Network glitch

Introduction

The Network Glitch (NG) is a temporary interruption of the network connection between a client and a DataBase (DB) server during an open session or a transaction. This kind of problem affected all the LHC experiments that indeed observed and warned about it in different Savannah tickets such as in the bug #58522.The problem appears as an Oracle Error ORA-03113 and/or ORA-03114.

Following the explanation on the web ORACLE site, this problem might be issued by many different situations, among them the relevant are:

- Oracle shadow process on the server died unexpectedly. So, if a running process were to suddenly encounter an ORA-03113 and /or 3114, the first place to check is the alert.log on the server to see if any other Oracle errors occurred;

- machine crash or network failure at the server side;

- there are two servers with the same node names on the same network;

- there are duplicate IP addresses on the network.

The previous error is then followed by ORA-24327, creating an infinitive loop that causes the crash of the jobs on the grid.

Methodology

The approach to solve the problem envisaged the creation of a test suit able to reproduce and study the details of the Oracle errors. The implementation was performed in Python.

The reproduction of the error was achieved by means a suitable environment where the NG was simulated using a “Local Port Forwarding” (LPF) between the client and the DB server across a gateway machine. The glitch was produced killing the communication between the client and the gateway. In fig. 1 the schema of the connection with the LPF is reported.

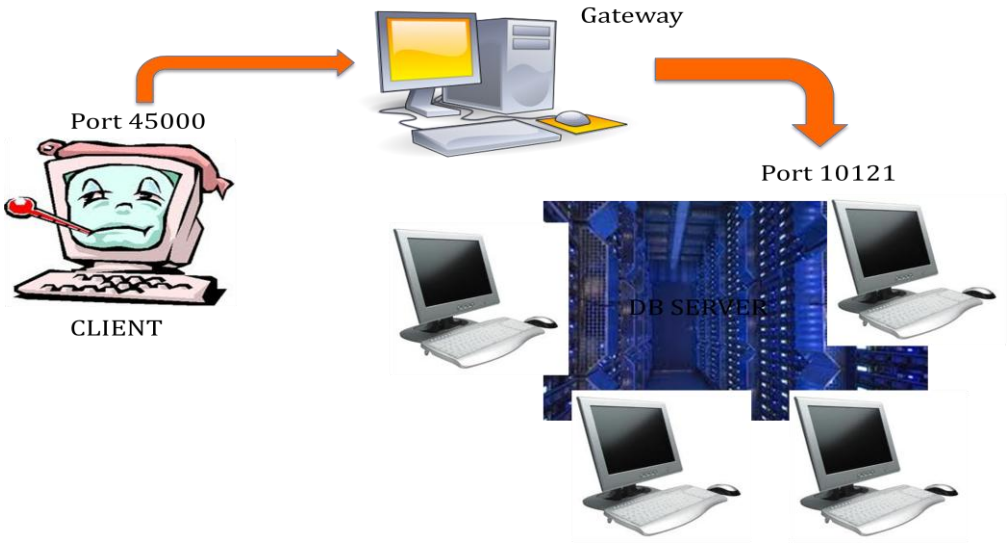

Fig.1

Afterwards a PyCoral unitest module was implemented to chart the workflow of the whole process including the NG after each step of the client DB session with different Isolation level and transaction type (ReadOnly serializable, ReadOnly not serializable, Read/Write). The unit test module in reported in appendix 1.

The set of tests was intended to be successful only when the expected behavior is observed. With this aim, the implementation of the new CORAL feature was tested and considered successful only when the reconnection for the ReadOnly not serializable transaction is achieved, throwing an exception in the other cases. In the following the reason of this choice will be clarified.

Implementation details

Objects involved in the DB client connection with OCI

The characters of the OCI client connection to a DB server are contained in two different packages of CORAL: ConnectionService, CoralCommon, OracleAccess.

The classes involved in ConnectionService are:

- SessionProxy;
- ConnectionHandle.

The classes involved in CoralCommon are:

- IDevConnection;
- IDevSession.

The class involved in OracleAccess are:

- Connection
- ConnectionProperties
- Session
- SessionProperties
- Transaction.

Normal workflow

In the class SessionProxy of ConnectionService the method “open” is invoked. It calls ConnectionPool getting a session from a new connection, by means the method “getSessionFromNewConnection”. The SessionHandle class is thus called which open ConnectionHandle, which in turn invokes the new OracleAccess Session object. It set the OCI parametrs fro the DB client session. The sequence is represented in the Fig.2 .

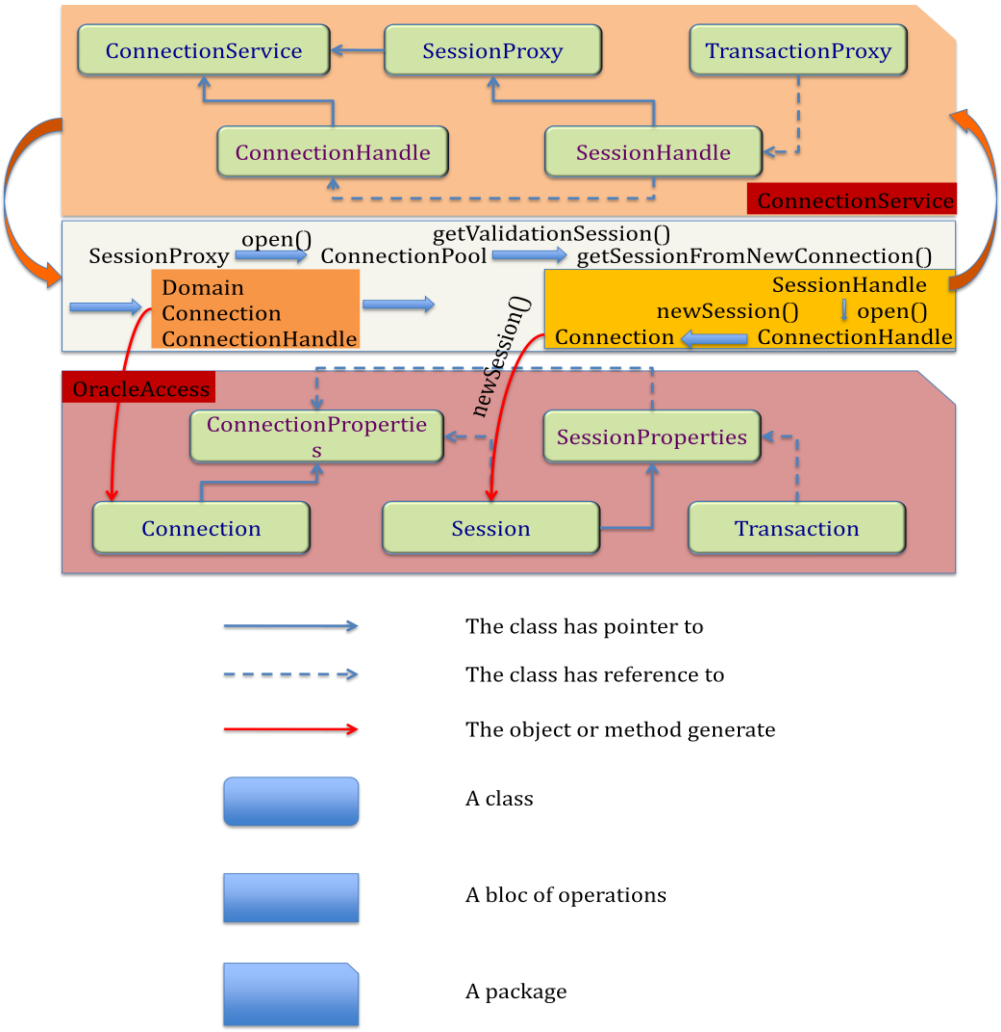

Fig. 2

Introduction to OCI

Almost all OCI calls include in their parameter list one or more handles. A handle is an opaque pointer to a storage area allocated by the OCI library. You use a handle to store context or connection information, (for example, an environment or service context handle), or it may store information about OCI functions or data (for example, an error or describe handle).

The used OCI pointers are:

- OCI environment handle OCIEnv
- OCI error handle OCIError
- OCI server handle OCIServer
- OCI service context handle OCISvcCtx
- OCI transaction handle OCITrans

The environment handle is allocated and initialized with a call to OCIEnvCreate()

All user-allocated handles are initialized using the OCI handle allocation call, OCIHandleAlloc().

An application must free all handles when they are no longer needed. The OCIHandleFree() function frees all handles.

Environment Handle
The environment handle defines a context in which all OCI functions are invoked. Each environment handle contains a memory cache, which enables fast memory access. All memory allocation under the environment handle is done from this cache. Access to the cache is serialized if multiple threads try to allocate memory under the same environment handle. When multiple threads share a single environment handle, they may block on access to the cache. The environment handle is passed as the parent parameter to the OCIHandleAlloc() call to allocate all other handle types. Bind and define handles are allocated implicitly.

Error Handle
The error handle is passed as a parameter to most OCI calls. The error handle maintains information about errors that occur during an OCI operation. If an error occurs in a call, the error handle can be passed to OCIErrorGet() to obtain additional information about the error that occurred. Allocating the error handle is one of the first steps in an OCI application because most OCI calls require an error handle as one of its parameters.

Service Context and Associated Handles
A service context handle defines attributes that determine the operational context for OCI calls to a server. The service context contains three handles as its attributes, that represents a server connection, a user session, and a transaction.
A server handle identifies a connection to a database. It translates into a physical connection in a connection-oriented transport mechanism.
A user session handle defines a user's roles and privileges (also known as the user's security domain), and the operational context in which the calls execute.
A transaction handle defines the transaction in which the SQL operations are performed. The transaction context includes user session state information, including any fetch state and package instantiation.
Breaking the service context down in this way provides scalability and enables programmers to create sophisticated multitiered applications and transaction processing (TP) monitors for execute requests on behalf of multiple users on multiple application servers and different transaction contexts.
You must allocate and initialize the service context handle with OCIHandleAlloc() or OCILogon() before you can use it. The service context handle is allocated explicitly by OCIHandleAlloc(). It can be initialized using OCIAttrSet() with the server, session, and transaction handle. If the service context handle is allocated implicitly using OCILogon(), it is already initialized.
Applications maintaining only a single user session for each database connection at any time can call OCILogon() to get an initialized service context handle.

In applications requiring more complex session management, the service context must be explicitly allocated, and the server and user session handles must be explicitly set into the service context. OCIServerAttach() and OCISessionBegin() calls initialize the server and user session handle respectively.
An application will only define a transaction explicitly if it is a global transaction or there are multiple transactions active for sessions. It will be able to work correctly with the implicit transaction created automatically by OCI when the application makes changes to the database.

Network Glitch error origin and solution strategy

If during the client–DB server communication a NG occurs the OCI pointers, set to establish the physical connection, are no longer valid. Therefore, as soon as they are used by other OCI functions, an exception is thrown with the following error:

CORAL/RelationalPlugins/oracle Error ORA-03113: end-of-file on communication channel

The strategy adopted to solve the problem is a check of the validity of the OCI able to trigger the glitch and restart the connection where needed. All the OCI pointers are thus re-initialized in order to renew the validity of the session and transaction.
However this procedure is started only for the ReadOnly not serializable transaction. The default in CORAL for the Isolation Level is defined by serializable for both Read/Write and ReadOnly transaction. Since a serializable transaction provides a consistent view of data as it existed at its start time T0, in case of network glitch the reconnection and consequently the restart of the session and the transaction would move T0 towards T1 without the user being aware, getting the same effect of the not serializable transaction but in a dirty and unclear way. Therefore the coherence with the data integrity suggests to trigger the reconnection procedure only for not serializable transaction, throwing an exception for any other case.

In the fig.3 the workflow of the main test.

