When distribution metrics are not received, connection are selected using a random choice.

10.3.2 Setting Up Run-Time Connection Load Balancing

Run-time connection load balancing requires that FCF is enabled and configured properly.

In addition, you must configure the Oracle RAC Load Balancing Advisory with service-level goals for each service for which load balancing is enabled:

- The service goal must be set to one of the following:
  - `DBMS_SERVICE.SERVICE_TIME`
  - `DBMS_SERVICE.THROUGHPUT`

  The service goal can be set using the `goal` parameter, and the connection balancing goal can be set using the `clb_goal` parameter.
- The connection balancing goal must be set to `SHORT`. For example,

  ```
  EXECUTE DBMS_SERVICE.MODIFY_SERVICE (service_name => 'sjob' -, goal =>
     DBMS_SERVICE.GOAL_THROUGHPUT -, clb_goal => DBMS_SERVICE.CLB_GOAL_SHORT);
  ```

  Or

```
EXECUTE DBMS_SERVICE.MODIFY_SERVICE (service_name => 'sjob' -, goal =>
   DBMS_SERVICE.GOAL_SERVICE_TIME -, clb_goal => DBMS_SERVICE.CLB_GOAL_SHORT);
```

The connection balancing goal can also be set by calling the `DBMS_SERVICE.create_service` procedure.

Note:

You can set the connection balancing goal to `LONG`. However, this is mostly useful for closed workloads, that is, when the rate of completing work is equal to the rate of starting new work.

Related Topics

- About Fast Connection Failover

See Also:

Oracle Real Application Clusters Administration and Deployment Guide

10.4 About Connection Affinity

This section contains the following subsections:

- Overview of Connection Affinity
- Setting Up Connection Affinity

10.4.1 Overview of Connection Affinity

UCP JDBC connection pools leverage affinity functionality provided by an Oracle RAC database. Connection affinity requires the use of an Oracle JDBC driver and an Oracle RAC database version 11.1.0.6 or higher.

Connection affinity is a performance feature that enables a connection pool to select connections that are directed at a specific Oracle RAC instance. The pool uses run-time connection load balancing (if configured) to select an Oracle RAC instance to create the first connection and then subsequent connections are created with an affinity to the same instance.

See Also:

- "Strict Affinity Mode"
- *Oracle Real Application Clusters Administration and Deployment Guide* for more information about setting up an Oracle RAC database.

UCP JDBC connection pools support the following three types of connection affinity:

- Transaction-Based Affinity

- Web Session Affinity
- Oracle RAC Data Affinity

10.4.1.1 Transaction-Based Affinity

Transaction-based affinity is an affinity to an Oracle RAC instance that can be released by either the client application or a failure event. Applications typically use this type of affinity when long-lived affinity to an Oracle RAC instance is desired or when the cost (in terms of performance) of being redirected to a new Oracle RAC instance is high. Distributed transactions are a good example of transaction-based affinity. XA connections that are enlisted in a distributed transaction keep an affinity to the Oracle RAC instance for the duration of the transaction. In this case, an application would incur a significant performance cost if a connection is redirect to a different Oracle RAC instance during the distributed transaction.

10.4.1.2 Web Session Affinity

Web session affinity is an affinity to an Oracle RAC instance that can be released by either the instance, a client application, or a failure event. The Oracle RAC instance uses a hint to communicate to a connection pool whether affinity has been enabled or disabled on the instance. An Oracle RAC instance may disable affinity based on many factors, such as performance or load. If an Oracle RAC instance can no longer support affinity, the connections in the pool are refreshed to use a new instance and affinity is established once again.

Applications typically use this type of affinity when short-lived affinity to an Oracle RAC instance is expected or if the cost (in terms of performance) of being redirected to a new Oracle RAC instance is minimal. For example, a mail client session might use Web session affinity to an Oracle RAC instance to increase performance and is relatively unaffected if a connection is redirected to a different instance.

10.4.1.3 Oracle RAC Data Affinity

Data affinity describes the concept of ensuring that a group of related cache entries is contained within a single cache partition.

Starting from Oracle Database Release 18c, UCP supports Oracle RAC Data Affinity. When you enable Data Affinity on the Oracle RAC database, data on the affinitized tables are partitioned in such a way that a particular partition or subset of rows for a table is affinitized to a particular Oracle RAC database instance. The affinity leads to higher performance and scalability for the applications due to improved cache locality and reduced internode synchronization and block pings among the RAC instances.

