Table of Contents

CORAL connection service configuration

This page describes the implementation of the CORAL connection service. The main hook for configuring this component is the IConnectionServiceConfiguration.h C++ interface.

Additional information (partially obsolete!) is available in the original documentation for the CORAL ConnectionService component.

1. CORAL connection pool configuration

CORAL distinguishes between database connections and database sessions.

- A CORAL **connection** represents a physical connection (a socket) between the client and the database.
- A CORAL **session** represents a logical session allowing a user on the local client to interact with the remote database. Each session must be opened over an existing connection.
- If **connection sharing** is disabled, each connection may host only 0 (idle) or 1 session. If connection sharing is enabled, each connection may host 0 (idle), 1 or more sessions.
- Connections are created/opened before sessions and must be deleted/closed after sessions.

CORAL connections are managed by a **connection pool**.

- Each connection is uniquely associated to a **URL** (a CORAL 'connection string'). Different URLs require different connections. For the same URL there can be more than one connections opened in the connection pool.
- When a new session is created on a new URL, a new connection is created.
- When a new session is created on a URL for which there are already connections in the connection pool, it can be created on an existing connection (if the connection is idle, or if connection sharing is enabled and the maximum number of sessions per connection has not been reached), or trigger the creation of a new connection for that URL.

Within the connection pool, connections have a limited lifetime.

- When the last logical session is closed on a connection, the connection is said to be **idle**.
- The **connection timeout** parameter determines the time after which an idle connection is considered **expired**. Expired connections are scheduled for deletion by the automatic cleanup thread of the connection pool: however they can be reused if new sessions are requested before the cleanup thread has deleted them.

The cleanup of connections from the pool is managed either by the pool itself or by a separate **pool automatic cleanup thread**, which may enabled or disabled.

- If the cleanup thread is enabled, it regularly checks for expired connections. Every **pool automatic cleanup period**, expired connections are deleted and removed from the pool.
- In the main client thread, the pool also checks for expired connections whenever a new session is opened (for instance, if two connections are expired, one will be used to start a new session and one will be deleted), when the pool is deleted at the end of the client job, and also on explicit user calls of the `purgeConnectionPool` method. This is the only cleanup mechanism available if the clenup thread is disabled.
- If the connection timeout parameter is set to 0, idle connections become immediately expired and are immediately deleted and removed from the pool. In this case, the automatic cleanup thread would be useless and is never started (even if it is set as 'enabled').

Connection timeout

As described above, the **connection timeout** parameter determines after how long an **idle** physical connection is marked as **expired** and becomes a candidate to be cleaned up (i.e. closed) by the **connection pool** or the **pool automatic cleanup thread**.

This parameter can be set using `setConnectionTimeOut()` and checked using `connectionTimeOut()`.

The connection timeout parameter is NOT the connection **pool automatic cleanup period**! The latter is described in the next section.

Automatic cleanup thread

The **pool automatic cleanup thread** can be enabled and disabled using `enablePoolAutomaticCleanUp()` and `disablePoolAutomaticCleanUp()` and checked using `isPoolAutomaticCleanUpEnabled()`.

Note that there will be NO automatic cleanup thread if `connectionTimeOut()` is 0, even if `isPoolAutomaticCleanUpEnabled()` is true! See the implementation `coral::ConnectionService::ConnectionPool::startPool()`: the thread is not started if this parameter is 0. In other words:

- if connection timeout is 0, this implies that the connection cleanup thread is disabled (it is therefore recommended not to set connection timeout to 0 and enable the connection pool cleanup thread, as the result may be counter-intuitive)
- if the connection cleanup thread is disabled, this does NOT imply that the connection timeout is 0: as mentioned above, an idle connection can still be reused in this case, even in a single threaded program, if a new session is opened on it before the connection becomes expired

For these reasons, the connection service is often configured to have BOTH the cleanup thread disabled AND the timeout set to 0 (the latter would be enough, but not the former).

Note that the pool automatic cleanup thread can be started any time if it has not started yet, but it can never be disabled if it has already started, i.e. if the `connect()` method has already been called at least once (see CORALCOOL-948).

Connection pool automatic cleanup period

The connection **pool automatic cleanup period** determines the frequency with which the connection pool checks whether any of its idle connections are expired and can be closed.

Until CORAL_2_3_11 included there was no way to change the default value of this parameter (which was hardcoded to 10 seconds). More recently the possibility to change the default value via the environment variable `CORAL_CONNECTIONPOOL_CLEANUPPERIOD` has been added (see CORALCOOL-847).

2. CORAL connection retrial

The CORAL connection service implements a connection retrial functionality. The creation of a new connection to a specific physical replica, when requested, may fail for various reasons. In some cases, CORAL will attempt to retry to connect to the same replica, while in other cases it will simply give up and will try to fail over to the next available replica instead.

In the specific case of Oracle, there are several cases where CORAL will not attempt to reconnect, because it has been observed in the past that these connection attempts were effectively leading to application "hangs"

from a user point of view. The list of Oracle errors on which retrial is not attempted is implemented in the OracleAccess Session.cpp file. Examples include ORA-28000 (account locked - fixing the hang reported in CORALCOOL-1200), ORA-28001 (password expired - fixing the hang reported in CORALCOOL-998), ORA-01005 (null password given - fixing the hang reported in CORALCOOL-1186) and many more.

Connection retrial is attempted with a specific period of time between each retrial (retry period), during a well defined total time (retry timeout), on the given physical replica. If retrial still fails after the retrial timeout, the corresponding connection string is registered on an internal list of excluded replicas, for a specific duration of time (exclusion timeout). After this timeout, the connection is re-admitted in the replica list. The three parameters retry period, retry timeout and exclusion time can be modified at run time by the user, via the IConnectionServiceConfiguration.h C++ interface.

Tuning these parameters may be important to avoid connection issues under certain circumstances. Some examples of real life issues where this may have been useful are linked to ticket CORALCOOL-1559.