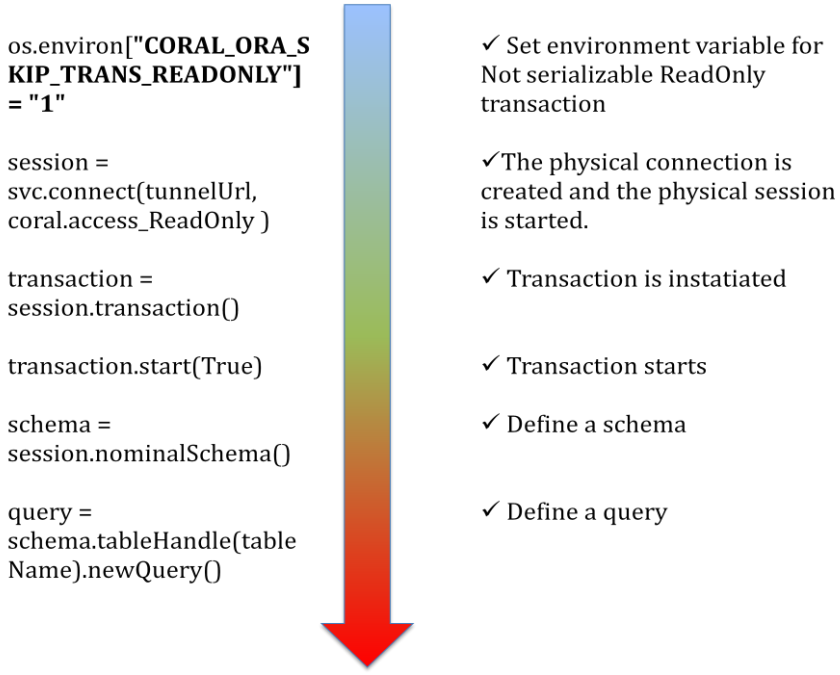

Fig.3

The method “connect” of ConnectionService object (svc) access the class Connection of Oracle Access package and open the DB connection initializing the OCI pointers. Afterwards a new session for the user starts (fig.4). Thus the not serializable transaction can be initialized and started. The schema and the query are then initialized as well.
The network glitch could occur in any time of this process. Since in each step an OCI pointer is initialized using the ones defined for the connection, the strategy envisage a check at the beginning of each call in order to prevent the usage of invalidated OCI as consequence of the lost connection.

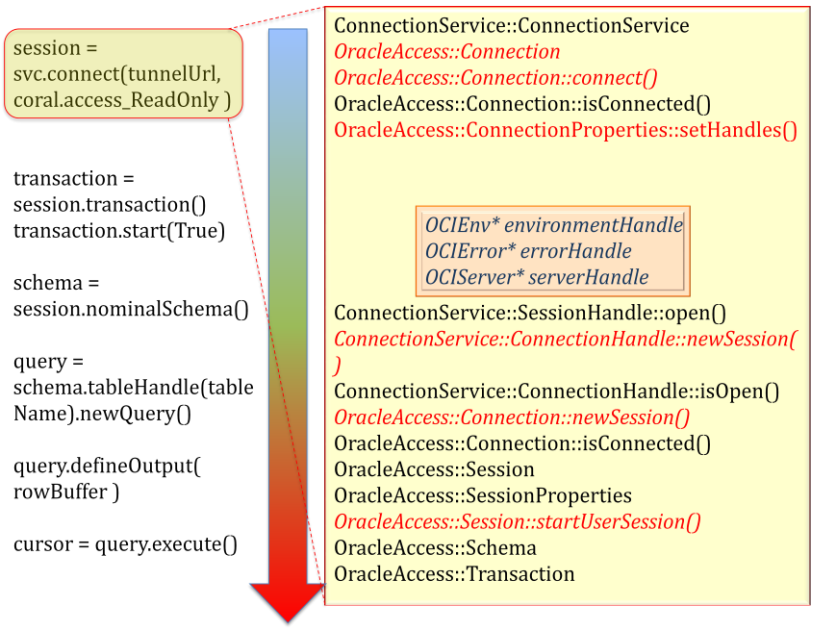

Fig.4

For this reason in the class Session of OracleAccess a new function “checkConnection” was included. In the fig.5 its implementation is reported:

```
void
coral::OracleAccess::Session::checkConnection()
{
  if ( m_properties->wasConnectionLost() )
  {
    m_connection.invalidateAll();

    if(m_properties->restartConnectionWithTimeOut())
    {
      restartSession();
    }
  }
  else if(m_properties->wasSessionLost())
  {
    restartSession();
  }
}
```

This function checks whether the connection was lost. In this case, it invalidates all the sessions and opens a new connection attempting more times until it succeeds and restart a session or times out. The validity of a session is defined by means a Boolean value included as member of the Session class, as inherited by CoralCommon::IDevSession. Therefore the method “invalidateAll”, when invoked, change the value from True to False for any open session.
Sometime, if two different sessions have been started in the same connection, it could happen that one session has already re-initialized the OCI for the connection after a glitch. Thus, for the second session is not necessary to open a new connection. On the contrary this is

prevented because of the sharing of the OCI across the sessions in the same connection. Instead, simply a new session starts.
All the functions used in the "checkConnection" are implemented as members of the OracleAccess::SessionProperties. Their implementation is reported below:

```
bool
coral::OracleAccess::SessionProperties::wasConnectionLost()
{
  //boost::mutex::scoped_lock lock( m_mutex );

  bool wasLost = false;
  text serverVersion[1000];
  sword status = OCIServerVersion( m_connectionProperties.serverHandle(),
m_connectionProperties.errorHandle(),
                                   serverVersion, 1000, OCI_HTYPE_SERVER );

  if ( status != OCI_SUCCESS )
  {
    wasLost = true;
  }

  return wasLost;
}

bool
coral::OracleAccess::SessionProperties::wasSessionLost()
{
  //boost::mutex::scoped_lock lock( m_mutex );

  bool wasLost;

  if( !static_cast<coral::IDevSession&>(m_session).isSessionValid() )
  {
    wasLost = true;

    m_environmentHandle = m_connectionProperties.environmentHandle();
    m_errorHandle = m_connectionProperties.errorHandle();
    m_serverHandle = m_connectionProperties.serverHandle();
  }

  else
  {
    wasLost = false;
  }

  return wasLost;
}
```

To check whether the connection is still valid the OCIServerVersion is used, returning the version string of the Oracle server in case of valid connection.
This procedure applies whenever the methods "transaction", "nominalSchema" and "schema" of OracleAccess::Session are invoked, such as in the following lines of the python test:

```
transaction = session.transaction()

schema = session.nominalSchema()

query = schema.tableHandle(tableName).newQuery()
```

In the method "start" of the OracleAccess::Transaction class a similar procedure is applied, checking also whether a previous transaction was already active. The code is reported in the following:

```
coral::OracleAccess::Transaction::start( bool readOnly )
{
  if ( m_sessionProperties.wasConnectionLost())
  {

    const ConnectionProperties& cproperties =
```

```
m_sessionProperties.connectionProperties();
    // AK: small hack
    ///AK: FIXME: access to connection from transaction?

const_cast<ConnectionProperties&>(cproperties).getConnection().invalidateAll
();

    if(m_sessionProperties.restartConnectionWithTimeOut())
    {
      m_sessionProperties.restartSession();
    }
  }
  else if(m_sessionProperties.wasSessionLost())
  {
    m_sessionProperties.restartSession();
  }

  if ( this->isActive() ) {
    coral::MessageStream log(
m_sessionProperties.domainProperties().service()->name() );
    log << coral::Warning << "A transaction is already active" <<
coral::MessageStream::endmsg;
    return;
  }
```

The new implementation is able to recover the glitch until the query execution. If the glitch occurs later, for instance during the loop of a cursor on the table data, an exception arises.

In fig.5a the UML is represented of the most important class of the ConnectionService package involved in the network glitch implementation.

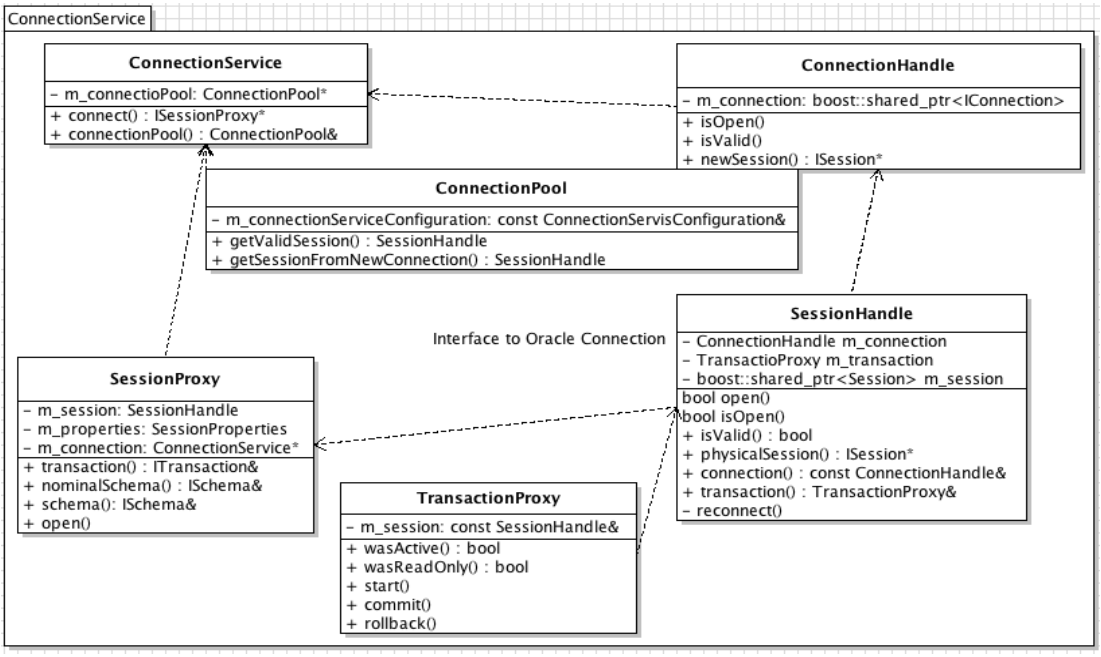

Fig.5a

In the fig.5b the UML for the most important class of OracleAccess package are reported highlighting in red the change with respect the original version.

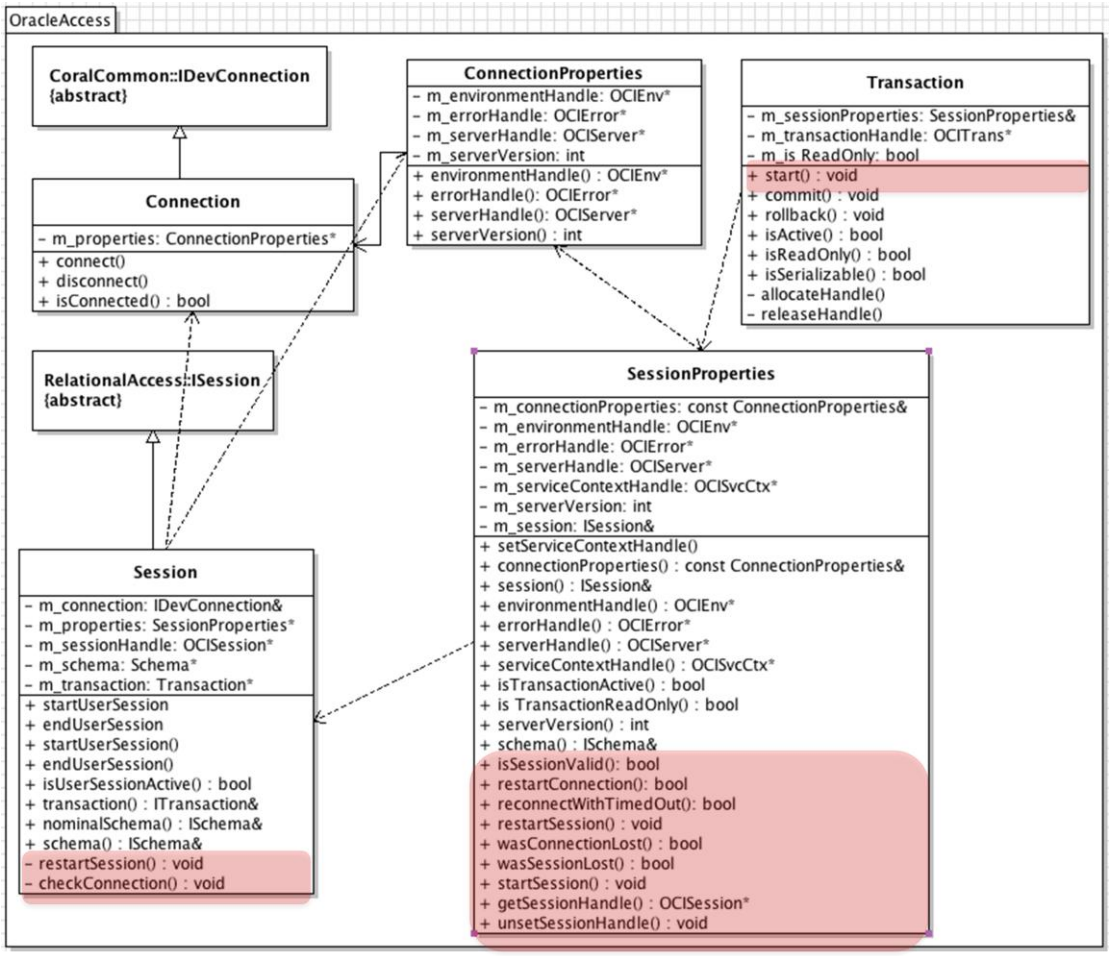

Fig.5b

Appendix 0

Transaction in Oracle

The transaction is the bedrock of the data integrity in a multiuser databases.
A transaction is defined as e single indivisible piece of work that affects some data. Once a transaction is committed, the changes become permanent and are made visible to other transactions and other users.