See Also:

Enabling a Custom Partition Assignment Strategy

To use the Oracle RAC Data Affinity feature, the clients accessing the database through UCP must provide the data affinity key in their connection requests. UCP has the following capabilities when pooling connections for an affinity enabled RAC database:

1. UCP learns the topology that contains the data affinity of the data partitions across Oracle RAC instances at pool start up.

2. UCP connection requests that need to leverage the Oracle RAC Data Affinity feature provides the data affinity key using the sharding key builder and use the connection builder as follows:

```
  PoolDataSource pds = new PoolDataSourceImpl();
   // configure the datasource with the database connection properties

/* Builds the RAC data affinity key using the sharding key builder API
and gets a connection from the pool using UCP connection builder */
  OracleShardingKey dataAffinityKey =  pds.createShardingKeyBuilder()
          .subkey(1000, OracleType.NUMBER)
          .build();

  Connection connection = pds.createConnectionBuilder()
          .shardingKey(dataAffinityKey)
          .build();
```

> **Note:**
>
> You can still make connection requests to Oracle RAC Data Affinity-enabled without providing the data affinity key. However, in this case, you will not see the benefits of Oracle RAC Data Affinity feature.

3. UCP determines the affinitized instance for the shard key provided in the request and checks if a connection for that instance exists in the pool. If the connection exists, then it is used to serve the request. If a matching connection does not exist in the pool, then a fallback to Run-Time Load Balancing chooses a connection for the request and serves it. If a new connection needs to be created to serve the request, then the request is routed to the affinitized instance corresponding to the provided shard (data affinity) key.
4. UCP keeps its topology of the data partitions in sync with the server side when there are HA events or when there is a change in the affinity of data partitions on Oracle RAC.

10.4.2 Setting Up Connection Affinity

Perform the following steps to set up connection affinity:

- Enable FCF.

> **See Also:**
>
> "About Fast Connection Failover"

- Enable run-time connection load balancing.

> **See Also:**
>
> "About Run-Time Connection Load Balancing"

- Create a connection affinity callback.

- Register the callback.

> **Note:**
>
> Transaction-based affinity is strictly scoped between the application/middle-tier and UCP. Therefore, transaction-based affinity requires only the `setFastConnectionFailoverEnabled` property be set to `true` and does not require complete FCF configuration.
>
> In addition, transaction-based affinity does not technically require run-time connection load balancing. However, it can help with performance and is usually enabled regardless. If run-time connection load balancing is not enabled, the connection pool randomly picks connections.

This section contains the following subsections:

- Creating a Connection Affinity Callback
- Registering a Connection Affinity Callback
- Removing a Connection Affinity Callback

10.4.2.1 Creating a Connection Affinity Callback

Connection affinity requires the use of a callback. The callback is an implementation of the `ConnectionAffinityCallback` interface which is located in the `oracle.ucp` package. The callback is used by the connection pool to establish and retrieve a connection affinity context and is also used to set the affinity policy type (transaction-based or Web session).

The following example demonstrates setting an affinity policy in a callback implementation. The example also demonstrates manually setting an affinity context. typically, the connection pool sets the affinity context inside an application. However, the ability to manually set an affinity context is provided for applications that want to customize affinity behavior and control the affinity context directly.

```
public class AffinityCallbackSample
   implements ConnectionAffinityCallback {

   Object appAffinityContext = null;
   ConnectionAffinityCallback.AffinityPolicy affinityPolicy =
   ConnectionAffinityCallback.AffinityPolicy.TRANSACTION_BASED_AFFINITY;

   //For Web session affinity, use WEBSESSION_BASED_AFFINITY;

   public void setAffinityPolicy(AffinityPolicy policy)
   {
      affinityPolicy = policy;
   }

   public AffinityPolicy getAffinityPolicy()
   {
      return affinityPolicy;
   }

   public boolean setConnectionAffinityContext(Object affCxt)
   {
      synchronized (lockObj)
      {
         appAffinityContext = affCxt;
      }
      return true;
   }

   public Object getConnectionAffinityContext()
   {
      synchronized (lockObj)
      {
         return appAffinityContext;
      }
   }
}
```

10.4.2.2 Registering a Connection Affinity Callback

A connection affinity callback is registered on a connection pool using the `registerConnectionAffinityCallback` method. The callback is registered when creating the connection pool. Only one callback can be registered per connection pool.