For the transaction the notion of ACID properties was introduced:

Atomic: the entire transaction succeeds or fails as a complete unit;
Consistent: a completed transaction leaves the affected data in a consistent or correct state;
Isolated: each transaction executes in isolation and does not affect the state of the others;
Durable: the changes resulting from committed transactions are persistent.

The mechanism used to enforce transaction isolation is the LOCK. In Oracle the so-called "write lock" or "exclusive lock" is used. It is applied and held while the change is made to data in the course of a transaction and released when the transaction is ended by either a commit or a rollback statement. Only one user can hold a write lock at a time, so only that user can change that data. The locks enforcing isolation between concurrent users of data can lead its own problems. As the locks prevent other transaction from completing, a problem of contention arises. However some basic integrity problems can result if transaction isolation is not properly enforced. The most common problem are:

- lost update: it occurs when two writers are both changing the same piece of data so that one overwrite the change of the other;
- dirty reads: it occurs when a database allows transaction to read the data that has been changed by another transaction but has not been committed yet. If the change are rolled back the data may turn out to be incorrect.
- Nonrepeatable reads: it occurs as a result of change made by another transaction. One transaction makes a query based on a particular condition. After the data has been returned to the first transaction, but before the first transaction is complete, another changes the data so that some of the previously retrieved data no longer satisfies the selection condition. If the query were repeated in the same transaction, it would return a different set of results.
- Phantom reads: it occurs as a result of change made by another transaction. One transaction makes a query based on a particular condition, after data has been returned to the first transaction, but before the first transaction is complete, another transaction inserts into the database new rows that meet the selection criteria for the first transaction. If the SQL statement in a transaction is returned the numbers of rows that initially satisfied the selection criteria, and then performed an action on the rows those rows later, the number of rows affected would be different from the initial numbers of rows indicated, based on the inclusion of the new phantom rows.

The solution offered by Oracle to solve the concurrency problem is to provide the highest level of isolation between the actions of different users accessing the same data, that's serialization.
The different transactions are thus conceived in a series, so that each of them is isolated from any changes that occur to its data from subsequent transactions. The serializable transaction looks as though it has the exclusive use of the database for the duration of the transaction. Serializable transactions are predictable and reproducible, the two cardinal virtues of the data integrity. The exclusive use of the database for simultaneous users is an issue that Oracle solved by means a technology used MultiVersion Read Consistency (MVRC).
If another user changes the underlying data during the query execution, Oracle maintains a version of the data as it existed at the time the query began. The data returned to the query

will reflect all the committed transaction at the time the query started.
An application programmer can set an isolation level at the session level (ALTERSESSION) or transaction level (SET TRANSACTION). The basic isolation levels frequently used by Oracle are: READ COMMITTED, SERIALIZABLE, READONLY. The first two isolation levels create serializable operations:

RAED COMMITTED
Enforces serialization at the statement level. This means that every statement will get a consistent view of the data as it existed at the start of the statement. However, since a transaction can contain more than one statement, it is possible that ronrepeatable reads and phantom reads can occur within the context of the complete transaction. The READ COMMITTED isolation level is the default isolation for Oracle.

SERIALIZABLE
Enforce serialization at the transaction level. This means that every statement within a transaction will get the same consistent view of the data as it existed at the start of the transaction.

READONLY
This level explicitly prohibits any write operations and provides an accurate view of all the data at the time the transaction began.

Three feature are used by Oracle to implement multiversion read consistency:

Rollback segments:
Rollback segments are structure in the Oracle database that store "undo" information for transactions in case of rollback. When a transaction starts changing some data in a block, it first writes the old image of the data to a rollback segment, in order to make available the old image to support the rollback and the multiversion read consistency.
A rollback segment is different from a redo log. The redo log is used to log all transactions to the database and recover the database in the event of the system failure, while the rollback segment provides rollback for the transactions and read consistency. Blocks of rollback segments are cached in the System Global Area just like block of tables and indexes.

System Change Number (SCN)
To preserve the integrity of the data in the database and enforce any type of serialization, it is critical to keep track of the order in which actions were performed. The SCN is a logical timestamp that tracks the order of the transactions occurred and it is used in the redo log to recover the original version when a redo is applied.

Locks in a data block
A database must have a way of determining if a particular row is locked. A data block is the smallest amount of data that can be read from disk, so whenever the row is requested, the block is read, and the lock is available within the block.

Appendix 1

test_PyCoral_NetworkGlitch.py

in coral / Tests / PyCoral_NetworkGlitch /

```
#!/usr/bin/env python
import os, sys, unittest, fcntl
from NetworkTunnel import NetworkTunnel
from time import sleep, time, localtime, strftime
from signal import SIGKILL
import PyCoralTest

#==============================================================================
=

def msghdr( ichild=0 ):
    tnow = time()
    now = strftime("%H:%M:%S",localtime(tnow))+str("%.6f"%(tnow-
int(tnow)))[1:]
    if ichild == 0 :
        MSGHDR = "+++ PYTEST MAIN (%5i) @%s +++"%(os.getpid(),now)
    else:
        MSGHDR = "+++ PYTEST child(%5i) @%s +++"%(os.getpid(),now)
    return MSGHDR

#==============================================================================
=

def createTable():
    ###print msghdr(), "Recreate the table"
    session = svc.connect( tunnelUrl, coral.access_Update )
    session.transaction().start(False)
    session.nominalSchema().dropIfExistsTable( tableName )
    description = coral.TableDescription()
    description.setName( tableName)
    description.insertColumn( 'ID', 'int' )
    description.insertColumn( 'Data', 'float' )
    description.setPrimaryKey( 'ID' )
    tableHandle = session.nominalSchema().createTable( description )
    ###print msghdr(), "Fill the table"
    bulkInserter = tableHandle.dataEditor().bulkInsert( rowBuffer, 100 )
    for i in range(100000):
        rowBuffer["ID"].setData(i)
        rowBuffer["Data"].setData(i)
        bulkInserter.processNextIteration()
    bulkInserter.flush()
    session.transaction().commit()
    sleep(1)

#==============================================================================
=

class TestNetworkGlitch( PyCoralTest.TestCase ):

    #--------------------------------------------------------------------------
-

    def setUp(self):
        # Call the base class method
        PyCoralTest.TestCase.setUp(self)
        # Use CORAL defaults for retrial parameters
        svc.configuration().setConnectionRetrialPeriod( retrialPeriod )
        svc.configuration().setConnectionRetrialTimeOut( retrialTimeOut )
        # Use CORAL defaults for connection sharing (enabled)
        svc.configuration().enableConnectionSharing()
        # Configure the connection service (see bug #71449)
        # - disable the CORAL connection pool cleanup
        # - connection timeout=0: "idle" connections are immediately
"expired"
        svc.configuration().disablePoolAutomaticCleanUp()
        svc.configuration().setConnectionTimeOut(0)
        # Use CORAL defaults for RO transactions (serializable)
        if "CORAL_ORA_SKIP_TRANS_READONLY" in os.environ:
            del os.environ["CORAL_ORA_SKIP_TRANS_READONLY"]
        # Disable the hack to sleep while connecting to cause ORA-24327
        if "CORAL_ORA_TEST_ORA24327_SLEEP10S" in os.environ:
```

```
            del os.environ["CORAL_ORA_TEST_ORA24327_SLEEP10S"]

    #------------------------------------------------------------------------
-

    def tearDown(self):
        # Purge the connection pool after each test
        svc.purgeConnectionPool()
        # Call the base class method
        PyCoralTest.TestCase.tearDown(self)

    #------------------------------------------------------------------------
-

    # Simple test that succeeds (just to test the test infrastructure!)
    def test010_createQueryAfterGlitch(self):
        dbg = True
        if dbg: print ""
        if dbg: print msghdr(), "Connect R/O to:", tunnelUrl
        session = svc.connect( tunnelUrl, coral.access_ReadOnly )
        ###if dbg: print msghdr(), "Session proxy is", session
        if dbg: print msghdr(), "Start a R/O transaction"
        if dbg: print msghdr(), "GLITCH.....!"
        nt.glitch() # glitch when the transaction is not yet active...
        if dbg: print msghdr(), "GLITCH OVER!"
        session.transaction().start(True)
        if dbg: print msghdr(), "Trigger reconnect (retrieve nominal
schema)"
        session.nominalSchema()
        if dbg: print msghdr(), "Create a query"
        query = session.nominalSchema().tableHandle(tableName).newQuery()
        query.addToOrderList( "ID ASC" )
        query.defineOutput( rowBuffer )
        query.setMemoryCacheSize(0)
        query.setRowCacheSize(1)
        cursor = query.execute()
        if dbg: print msghdr(), "Retrieve the first 5 rows"
        counter = 0
        while ( counter < 5 and cursor.next() ):
            print rowBuffer[0], rowBuffer[1]
            counter += 1
        if dbg: print msghdr(), "Commit the R/O transaction"
        session.transaction().commit()

    #------------------------------------------------------------------------
-

    # Test for bug #73334 (presently CRASHES... to be fixed!)
    def _test011_ICursorPVM_bug73334(self): # disabled (crashes!)
        dbg = True
        if dbg: print ""
        if dbg: print msghdr(), "Connect R/O to:", tunnelUrl
        session = svc.connect( tunnelUrl, coral.access_ReadOnly )
        ###if dbg: print msghdr(), "Session proxy is", session
        if dbg: print msghdr(), "Start a R/O transaction"
        session.transaction().start(True)
        if dbg: print msghdr(), "Create a query"
        query = session.nominalSchema().tableHandle(tableName).newQuery()
        query.addToOrderList( "ID ASC" )
        query.defineOutput( rowBuffer )
        query.setMemoryCacheSize(0)
        query.setRowCacheSize(1)
        cursor = query.execute()
        if dbg: print msghdr(), "Retrieve the first 5 rows"
        counter = 0
        while ( counter < 5 and cursor.next() ):
            print rowBuffer[0], rowBuffer[1]
            counter += 1
        if dbg: print msghdr(), "GLITCH.....!"
        nt.glitch()
        if dbg: print msghdr(), "GLITCH OVER!"
        if dbg: print msghdr(), "Trigger reconnect (retrieve nominal
schema)"
        session.nominalSchema()
        if dbg: print msghdr(), "Retrieve the 6th row (this may fail)"
        cursor.next()
        if dbg: print msghdr(), "The 6th row was retrieved: print it"
```

```
        print rowBuffer[0], rowBuffer[1]
        counter += 1
        if dbg: print msghdr(), "Retrieve additional rows (up to 20 in
total)"
        while ( counter<20 and cursor.next() ):
            print rowBuffer[0], rowBuffer[1]
            counter += 1

    #------------------------------------------------------------------------
-

    # Test for bug #73688 aka bug #57639 (presently succeeds)
    def test021_glitchInActiveTransRW_bug73688_akaBug57639(self):
        dbg = True
        if dbg: print ""
        if dbg: print msghdr(), "Connect R/W to:", tunnelUrl
        session = svc.connect( tunnelUrl, coral.access_Update )
        if dbg: print msghdr(), "Start a R/W transaction"
        session.transaction().start(False)
        if dbg: print msghdr(), "GLITCH.....!"
        nt.glitch()
        if dbg: print msghdr(), "GLITCH OVER!"
        if dbg: print msghdr(), "Try to commit... it should fail"
        try:
            session.transaction().commit()
            self.fail( "Commit should fail for RW transaction after glitch!"
)
        except coral.Exception, e:
            if dbg: print msghdr(), "Commit has failed: OK"

    #------------------------------------------------------------------------
-

    # Test for bug #57117 (presently fails... to be fixed)
    def _test022_glitchInActiveTransSerialRO_bug57117(self):
        if "LCG_NGT_SLT_NUM" in os.environ: return # disable in nightlies
        dbg = True
        if dbg: print ""
        if dbg: print msghdr(), "Connect R/O to:", tunnelUrl
        session = svc.connect( tunnelUrl, coral.access_ReadOnly )
        if dbg: print msghdr(), "Start a R/O transaction"
        session.transaction().start(True)
        if dbg: print msghdr(), "GLITCH.....!"
        nt.glitch()
        if dbg: print msghdr(), "GLITCH OVER!"
        if dbg: print msghdr(), "Try to commit... it should fail"
        session.transaction().commit()
        try:
            session.transaction().commit()
            self.fail( "Commit should fail for serializable RO transaction
after glitch!" )
        except coral.Exception, e:
            if dbg: print msghdr(), "Commit has failed: OK"

    #------------------------------------------------------------------------
-

    # Test for bug #65597 (presently succeeds... should check it used to
fail!)
    def test023_glitchInActiveTransNonSerialRO_bug65597(self):
        os.environ["CORAL_ORA_SKIP_TRANS_READONLY"] = "1"
        dbg = True
        if dbg: print ""
        if dbg: print msghdr(), "Connect R/O to:", tunnelUrl
        session = svc.connect( tunnelUrl, coral.access_ReadOnly )
        if dbg: print msghdr(), "Start a R/O transaction"
        session.transaction().start(True)
        if dbg: print msghdr(), "GLITCH.....!"
        nt.glitch()
        if dbg: print msghdr(), "GLITCH OVER!"
        if dbg: print msghdr(), "Try to commit... it should succeed"
        session.transaction().commit()
        if dbg: print msghdr(), "Commit has succeeded: OK"

    #------------------------------------------------------------------------
-
```

```
    # Test ... (presently succeeds)
    def test031_IConnectionToCursor(self):
        self.internalTest03x_IConnectionToCursor(1)

    # Test ... (presently fails)
    def _test032_IConnectionToCursor(self):
        if "LCG_NGT_SLT_NUM" in os.environ: return # disable in nightlies
        self.internalTest03x_IConnectionToCursor(2)

    # Test ... (presently fails)
    def _test033_IConnectionToCursor(self):
        if "LCG_NGT_SLT_NUM" in os.environ: return # disable in nightlies
        self.internalTest03x_IConnectionToCursor(3)

    # Test ... (presently fails)
    def _test034_IConnectionToCursor(self):
        if "LCG_NGT_SLT_NUM" in os.environ: return # disable in nightlies
        self.internalTest03x_IConnectionToCursor(4)

    # Test ... (presently fails)
    def _test035_IConnectionToCursor(self):
        if "LCG_NGT_SLT_NUM" in os.environ: return # disable in nightlies
        self.internalTest03x_IConnectionToCursor(5)

    def internalTest03x_IConnectionToCursor(self, i):
        dbg = True
        if dbg: print ""
        if dbg: print msghdr(), "Open a network tunnel"
        nt.open()
        if dbg: print msghdr(), "Connect R/O to:", tunnelUrl
        session = svc.connect(tunnelUrl, coral.access_Update )
        if i == 1:
            if dbg: print msghdr(), "GLITCH.....!"
            nt.glitch()
            if dbg: print msghdr(), "GLITCH OVER!"
            if dbg: print msghdr(), "Glitch before transaction
instantiation"
        transaction = session.transaction()
        if i == 2:
            if dbg: print msghdr(), "GLITCH.....!"
            nt.glitch()
            if dbg: print msghdr(), "GLITCH OVER!"
            if dbg: print msghdr(), "Glitch before transaction start"
        transaction.start()             # AV 'will not reconnect'? (have a
look)
        #session.transaction().start() # AV 'will reconnect'? (have a look)
        if i == 3:
            if dbg: print msghdr(), "GLITCH.....!"
            nt.glitch()
            if dbg: print msghdr(), "GLITCH OVER!"
            if dbg: print msghdr(), "Glitch before Nominal Schema"
        schema = session.nominalSchema()
        if i == 4:
            if dbg: print msghdr(), "GLITCH.....!"
            nt.glitch()
            if dbg: print msghdr(), "GLITCH OVER!"
            if dbg: print msghdr(), "Glitch before Query"
        query = schema.tableHandle(tableName).newQuery()
        if i == 4:
            if dbg: print msghdr(), "GLITCH.....!"
            nt.glitch()
            if dbg: print msghdr(), "GLITCH OVER!"
            if dbg: print msghdr(), "GLITCH before Query execution"
        query.defineOutput( rowBuffer )
        counter = 0
        cursor = query.execute()

        while ( cursor.next() ):
            if counter == 21: break
            print rowBuffer[0], rowBuffer[1]
            if counter == 10 and i == 5:
                if dbg: print msghdr(), "GLITCH.....!"
                nt.glitch()
                if dbg: print msghdr(), "GLITCH OVER!"
                if dbg: print msghdr(), "GLITCH during the loop on the
Cursor"
            counter += 1
```

```
    #------------------------------------------------------------------------
-

    # Test for ORA-24327 (presently fails)
    # This is bug #58522 (aka bug #65709, bug #75596)
    def test040_glitchInConnect_ora24327_bug58522(self):
        os.environ["CORAL_ORA_TEST_ORA24327_SLEEP10S"] = "1"
        svc.configuration().disableConnectionSharing() # force new
connection!
        svc.configuration().setConnectionRetrialPeriod( 5 )
        newTimeOut = 15
        svc.configuration().setConnectionRetrialTimeOut( newTimeOut )
        dbg = True # needed?
        print ""
        print msghdr(), "Fork and delay dump to resync parent and child"
        # See http://stackoverflow.com/questions/871447
        rpipe, wpipe = os.pipe()
        child_pid = os.fork()
        # Parent process: connect (allow a 10s window for the glitch)
        if child_pid != 0:
            os.close(wpipe)
            rtmp, wtmp = os.pipe()
            # See
http://www.parallelpython.com/component/option,com_smf/Itemid,29/topic,414.m
sg1225#msg1225
            fd1 = sys.__stdout__.fileno() # This fd is normally 1
            savestdout = os.dup(fd1)
            os.dup2(wtmp,fd1) # Redirect stdout to tmp pipe
            if dbg: print msghdr(), "Parent process: PID=%s" % os.getpid()
            if dbg: print msghdr(), "Connect R/O to:", tunnelUrl
            time0 = time()
            connectfailed = None
            connected = None
            try:
                session = svc.connect( tunnelUrl, coral.access_ReadOnly )
                # Internal error (e.g. sleep and/or glitch did not work)
                self.fail( "Connection succeeded? It should fail!" )
            except coral.Exception, connectfailed:
                delta = time() - time0
                print msghdr(), \
                      "Exception caught after %.3f seconds"%delta
                print msghdr(), \
                      "Sleep + Retrial timeout is %.3f
seconds"%(10+newTimeOut)
                # The test succeeds if CORAL immediately throws
                if delta > 10+newTimeOut:
                    print msghdr(), \
                          "CORAL waited for the retrial timeout: FAILURE"
                else:
                    print msghdr(), \
                          "CORAL did not wait for the retrial timeout:
SUCCESS"
                    connectfailed = None
            except Exception, connected:
                pass
            os.dup2(savestdout,fd1) # Restore stdout
            os.close(wtmp)
            # Cleanup (before rethrowing an exception if necessary)
            try: os.kill(child_pid,SIGKILL) # Terminate subprocess if any
            except Exception, nosubprocess: pass # No subprocess
            fcntl.fcntl(rpipe, fcntl.F_SETFL, os.O_NONBLOCK)
            rpipe = os.fdopen(rpipe,'r',0)
            fcntl.fcntl(rtmp, fcntl.F_SETFL, os.O_NONBLOCK)
            rtmp = os.fdopen(rtmp,'r',0)
            cline = None  # signal that child line must be read
            pline = None  # signal that parent line must be read
            pfirst = True # print parent line first by default
            while True:
                if cline is None:
                    try: cline = rpipe.readline()
                    except: cline = "" # This pipe has been closed
                if pline is None:
                    try: pline = rtmp.readline()
                    except: pline = "" # This pipe has been closed
                if cline == "" and pline == "":
                    break # Both pipes have been closed
```

```
                elif cline == "":
                    pfirst = True
                elif pline == "":
                    pfirst = False
                else:
                    cat = cline.find('@')
                    pat = pline.find('@')
                    if cat >=0 and pat >=0:
                        pfirst = ( pline[pat+1:pat+16] < cline[cat+1:cat+16]
)
                    else:
                        pass # reuse last order (no line time, reuse last
time)
                if pfirst:
                    sys.stdout.write(pline)
                    pline = None # signal that it must be read again
                else:
                    sys.stdout.write(cline)
                    cline = None # signal that it must be read again
            rpipe.close()
            rtmp.close()
            # Rethrow an exception if necessary
            print msghdr(), "Delayed dump completed"
            if connectfailed: raise connectfailed
            if connected: raise connected
        # Child subprocess: trigger a glitch after 2s and exit
        else:
            os.close(rpipe)
            sleep(1)
            wpipe = os.fdopen(wpipe,'w',0)
            sys.stdout = wpipe # Redirect stdout to pipe
            print msghdr(1), " Child process: PID=%s" % os.getpid()
            sleep(2)
            print msghdr(1), " GLITCH.....!"
            nt.glitch()
            print msghdr(1), " GLITCH OVER!"
            print msghdr(1), " Child process: exit!"
            sys.stdout = sys.__stdout__ # Restore stdout
            wpipe.close()
            os.kill(os.getpid(),SIGKILL) # Exit

#============================================================================
=

if __name__ == '__main__':

    print "-"*70

    # Build the unique table name and port number
    lport = 45000
    tableName = "NETWORKGLITCHTEST"
    import PyCoralTest
    tableName = PyCoralTest.buildUniqueTableName( tableName )
    lport = PyCoralTest.buildUniquePortNumber()
    print msghdr(), "Table name:", tableName
    print msghdr(), "Local port number:", lport

    # Create the appropriate tnsnames.ora
    # [NB 'lcg_coral_nightly_proxy' is in
$CORAL_AUTH_PATH/authentication.xml]
    tnsalias = "lcg_coral_nightly_proxy"
    tunnelUrl = "oracle://"+ tnsalias + "/lcg_coral_nightly"
    os.environ['CORAL_AUTH_PATH']='/afs/cern.ch/sw/lcg/app/pool/db'
    tnsdir = "/tmp/" + os.environ['USER']
    if "CMTCONFIG" in os.environ:
        tnsdir += "/" + os.environ["CMTCONFIG"]
    if "LCG_NGT_SLT_NAME" in os.environ:
        tnsdir += "/" + os.environ["LCG_NGT_SLT_NAME"]
    os.system( "\\mkdir -p " + tnsdir )
    tnsfile = tnsdir + "/tnsnames.ora"
    print msghdr(), "TNS_ADMIN:", tnsdir
    os.environ['TNS_ADMIN'] = tnsdir
    tnsoutput =
tnsalias+"=(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)(HOST=localhost
)(PORT="+str(lport)+")))(CONNECT_DATA=(SERVICE_NAME=test1.cern.ch)(INSTANCE_
NAME=test11)(SERVER= DEDICATED)))"
    fd = open(tnsfile, 'w')
```

```
    fd.write(tnsoutput)
    fd.close()

    # Bootstrap CORAL
    ###os.environ['CORAL_MSGLEVEL']='Verbose'
    ###os.environ['CORAL_MSGLEVEL']='Info'
    os.environ['CORAL_MSGLEVEL']='Warning'
    import coral
    rowBuffer = coral.AttributeList()
    rowBuffer.extend("ID","int")
    rowBuffer.extend("Data","float")
    print msghdr(), "Instantiate the PyCoral connection service"
    svc = coral.ConnectionService()

    # Save CORAL default retrial parameters as global variables
    retrialPeriod=svc.configuration().connectionRetrialPeriod()
    retrialTimeOut=svc.configuration().connectionRetrialTimeOut()

    # Open the network tunnel
    print msghdr(), "Open a network tunnel"
    nt = NetworkTunnel(lport)
    ###nt = NetworkTunnel(lport,debug=True)
    nt.open()

    # Recreate the table?
    doCreateTable = False # default for interactive tests
    if "LCG_NGT_SLT_NUM" in os.environ:
        doCreateTable = True # default for nightly tests
    if len(sys.argv) > 1 and sys.argv[1] == "-createTable":
        doCreateTable = True # undocumented (argv[1] can also be a test)
        sys.argv.pop(1)
    if doCreateTable:
        print msghdr(), "Create the table"
        createTable()
    else:
        print msghdr(), "Do not create the table"

    # Start the unit test (can specify one specific test as cl argument)
    unittest.main( testRunner =
                   unittest.TextTestRunner(stream=sys.stdout,verbosity=2) )
```