The following example demonstrates registering a connection affinity callback implementation:

```
ConnectionAffinityCallback callback = new MyCallback();

PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionPoolName("AffinitySamplePool");
pds.registerConnectionAffinityCallback(callback);
...
```

10.4.2.3 Removing a Connection Affinity Callback

A connection affinity callback is removed from a connection pool using the `removeConnectionAffinityCallback` method. For example:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionPoolName("AffinitySamplePool");
pds.removeConnectionAffinityCallback();
...
```

10.4.2.4 Strict Affinity Mode

By default, affinity is only a hint. A connection pool selects a new Oracle RAC instance for connections if it does not find a connection on a desired instance. You can change this behavior by switching the strict affinity mode on. The strict affinity mode throws a UCP exception if a connection on a desired instance is not found.

Use the following pool properties to switch on the strict affinity mode:

- The `useStrictWebSessionAffinity` property

  Set the `useStrictWebSessionAffinity` property to `true` or `false` for switching the strict Web session affinity mode on or off respectively.
- The `useStrictXAAffinity` property

  Set the `useStrictXAAffinity` property to `true` or `false` for switching the strict transaction-based affinity mode on or off respectively.

These properties can be handled through the `UniversalConnectionPoolMBean`.

Related Topics

- UniversalConnectionPoolMBean

10.5 Global Data Services

This section describes the new Global Data Services (GDS) feature that can be used with Universal Connection Pool:

- Overview of Global Data Services
- Configuring an Application for Using GDS

10.5.1 Overview of Global Data Services

Global Data Services (GDS) feature is available since Oracle Database 12*c* Release 1 (12.1). Through this feature, Fast Connection Failover, Run-time Connection Load-Balancing, and Connection Affinity features that were available only in Oracle RAC, were extended to a set of replicated databases offering common services.

The set of databases may include Oracle RAC and single-instance Oracle databases interconnected through Data Guard, GoldenGate, or any other replication technology. A database service that can be provided by multiple databases is called a global service, so that it can be distinguished from the traditional service that can be provided only by a single database. This combination enables services to be deployed anywhere within this globally distributed configuration, supporting load balancing, high availability, database affinity, and so on.

See Also:

Oracle Database Global Data Services Concepts and Administration Guide

10.5.2 Configuring an Application for Using GDS

UCP connects to Global Data Services in the same way that it connects to local services on an Oracle RAC. The service name in the connection string should be the name of the global service. The endpoint should be the endpoint of a GDS listener instead of the endpoint for the local, remote, or SCAN listener of a database.

A client must specify its region in the `REGION` parameter of the connection string. This is a new requirement for GDS. The region name is required because, in case of GDS, Run-time Load Balancing advisory is customized for particular regions. Following is an example of a typical connection string:

```
(DESCRIPTION=
  (ADDRESS=(GDS_protocol_address_information))
  (CONNECT_DATA=
   (SERVICE_NAME=global_service_name)
   (REGION=region_name)))
```

Like with local services, UCP can specify multiple GDS listeners in the same connection string for listener failover, load balancing, or both.

Note:

SCAN is not supported for GDS listeners, therefore endpoint for each listener must be specified.

```
(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)))
```

The `REGION` parameter is optional if only global service managers from the local region are specified in the client connection string. This is the case when there is only one region in the GDS configuration, or can be the case when there are multiple regions. But, it is not feasible to change the connection string of the an existing client designed to work with a single database. If the `REGION` parameter is not specified, then the client's region is assumed to be the region of the global service manager used to connect to the global service.

Note:

Unless the `REGION` parameter is specified in the connection string, you can use a pre-12c thin JDBC client only with a GDS configuration that has a single region.

All GDS listeners in the preceding example belong to the same region where UCP is running, that is the local region. To provide high availability, when all GDSs in the local region are unavailable, you can specify the GDS listeners for the buddy region in additional `ADDRESS_LIST` descriptors.

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

You do not need manual ONS configuration because UCP automatically retrieves the ONS connection information that is optimally customized for the UCP region from GDS.

Note:

- To enable automatic ONS configuration for GDS, you must enable Fast Connection Failover (FCF) on UCP.
- Automatic ONS configuration works only with Oracle GDS and Oracle RAC. It does not work with single-instance Oracle Databases.

  Automatic ONS configuration does not support ONS wallet or keystore parameters. If your application requires any of these parameters, then you must configure ONS explicitly in either of the following two ways:

  – Calling the `PoolDataSource.setONSConfiguration(String)` method

  – Adding the ONS wallet or keystore parameters in the local ONS configuration file