NetworkTunnel.py

in coral / Tests / PyCoral_NetworkGlitch

```
#!/bin/env python
import os, sys
from subprocess import Popen, PIPE, STDOUT
from time import sleep
import socket

#------------------------------------------------------------------------------

sshcmd0 = "/usr/bin/ssh"
sshcmd1 = "/tmp/" + os.environ['USER'] + "/sshTunnel"

# Local host (OCI client and tunnel gateway), target host (Oracle server)
ghost = socket.gethostname()
tport = "itrac507-v.cern.ch:10121"

class NetworkTunnel:

    def __init__( self, lport, debug=False, kill=True ):
        self.lport = lport
        self.debug = debug
        self.sshcmd = sshcmd1 + str(lport)
        if kill: self.killall()
        os.system( "\\rm " + self.sshcmd + " > /dev/null 2>&1" )
        os.system( "\\ln -sf " + sshcmd0 + " " + self.sshcmd )

    def __del__( self ):
        self.killall() # Fix bug #73763 (avoid hang in sshTunnel bkg
process)
```

```
    def dbg( self, msg ):
        if self.debug: print "NetworkTunnel (" + str(os.getpid()) + ") " +
msg

    # Kill any open sshTunnel processes for $USER
    def killall( self ):
        pids = self.ps( debug=0 )
        if ( len( pids ) == 0 ) : return
        self.dbg( "Kill all open network tunnels" )
        for pid in pids:
            print "ssh("+str(pids[0])+") Process will be killed"
            os.system( "kill -KILL " + pid )
        sleep(1)
        pids = self.ps( debug=0 )
        if ( len( pids ) != 0 ) :
            sys.stderr.write( "FATAL ERROR! Could not kill network
tunnels\n" )
            for pid in pids: sys.stderr.write( "PID: " + str(pid) + "\n" )
            sys.exit(1)

    # Check if an sshTunnel process is running for $USER and return its pid
    # Return 0 if no sshTunnel process is running for $USER
    # Return -1 if more than one sshTunnel processes are running for $USER
    def ps( self, debug=1 ):
        if debug == 1: self.dbg( "Check if any network tunnels are open" )
        cmd = "ps -ef | grep " + os.environ['USER'] + " | grep " +
self.sshcmd + " | grep -v grep"
        if debug == 1: self.dbg( "Execute: '" + cmd + "'" )
        processes = [ item[:-1] for item in Popen( cmd, shell=True,
stdout=PIPE ).stdout ]
        pids = []
        if len( processes ) == 0:
            if debug == 1: self.dbg( "No network tunnel is open" )
        elif len( processes ) == 1:
            pid = processes[0].split(' ',1)[1].lstrip(' ').split(' ',1)[0]
            pids.append(pid)
            if debug == 1: self.dbg( "One network tunnel is open: pid="+pid)
        else:
            for process in processes:
                pid = process.split(' ',1)[1].lstrip(' ').split(' ',1)[0]
                pids.append(pid)
            if debug == 1: self.dbg( "WARNING! Several network tunnels are
open: pid = "+str(pids) )
        for process in processes:
            if debug == 1 and self.debug: print process
        return pids

    def open(self):
        pids = self.ps( debug=0 )
        if ( len( pids ) != 0 ) :
            self.dbg( "ERROR! Cannot open a new network tunnel: one or more
are already open" )
            return
        self.dbg( "Open a new network tunnel from localhost:" +
str(self.lport) + " to " + tport + " through " + ghost )
        # See http://linuxcommando.blogspot.com/2008/10/how-to-disable-ssh-
host-key-checking.html
        cmd = self.sshcmd + " -o UserKnownHostsFile=/dev/null -o
StrictHostKeyChecking=no -N -L " + str(self.lport) + ":" + tport + " " +
ghost + " 2>&1 | grep ssh" # Output/Error to stdout
        ###cmd = self.sshcmd + " -N -L " + str(self.lport) + ":" + tport + "
" + ghost + " 2>&1 | grep ssh" # Output/Error to stdout
        self.dbg( "Execute: '" + cmd + "'" )
        Popen( cmd, shell=True )
        ###cmd = self.sshcmd + " -N -L " + str(self.lport) + ":" + tport + "
" + ghost + " 2>&1 | grep ssh &" # Output/Error to stdout
        ###cmd = self.sshcmd + " -N -L " + str(self.lport) + ":" + tport + "
" + ghost + " -f 2>&1 | grep ssh" # Output/Error to stdout
        ###self.dbg( "Execute: '" + cmd + "'" )
        ###os.system( cmd )
        self.dbg( "Command executed" )
        pids = self.ps( debug=0 )
        if ( len( pids ) != 1 ) :
            print "WARNING! Could not open network tunnel (yet?)"
            print "WARNING! ps returns",len(pids),"tunnels (one expected):"
            print pids
```

```
            print "WARNING! Sleep 3s and run ps again"
            sleep(3) # Attempt a workaround for bug #76917
            pids = self.ps( debug=0 )
        if ( len( pids ) != 1 ) :
            print "FATAL ERROR! Could not open network tunnel"
            print "FATAL ERROR! ps returns",len(pids),"tunnels (one
expected):"
            print pids
            print "FATAL ERROR! print error on stderr and exit"
            sys.stderr.write( "FATAL ERROR! Could not open network tunnel\n"
)
            sys.exit(1)
        print "ssh("+str(pids[0])+") Process started"
        self.dbg( "Sleep 1s" )
        sleep(1) # Else may get ORA-12541 on first connection attempt

    def close(self):
        self.dbg( "Close network tunnel" )
        pids = self.ps( debug=0 )
        if ( len( pids ) == 0 ) :
            self.dbg( "ERROR! Cannot close the network tunnel: none is open"
)
            return
        elif ( len( pids )>1 ) :
            self.dbg( "WARNING! Will close more than one network tunnels" )
        self.killall()

    def glitch(self):
        self.dbg( "*** GLITCH START ***" )
        self.close()
        sleep(1)
        self.open()
        self.dbg( "*** GLITCH END   ***" )

#-------------------------------------------------------------------------------
--

if __name__ == '__main__':
    print "__main__ Start"
    print "__main__ Instantiate NetworkTunnel manager"
    ###nt = NetworkTunnel(45000)
    nt = NetworkTunnel(45000,debug=True)
    print "__main__ Check processes"
    nt.ps()
    print "__main__ Open tunnel"
    nt.open()
    print "__main__ Glitch tunnel"
    nt.glitch()
    print "__main__ Check processes"
    nt.ps()
    ###print "__main__ Open tunnel"
    ###nt.open() # WILL FAIL (already open)
    print "__main__ Close tunnel"
    nt.close()
    ###print "__main__ Close tunnel"
    ###nt.close() # WILL FAIL (already closed)
    print "__main__ Delete NetworkTunnel manager"
    nt = 0
    print "__main__ Exit"
```

Appendix 2

Session.h

in coral/OracleAccess/src

```
#ifndef ORACLEACCESS_SESSION_H
#define ORACLEACCESS_SESSION_H

#include "RelationalAccess/AccessMode.h"
#include "CoralCommon/IDevSession.h"
#include <map>
#include "CoralBase/boost_thread_headers.h"
#include "CoralBase/MessageStream.h"
#include "QueryMgr.h"

struct OCIServer;

namespace coral {

  namespace OracleAccess {

    class ConnectionProperties;
    class SessionProperties;
    class MonitorController;
    class Transaction;
    class Schema;

    /**
     * Class Session
     *
     * Implementation of the ISession interface for the OracleAccess module
     */
    class Session : virtual public coral::IDevSession
    {
    public:
      /// Constructor
      Session( coral::IDevConnection& connection,
               ConnectionProperties& connectionProperties,
               const std::string& schemaName,
               coral::AccessMode mode );

      /// Destructor
      ~Session();

      /**
       * Returns the reference to the underlying IMonitoring object.
       */
      coral::IMonitoring& monitoring();

      /**
       * Authenticates with the database server using a user/password pair.
       * If the authentication fails an AuthenticationFailureException is
thrown.
       */
      void startUserSession( const std::string& userName,
                             const std::string& password );

      /**
       * Terminates a user session without dropping the connection to the
database server
       */
      void endUserSession();

      /**
       * Returns the status of a user session.
       */
      bool isUserSessionActive() const;

      /**
```

```
       * Returns the corresponding ITransaction object.
       * If a connection is not yet established, a
ConnectionNotActiveException is thrown.
       */
      coral::ITransaction& transaction();

      /**
       * Returns a reference to the working ISchema object.
       * If a connection is not yet established, a
ConnectionNotActiveException is thrown.
       */
      coral::ISchema& nominalSchema();

      /**
       * Returns a reference to the ISchema object corresponding to the
specified name.
       * If a connection is not yet established, a
ConnectionNotActiveException is thrown.
       * If no schema exists corresponding to the specified name an
InvalidSchemaNameException is thrown.
       */
      coral::ISchema& schema( const std::string& schemaName );

      /**
       * Returns the technology name for the remote session.
       * For plugins establishing a database connection through a middle-
tier
       * (e.g. CoralAccess), this is discovered when establishing the remote
session.
       */
      std::string remoteTechnologyName() const
      {
        return "Oracle"; // same as Domain.flavorName()
      }

    private:

      void restartSession();

      void checkConnection();

    private:

      /// The properties of the session
      SessionProperties* m_properties;

      /// The monitoring controller
      MonitorController* m_monitorController;

      /// The schema object
      Schema*            m_schema;

      /// The transaction object
      Transaction*       m_transaction;

      /// Mutex for the handles
      mutable boost::mutex m_mutex;

      /// Mutex for the schemas
      mutable boost::mutex m_mutexForSchemas;

      /// A map of the other schemas
      std::map< std::string, Schema* > m_schemas;

      /// SharedPointer to store the transaction status information
      TransactionStatusInstance m_tsi;


    };

  }

}

#endif
```

Session.cpp

in coral/OracleAccess/src

```
#ifdef WIN32
#include <WTypes.h>
#endif

#include "Session.h"
#include "SessionProperties.h"
#include "ConnectionProperties.h"
#include "DomainProperties.h"
#include "MonitorController.h"
#include "OracleErrorHandler.h"
#include "Transaction.h"
#include "Schema.h"
#include "OracleStatement.h"

#include "RelationalAccess/SessionException.h"
#include "RelationalAccess/IMonitoringService.h"

#include "CoralCommon/MonitoringEventDescription.h"
#include "CoralCommon/IDevConnection.h"

#include "CoralBase/AttributeList.h"

#include "CoralKernel/Service.h"
#include "CoralKernel/Context.h"

#include "oci.h"

#include <cstring> // fix bug #58581
#include <locale>
#include <iostream>

coral::OracleAccess::Session::Session(  coral::IDevConnection& connection,

coral::OracleAccess::ConnectionProperties& connectionProperties,
                                        const std::string& _schemaName,
                                        coral::AccessMode mode )
: coral::IDevSession( connection )
, m_schema( 0 )
, m_transaction( 0 )
, m_mutex()
, m_mutexForSchemas()
, m_schemas()
, m_tsi( new TransactionStatus )
{
  // Convert the schema name to uppercase
  std::string schemaName = _schemaName;
  for ( std::string::size_type i = 0; i < schemaName.size(); ++i ) {
    schemaName[i] = std::toupper( schemaName[i], std::locale::classic() );
  }
  m_properties = new coral::OracleAccess::SessionProperties(
connectionProperties, schemaName, static_cast<coral::ISession&>( *this ),
mode == coral::ReadOnly);
  m_monitorController = new coral::OracleAccess::MonitorController(
*m_properties );
}


coral::OracleAccess::Session::~Session()
{
  if ( this->isUserSessionActive() ) this->endUserSession();
  delete m_monitorController;
  delete m_properties;
}


coral::IMonitoring&
coral::OracleAccess::Session::monitoring()
{
  return *m_monitorController;
}
```

```
void
coral::OracleAccess::Session::startUserSession( const std::string& userName,
                                                const std::string& password
)
{
  if ( ! this->isSessionValid() ) return;

  boost::mutex::scoped_lock lock( m_mutex );

  m_properties->startSession(userName, password);

// Create new schema and transaction objects.
  m_schema = new coral::OracleAccess::Schema( *m_properties, m_properties-
>schemaName(), m_tsi );

  bool skipTransRO = ( ::getenv( "CORAL_ORA_SKIP_TRANS_READONLY" ) != 0 );
  {
    coral::MessageStream log( m_properties->domainProperties().service()-
>name() );
    if ( skipTransRO )
      log << coral::Warning << "CORAL_ORA_SKIP_TRANS_READONLY is set: this
session will skip OCI serializable read-only transactions" <<
coral::MessageStream::endmsg;
    else
      log << coral::Warning << "CORAL_ORA_SKIP_TRANS_READONLY is not set:
this session will use OCI serializable read-only transactions" <<
coral::MessageStream::endmsg;
  }

  m_transaction = new coral::OracleAccess::Transaction( *m_properties,
*m_schema, m_tsi, !skipTransRO );
  // Record the beginning of the session
  if ( m_properties->monitoringService() ) {
    m_properties->monitoringService()->record( "oracle://" + m_properties-
>connectionString() + "/" + m_properties->schemaName(),
                                               coral::monitor::Session,
                                               coral::monitor::Info,

monitoringEventDescription.sessionBegin() );
  }
}

bool
coral::OracleAccess::Session::isUserSessionActive() const
{
  if ( ! this->isSessionValid() ) return false;
  boost::mutex::scoped_lock lock( m_mutex );
  return ( m_properties->getSessionHandle() != 0 );
}



void
coral::OracleAccess::Session::endUserSession()
{
  if ( ! this->isUserSessionActive() ) return;

  // Abort any active transaction
  if ( this->transaction().isActive() ) this->transaction().rollback();

  boost::mutex::scoped_lock lock( m_mutex );

  // Stop the tracing
  static const char* cerntrace_on = ::getenv( "CORAL_ORA_CERN_TRACE_ON" );
  if ( cerntrace_on ) {
    this->transaction().start();
    static const std::string sqlStatement = "BEGIN CERN_TRACE.CSTOP_TRACE('"
+ std::string( cerntrace_on ) + "'); END;";
    coral::OracleAccess::OracleStatement statement( *m_properties,
m_properties->schemaName(), sqlStatement );
    if ( ! statement.execute( coral::AttributeList() ) ) {
      coral::MessageStream log( m_properties-
>connectionProperties().domainServiceName() );
      log << coral::Warning << "Could not enable SQL TRACE" <<
coral::MessageStream::endmsg;
    }
```

```
    this->transaction().commit();
  }


  if ( m_transaction ) {
    delete m_transaction;
    m_transaction = 0;
  }

  boost::mutex::scoped_lock lockS( m_mutexForSchemas );
  for ( std::map< std::string, coral::OracleAccess::Schema* >::iterator
iSchema = m_schemas.begin();
        iSchema != m_schemas.end(); ++iSchema ) {
    delete iSchema->second;
  }
  m_schemas.clear();
  if ( m_schema ) {
    delete m_schema;
    m_schema = 0;
  }

  // Debug
  {
    coral::MessageStream log( m_properties-
>connectionProperties().domainServiceName() );
    log << coral::Verbose << "End user session with OCISessionEnd"
        << " (OCISession*=" << m_properties->getSessionHandle()
        << ", OCISvcCtx*=" << m_properties->serviceContextHandle()
        << ") on connection (OCIServer*=" << m_properties->serverHandle() <<
")"
        << coral::MessageStream::endmsg;
  }

  // Close the user session
  OCISessionEnd( m_properties->serviceContextHandle(),
                 m_properties->errorHandle(),
                 m_properties->getSessionHandle(), OCI_DEFAULT );

  OCIHandleFree( m_properties->getSessionHandle(), OCI_HTYPE_SESSION );

//AK: TODO: implement a stop session method in SessionProperties
/*

  m_sessionHandle = 0;
*/
  m_properties->unsetSessionHandle();

  // Record the ending of the session
  if ( m_properties->monitoringService() ) {
    m_properties->monitoringService()->record( "oracle://" + m_properties-
>connectionString() + "/" + m_properties->schemaName(),
                                               coral::monitor::Session,
                                               coral::monitor::Info,

monitoringEventDescription.sessionEnd() );
  }

  OCISvcCtx* serviceContextHandle = m_properties->serviceContextHandle();
  OCIHandleFree( serviceContextHandle, OCI_HTYPE_SVCCTX );

  m_properties->setServiceContextHandle( 0 );
}


coral::ITransaction&
coral::OracleAccess::Session::transaction()
{
  // Check if the session is still valid
  checkConnection();

  if ( ! this->isUserSessionActive() )
    throw coral::ConnectionNotActiveException( m_properties-
>domainProperties().service()->name(),
                                               "ISession::transaction" );

  boost::mutex::scoped_lock lock( m_mutex );
  return *m_transaction;
```

```
}


coral::ISchema&
coral::OracleAccess::Session::nominalSchema()
{
  // Check if the session is still valid
  checkConnection();

  if ( ! this->isUserSessionActive() )
    throw coral::ConnectionNotActiveException( m_properties-
>domainProperties().service()->name() );

  boost::mutex::scoped_lock lock( m_mutex );
  return *m_schema;
}


coral::ISchema&
coral::OracleAccess::Session::schema( const std::string& _schemaName )
{
  // Check if the session is still valid
  checkConnection();

  if ( ! this->isUserSessionActive() )
    throw coral::ConnectionNotActiveException( m_properties-
>domainProperties().service()->name() );

  // Convert the schema name to uppercase
  std::string schemaName = _schemaName;
  for ( std::string::size_type i = 0; i < schemaName.size(); ++i ) {
    schemaName[i] = std::toupper( schemaName[i], std::locale::classic() );
  }

  // If this is the nominal schema then return it (fix bug #73530)
  if ( schemaName == m_properties->schemaName() ) return *m_schema;

  // Check first is the schema with the corresponding name exists in the map
  boost::mutex::scoped_lock lock( m_mutexForSchemas );
  std::map< std::string, coral::OracleAccess::Schema* >::iterator iSchema =
    m_schemas.find( schemaName );
  if ( iSchema != m_schemas.end() ) return *( iSchema->second );

  // Check in the database if a schema with such a name exists
  void* temporaryPointer = 0;
  sword status = OCIHandleAlloc( m_properties->environmentHandle(),
                                 &temporaryPointer,
                                 OCI_HTYPE_DESCRIBE, 0, 0 );
  if ( status != OCI_SUCCESS ) {
    throw coral::SessionException( m_properties-
>domainProperties().service()->name(),
                                   "Could not allocate a describe handle",
                                   "ISession::schema" );
  }
  OCIDescribe* describeHandle = static_cast<OCIDescribe*>(temporaryPointer);

  OCIDescribeAny( m_properties->serviceContextHandle(),
                  m_properties->errorHandle(),
                  const_cast<char *>( schemaName.c_str() ),
                  ::strlen( schemaName.c_str() ),
                  OCI_OTYPE_NAME,
                  OCI_DEFAULT,
                  OCI_PTYPE_SCHEMA,
                  describeHandle );
  coral::OracleAccess::OracleErrorHandler errorHandler( m_properties-
>errorHandle() );
  errorHandler.handleCase( status, "retrieving the describe handle of schema
" + schemaName );
  OCIHandleFree( describeHandle, OCI_HTYPE_DESCRIBE );
  if ( errorHandler.lastErrorCode() == 4043 )
  {
    coral::MessageStream log( m_properties->domainProperties().service()-
>name() );
    log << coral::Debug << errorHandler.message() <<
coral::MessageStream::endmsg;
    throw coral::InvalidSchemaNameException( m_properties-
>domainProperties().service()->name() );
```

```
  }

  // The schema exists. Insert it into the map of the known ones
  coral::OracleAccess::Schema* schema = new coral::OracleAccess::Schema(
*m_properties, schemaName, m_tsi );

  m_schemas.insert( std::make_pair( schemaName, schema ) );

  return *schema;
}

void
coral::OracleAccess::Session::restartSession()
{
  // Allocate new session handles
  m_properties->restartSession();
  // Do we have a valid transaction
  // If yes restart the transaction
  if( m_properties->isTransactionActive() )
  {
    // for R/W transactions throw an exception (fix bug #57639 aka bug
#73688)
    if( !m_transaction->isReadOnly() )
    {
      /// FIXME use special exception here
      throw coral::Exception( "R/W Transaction was lost in reconnection
process",
                              "SessionProperties::restartSession",
                              m_properties->domainProperties().service()-
>name() );
    }
    // for serializable transactions throw an exception
    if( m_transaction->isSerializable() )
    {
      /// FIXME use special exception here
      throw coral::Exception( "Serializable RO Transaction was lost in
reconnection process",
                              "SessionProperties::restartSession",
                              m_properties->domainProperties().service()-
>name() );
    }
    // Allocate new transaction handles
    m_transaction->start(true);
  }
}

void
coral::OracleAccess::Session::checkConnection()
{
  if ( m_properties->wasConnectionLost() )
  {
    m_connection.invalidateAll();

    if(m_properties->restartConnectionWithTimeOut())
    {
      restartSession();
    }
  }
  else if(m_properties->wasSessionLost())
  {
    restartSession();
  }
}
```

SessionProperties.h

in coral/OracleAccess/src

```
#ifndef ORACLE_SESSION_PROPERTIES_H
#define ORACLE_SESSION_PROPERTIES_H

#include "CoralBase/boost_thread_headers.h"
#include "ConnectionProperties.h"
```

```
#include "ISessionProperties.h"

struct OCIEnv;
struct OCIError;
struct OCIServer;
struct OCISvcCtx;
struct OCISession;

namespace coral
{

  class ISchema;
  //class ISession;

  namespace OracleAccess
  {

    /**
     * Class SessionProperties
     *
     * An implementation of the ISessionProperties interface.
     */

    class SessionProperties : virtual public ISessionProperties
    {

    public:

      /// Constructor
      SessionProperties( ConnectionProperties& connectionProperties,
                         const std::string& schemaName,
                         coral::ISession& session,
                         bool readOnly );

      /// Destructor
      virtual ~SessionProperties();

      /// Sets the service context handle
      void setServiceContextHandle( OCISvcCtx* serviceContextHandle );

      /// Sets the monitoring service
      void setMonitoringService( coral::monitor::IMonitoringService*
monitoringService );

      /// Returns the connection properties
      const ConnectionProperties& connectionProperties() const;

      /// Returns the domain properties
      const DomainProperties& domainProperties() const;

      /// Returns the connection string
      std::string connectionString() const;

      /// Returns the session object
      coral::ISession& session() const;

      /// Returns the type converter
      coral::ITypeConverter& typeConverter();
      const coral::ITypeConverter& typeConverter() const;

      /// Returns the OCI environment handle
      OCIEnv* environmentHandle() const;

      /// Returns the OCI error handle
      OCIError* errorHandle() const;

      /// Returns the OCI server handle
      OCIServer* serverHandle() const;

      /// Returns the OCI service context handle
      OCISvcCtx* serviceContextHandle() const;

      /// Returns the monitoring service
      coral::monitor::IMonitoringService* monitoringService() const;

      /// Returns the transaction state
      bool isTransactionActive() const;
```

```
  /// Returns the transaction mode
  bool isTransactionReadOnly() const;

  /// Returns the server version
  int serverVersion() const;

  /// Returns the corresponding schema
  coral::ISchema& schema() const;

  /// Returns the schema name
  std::string schemaName() const;

  /// Returns the readOnly flag
  bool isReadOnly() const;

  /// Can this session 'select any table'?
  /// Initially it is assumed that it can.
  bool selectAnyTable() const;

  /// Set the (mutable) 'select any table' flag to false.
  void cannotSelectAnyTable() const;


  /// NetworkGlitch changes ...


  /// NetworkGlitch: Check the validity of the connection
  bool wasConnectionLost();

  /// NetworkGlitch: Check the validity of the session
  bool wasSessionLost();

  /// NetworkGlitch: Reconnect
  bool restartConnection();

  /// NetworkGlitch: Reconnect with time out
  /// TODO: (merge with restartConnection with timeout parameter)
  bool restartConnectionWithTimeOut();

  /// NetworkGlitch: Restart user session
  void restartSession();

  /// NetworkGlitch: Start session
  void startSession( const std::string& userName,
                     const std::string& password )
  {
    m_userName = userName;
    m_password = password;
    // Allocate all OCI handles
    restartSession();
  }

  /// NetworkGlitch: Session handle
  OCISession* getSessionHandle() const
  {
    return m_sessionHandle;
  }

  void unsetSessionHandle()
  {
    m_sessionHandle = 0;
  }

private:

  /// The connection properties
  const ConnectionProperties& m_connectionProperties;

  /// The connection string
  std::string m_connectionString;

  /// The type converter
  coral::ITypeConverter& m_typeConverter;

  /// The environment handle
  OCIEnv* m_environmentHandle;
```

```
      /// The error handle
      OCIError* m_errorHandle;

      /// The server handle
      OCIServer* m_serverHandle;

      /// The server major version
      int m_serverVersion;

      /// The session reference
      coral::ISession& m_session;

      /// The OCI service context handle
      OCISvcCtx* m_serviceContextHandle;

      /// The monitoring service
      coral::monitor::IMonitoringService* m_monitoringService;

      /// The schema name
      std::string m_schemaName;

      /// The read-only flag
      bool m_isReadOnly;

      /// The mutex lock
      mutable boost::mutex m_mutex;

      /// The 'select any table' flag
      mutable bool m_selectAnyTable;


      /// NetworkGlitch: The session handle
      OCISession*        m_sessionHandle;

      /// NetworkGlitch: The username
      std::string m_userName;

      /// NetworkGlitch: The password
      std::string m_password;

    };

  }

}


// Inline methods
inline const coral::OracleAccess::ConnectionProperties&
coral::OracleAccess::SessionProperties::connectionProperties() const
{
  return m_connectionProperties;
}


inline std::string
coral::OracleAccess::SessionProperties::connectionString() const
{
  return m_connectionString;
}


inline coral::ITypeConverter&
coral::OracleAccess::SessionProperties::typeConverter()
{
  return m_typeConverter;
}


inline const coral::ITypeConverter&
coral::OracleAccess::SessionProperties::typeConverter() const
{
  return m_typeConverter;
}
```

```
inline OCIEnv*
coral::OracleAccess::SessionProperties::environmentHandle() const
{
  return m_environmentHandle;
}


inline OCIError*
coral::OracleAccess::SessionProperties::errorHandle() const
{
  return m_errorHandle;
}


inline OCIServer*
coral::OracleAccess::SessionProperties::serverHandle() const
{
  return m_serverHandle;
}


inline int
coral::OracleAccess::SessionProperties::serverVersion() const
{
  return m_serverVersion;
}


inline void
coral::OracleAccess::SessionProperties::setServiceContextHandle( OCISvcCtx*
serviceContextHandle )
{
  boost::mutex::scoped_lock lock(m_mutex);
  m_serviceContextHandle = serviceContextHandle;
}


inline OCISvcCtx*
coral::OracleAccess::SessionProperties::serviceContextHandle() const
{
  boost::mutex::scoped_lock lock(m_mutex);
  return m_serviceContextHandle;
}


inline void
coral::OracleAccess::SessionProperties::setMonitoringService(
coral::monitor::IMonitoringService* monitoringService )
{
  boost::mutex::scoped_lock lock(m_mutex);
  m_monitoringService = monitoringService;
}


inline coral::monitor::IMonitoringService*
coral::OracleAccess::SessionProperties::monitoringService() const
{
  boost::mutex::scoped_lock lock(m_mutex);
  return m_monitoringService;
}


inline coral::ISession&
coral::OracleAccess::SessionProperties::session() const
{
  return m_session;
}


inline bool
coral::OracleAccess::SessionProperties::isReadOnly() const
{
  return m_isReadOnly;
}


inline std::string
```

```
coral::OracleAccess::SessionProperties::schemaName() const
{
  return m_schemaName;
}

#endif
```

SessionProperties.cpp

in coral/OracleAccess/src

```
#include "CoralKernel/Service.h"
#include "CoralKernel/Context.h"

#include "CoralBase/MessageStream.h"
#include "CoralBase/AttributeList.h"

#include "CoralCommon/IDevSession.h"

#include "RelationalAccess/ISession.h"
#include "RelationalAccess/ITransaction.h"
#include "RelationalAccess/SessionException.h"

#include "ConnectionProperties.h"
#include "SessionProperties.h"
#include "DomainProperties.h"
#include "OracleErrorHandler.h"
#include "OracleStatement.h"
#include "Schema.h"

#include <oci.h>
#include <iostream>

coral::OracleAccess::SessionProperties::SessionProperties(
coral::OracleAccess::ConnectionProperties& connectionProperties,
                                                          const
std::string& schemaName,
                                                          coral::ISession&
session,
                                                          bool readOnly ) :
  m_connectionProperties( connectionProperties ),
  m_connectionString( connectionProperties.connectionString() ),
  m_typeConverter( connectionProperties.typeConverter() ),
  m_environmentHandle( connectionProperties.environmentHandle() ),
  m_errorHandle( connectionProperties.errorHandle() ),
  m_serverHandle( connectionProperties.serverHandle() ),
  m_serverVersion( connectionProperties.serverVersion() ),
  m_session( session ),
  m_serviceContextHandle( 0 ),
  m_monitoringService( 0 ),
  m_schemaName( schemaName ),
  m_isReadOnly( readOnly ),
  m_mutex(),
  m_selectAnyTable( true ),
  m_sessionHandle( 0 )
{
  //std::cout << "Create SessionProperties " << this << std::endl; // debug
bug #73334
}

coral::OracleAccess::SessionProperties::~SessionProperties()
{
  //std::cout << "Delete SessionProperties " << this << std::endl; // debug
bug #73334
}


const coral::OracleAccess::DomainProperties&
coral::OracleAccess::SessionProperties::domainProperties() const
{
  return m_connectionProperties.domainProperties();
}
```

```
coral::ISchema&
coral::OracleAccess::SessionProperties::schema() const
{
  return m_session.nominalSchema();
}


bool
coral::OracleAccess::SessionProperties::isTransactionActive() const
{
  return m_session.transaction().isActive();
}


bool
coral::OracleAccess::SessionProperties::isTransactionReadOnly() const
{
  return m_session.transaction().isReadOnly();
}


bool
coral::OracleAccess::SessionProperties::selectAnyTable() const
{
  return m_selectAnyTable;
}


void
coral::OracleAccess::SessionProperties::cannotSelectAnyTable() const
{
  m_selectAnyTable = false;
}

bool
coral::OracleAccess::SessionProperties::wasConnectionLost()
{
  //boost::mutex::scoped_lock lock( m_mutex );

  bool wasLost = false;
  text serverVersion[1000];
  sword status = OCIServerVersion( m_connectionProperties.serverHandle(),
m_connectionProperties.errorHandle(),
                                   serverVersion, 1000, OCI_HTYPE_SERVER );

  if ( status != OCI_SUCCESS )
  {
    wasLost = true;
  }

  return wasLost;
}

bool
coral::OracleAccess::SessionProperties::wasSessionLost()
{
  //boost::mutex::scoped_lock lock( m_mutex );

  bool wasLost;

  if( !static_cast<coral::IDevSession&>(m_session).isSessionValid() )
  {
    wasLost = true;

    m_environmentHandle = m_connectionProperties.environmentHandle();
    m_errorHandle = m_connectionProperties.errorHandle();
    m_serverHandle = m_connectionProperties.serverHandle();
  }

  else
  {
    wasLost = false;
  }

  return wasLost;
}
```

```
bool
coral::OracleAccess::SessionProperties::restartConnection()
{
  m_environmentHandle = 0;
  m_errorHandle = 0;
  m_serverHandle = 0;

  // Create the environment
  ub4 mode = OCI_OBJECT | OCI_THREADED; // 0x00000002 | 0x00000001
  static bool first = true;
  if ( first )
  {
    first = false;
    coral::MessageStream log( "" );
    if ( getenv( "CORAL_ORA_NO_OCI_THREADED" ) )
    {
      mode = OCI_OBJECT; // 0x00000002
      log << coral::Warning
          << "Env CORAL_ORA_NO_OCI_THREADED was specified: "
          << "OCIEnvCreate will use OCI_OBJECT "
          << "instead of OCI_OBJECT | OCI_THREADED" <<
coral::MessageStream::endmsg;
    }
    else if ( getenv( "CORAL_ORA_OCI_NO_MUTEX" )  )
    {
      mode = OCI_OBJECT | OCI_THREADED | OCI_NO_MUTEX; // 0x00000002 |
0x00000001 | 0x00000080
      log << coral::Warning
          << "Env CORAL_ORA_OCI_NO_MUTEX was specified: "
          << "OCIEnvCreate will use OCI_OBJECT | OCI_THREADED | OCI_NO_MUTEX
"
          << "instead of OCI_OBJECT | OCI_THREADED" <<
coral::MessageStream::endmsg;
    }
  }

  sword status = OCIEnvCreate( &m_environmentHandle, mode , 0,0,0,0,0,0 );

  if ( status != OCI_SUCCESS )
    {
      m_environmentHandle = 0;
      status = OCIEnvCreate( &m_environmentHandle, mode , 0,0,0,0,0,0 );
    }
  if ( status != OCI_SUCCESS )
    {
      throw coral::ServerException( "",
                                "Could not allocate an OCI environment handle"
);
    }
  // Creating the error handle
  void* temporaryPointer = 0;
  status = OCIHandleAlloc( m_environmentHandle, &temporaryPointer,
                           OCI_HTYPE_ERROR, 0, 0 );
  if ( status != OCI_SUCCESS ) {
    OCIHandleFree( m_environmentHandle, OCI_HTYPE_ENV );
    throw coral::ServerException( "",
                              "Could not allocate an OCI error handle" );
  }
  m_errorHandle = static_cast< OCIError* >( temporaryPointer );


  // Creating a server handle
  temporaryPointer = 0;
  //void* temporaryPointer = 0;
  //sword
  status = OCIHandleAlloc( m_environmentHandle, &temporaryPointer,
                           OCI_HTYPE_SERVER, 0, 0 );

  if ( status != OCI_SUCCESS ) {
    OCIHandleFree( m_errorHandle, OCI_HTYPE_ERROR );
    OCIHandleFree( m_environmentHandle, OCI_HTYPE_ENV );
    throw coral::ServerException( "",
                              "Could not allocate an OCI server handle" );
  }

  m_serverHandle = static_cast< OCIServer* >( temporaryPointer );
  // Creating an error handler to be used in the subsequent calls.
```

```
  coral::OracleAccess::OracleErrorHandler errorHandler( m_errorHandle );

  // Attaching the server
  status = OCIServerAttach( m_serverHandle, m_errorHandle,
                            reinterpret_cast<const text *>(
connectionString().c_str() ),
                            connectionString().size(),
                            OCI_DEFAULT );

  if ( status != OCI_SUCCESS ) {
    errorHandler.handleCase( status, "attaching a server
'"+connectionString()+"'" );
    coral::MessageStream log( "" );
    if ( errorHandler.isError() ) {
      log << coral::Error << errorHandler.message() <<
coral::MessageStream::endmsg;
      OCIHandleFree( m_serverHandle, OCI_HTYPE_SERVER );
      OCIHandleFree( m_errorHandle, OCI_HTYPE_ERROR );
      OCIHandleFree( m_environmentHandle, OCI_HTYPE_ENV );
      const sb4 lastErrorCode = errorHandler.lastErrorCode();
      if ( lastErrorCode == 12154 || lastErrorCode == 12538 ||
          lastErrorCode == 12545 || lastErrorCode == 12505) { // Wrong db
specification
       throw coral::DatabaseNotAccessibleException( "",
                                                    "IConnection::connect" );
      }
      if ( true ) { // Retry in these cases
       throw coral::ConnectionException( "",
                                         "IConnection::connect" );
      }
      else { // Do not retry
       throw coral::ServerException( "" );
      }
    }
    else {
      log << coral::Warning << errorHandler.message() <<
coral::MessageStream::endmsg;
    }
  }


  //Check if the reconnection is successfully restored

  bool logicalConnection = false;
  text serverVersion[1000];
  status = OCIServerVersion( serverHandle(), errorHandle(),
                             serverVersion, 1000, OCI_HTYPE_SERVER );

  if ( status == OCI_SUCCESS )
  {
    logicalConnection = true;
    // AK: Small hack for the const workaround
    //TODO: reconnect connection directly in ConnectionProperties
    const_cast<ConnectionProperties&>(m_connectionProperties).setHandles(
m_environmentHandle
                                                                         ,
m_errorHandle
                                                                         ,
m_serverHandle
                                                                         ,
m_serverVersion );
  }

  return logicalConnection;
}

bool
coral::OracleAccess::SessionProperties::restartConnectionWithTimeOut()
{
  int time_counter = 0;
  while( restartConnection() == false )
  {
    if(time_counter > 10)
    {
      /// FIXME use special exception here
      throw coral::Exception("Connection cannot be established with DB
```

```
server!",
                                "SessionProperties::restartSession",
                                 domainProperties().service()->name() );
    }
    sleep(10);
    time_counter++;
  }
  return true;
}

void
coral::OracleAccess::SessionProperties::restartSession()
{
  m_serviceContextHandle = 0;
  m_sessionHandle = 0;

  // Creating an error handler to be used in the subsequent calls.
  coral::OracleAccess::OracleErrorHandler errorHandler( m_errorHandle );

  // Creating a service context handle
  void* temporaryPointer = 0;
  sword status = OCIHandleAlloc( m_environmentHandle,
                                 &temporaryPointer,
                                 OCI_HTYPE_SVCCTX, 0, 0 );
  if ( status != OCI_SUCCESS )
  {
    throw coral::SessionException( "Could not allocate an OCI service
context handle",
                                   "ISession::connectAsUser",
                                   domainProperties().service()->name() );
  }
  OCISvcCtx* serviceContextHandle = static_cast< OCISvcCtx* >(
temporaryPointer );

  // Setting the server to the service context
  status = OCIAttrSet( serviceContextHandle, OCI_HTYPE_SVCCTX,
                       m_serverHandle, 0, OCI_ATTR_SERVER,
                       m_errorHandle );

  if ( status != OCI_SUCCESS )
  {
    errorHandler.handleCase( status, "setting the server to the service
context" );
    coral::MessageStream log( domainProperties().service()->name() );
    if ( errorHandler.isError() )
    {
      OCIHandleFree( serviceContextHandle, OCI_HTYPE_SVCCTX );
      log << coral::Error << errorHandler.message() <<
coral::MessageStream::endmsg;
      throw coral::SessionException( "Could not set the server to the
service context",
                                     "ISession::connectAsUser",
                                     domainProperties().service()->name() );
    }
    else
    {
      log << coral::Warning << errorHandler.message() <<
coral::MessageStream::endmsg;
    }
  }

  // Updating the context handle
  m_serviceContextHandle = serviceContextHandle;
  //m_properties->setServiceContextHandle( serviceContextHandle );

  // Allocating the user session handle
  temporaryPointer = 0;
  status = OCIHandleAlloc( m_environmentHandle, &temporaryPointer,
                           OCI_HTYPE_SESSION, 0, 0 );

  if ( status != OCI_SUCCESS )
  {
    throw coral::SessionException( "Could not allocate a user session
handle",
                                   "ISession::startUserSession",
                                   domainProperties().service()->name() );
  }
```

```
  m_sessionHandle = static_cast< OCISession* >( temporaryPointer );

  // Setting the user session to the service context
  status = OCIAttrSet( m_serviceContextHandle, OCI_HTYPE_SVCCTX,
                       m_sessionHandle, 0, OCI_ATTR_SESSION, m_errorHandle
);

  if ( status != OCI_SUCCESS )
  {
    errorHandler.handleCase( status, "setting the user session to the
service context" );
    coral::MessageStream log( domainProperties().service()->name() );
    if ( errorHandler.isError() )
    {
      log << coral::Error << errorHandler.message() <<
coral::MessageStream::endmsg;
      OCIHandleFree( m_sessionHandle, OCI_HTYPE_SESSION );
      m_sessionHandle = 0;
      throw coral::SessionException( "Could not allocate a user session
handle",
                                     "ISession::startUserSession",
                                     domainProperties().service()->name() );
    }
    else {
      log << coral::Warning << errorHandler.message() <<
coral::MessageStream::endmsg;
    }
  }

  // Setting the user name and the password
  status = OCIAttrSet( m_sessionHandle, OCI_HTYPE_SESSION,
                       const_cast< char *>( m_userName.c_str() ),
m_userName.size(),
                       OCI_ATTR_USERNAME, m_errorHandle );

  if ( status == OCI_SUCCESS )
  {
    status = OCIAttrSet( m_sessionHandle, OCI_HTYPE_SESSION,
                         const_cast< char *>( m_password.c_str() ),
m_password.size(),
                         OCI_ATTR_PASSWORD, m_errorHandle );
  }

  if ( status != OCI_SUCCESS )
  {
    errorHandler.handleCase( status, "setting the user name and the
password" );
    coral::MessageStream log( domainProperties().service()->name() );
    if ( errorHandler.isError() ) {
      log << coral::Error << errorHandler.message() <<
coral::MessageStream::endmsg;
      OCIHandleFree( m_sessionHandle, OCI_HTYPE_SESSION );
      m_sessionHandle = 0;
      throw coral::SessionException( "Could not set user name and password",
                                     "ISession::startUserSession",
                                     domainProperties().service()->name() );
    }
    else
    {
      log << coral::Warning << errorHandler.message() <<
coral::MessageStream::endmsg;
    }
  }

  // Authenticating
  status = OCISessionBegin( m_serviceContextHandle,
                            m_errorHandle,
                            m_sessionHandle,
                            OCI_CRED_RDBMS, OCI_DEFAULT);

  if ( status != OCI_SUCCESS )
  {
    errorHandler.handleCase( status, "authenticating" );
    coral::MessageStream log( domainProperties().service()->name() );
    if ( errorHandler.isError() )
    {
```

```
      log << coral::Error << errorHandler.message() <<
coral::MessageStream::endmsg;
      OCIHandleFree( m_sessionHandle, OCI_HTYPE_SESSION );
      m_sessionHandle = 0;
      if ( errorHandler.lastErrorCode() == 1017 )
      {
        // Failed to authenticate. Do not retry!
        throw coral::AuthenticationFailureException(
domainProperties().service()->name() );
      }
      // Workaround for bug #58522 (ORA-24327), aka bug #65709, aka bug
#75596
      // Do not retry to start a session if connection was lost with ORA-
03113
      // or ORA-03135 (or other signals that eventually result in ORA-24327)
      else if ( errorHandler.lastErrorCode() == 3113 ||
                errorHandler.lastErrorCode() == 3135 ||
                errorHandler.lastErrorCode() == 24327 ) { // Connection
lost. Do not retry to start a session on this connection!
        throw coral::SessionException( errorHandler.message(),
                                       "ISession::startUserSession",
                                       domainProperties().service()->name()
);
      }
      else if ( true ) { // Temporarily unavailable. Retry!
        throw coral::StartSessionException( domainProperties().service()-
>name(),
                                            "ISession::startUserSession" );
      }
      else { // All other failures. (NO PATH TO THIS STATEMENT!)
        throw coral::SessionException( errorHandler.message(),
                                       "ISession::startUserSession",
                                       domainProperties().service()->name()
);
      }
    }
    else {
      log << coral::Warning << errorHandler.message() <<
coral::MessageStream::endmsg;
    }
  }

  // Debug
  {
    coral::MessageStream log( domainProperties().service()->name() );
    log << coral::Verbose << "New user session started with OCISessionBegin"
        << " (OCISession*=" << m_sessionHandle
        << ", OCISvcCtx*=" << m_serviceContextHandle
        << ") on connection (OCIServer*=" << m_serverHandle << ")"
        << coral::MessageStream::endmsg;
  }

  // Enable sql trace
  static const char* cerntrace_on = ::getenv( "CORAL_ORA_CERN_TRACE_ON" );
  if ( cerntrace_on )
  {
    coral::MessageStream log( domainProperties().service()->name() );
    log << coral::Info << "Enable CERN SQL trace" <<
coral::MessageStream::endmsg;
    std::string sqlStatement = "BEGIN CERN_TRACE.CSTART_TRACE( 2 ); END;";

    coral::OracleAccess::OracleStatement statement( *this, schemaName(),
sqlStatement );

    if ( ! statement.execute( coral::AttributeList() ) )
    {
      log << coral::Warning << "Could not enable CERN SQL trace" <<
coral::MessageStream::endmsg;
    }
  }
  else
  {
    static const char* sqltrace_on = ::getenv( "CORAL_ORA_SQL_TRACE_ON" );
    if ( sqltrace_on ) {
      // SQL trace identifier
      static const char* sqltrace_id = ::getenv(
"CORAL_ORA_SQL_TRACE_IDENTIFIER" );
```

```
      if ( sqltrace_id )
      {
        coral::MessageStream log( domainProperties().service()->name() );
        log << coral::Info << "Set the SQL trace identifier" <<
coral::MessageStream::endmsg;
        static std::string sqlStatement = "ALTER SESSION SET
tracefile_identifier='" + std::string( sqltrace_id ) + "'";
        coral::OracleAccess::OracleStatement statement( *this, schemaName(),
sqlStatement );
        if ( ! statement.execute( coral::AttributeList() ) )
        {
          log << coral::Warning << "Could not set the SQL trace identifier"
<< coral::MessageStream::endmsg;
        }
      }
      // Turn on the appropriate SQL trace
      static std::string sqlStatement = "";
      if ( sqlStatement == "" )
      {
        if ( std::string( sqltrace_on ) == "3106" )
          sqlStatement = "ALTER SESSION SET EVENTS '3106 TRACE NAME
ERRORSTACK LEVEL 10'";
        else if ( std::string( sqltrace_on ) == "10046" )
          sqlStatement = "ALTER SESSION SET EVENTS '10046 TRACE NAME CONTEXT
FOREVER, LEVEL 12'";
        else if ( std::string( sqltrace_on ) == "10053" )
          sqlStatement = "ALTER SESSION SET EVENTS '10053 TRACE NAME CONTEXT
FOREVER, LEVEL 1'";
        else if ( std::string( sqltrace_on ) == "10132" )
          sqlStatement = "ALTER SESSION SET EVENTS '10132 TRACE NAME CONTEXT
FOREVER, LEVEL 12'";
        else
          sqlStatement = "ALTER SESSION SET SQL_TRACE=TRUE";
      }
      coral::MessageStream log( domainProperties().service()->name() );
      log << coral::Info << "Enable SQL trace by " << sqlStatement <<
coral::MessageStream::endmsg;
      coral::OracleAccess::OracleStatement statement( *this, schemaName(),
sqlStatement );
      if ( ! statement.execute( coral::AttributeList() ) )
      {
        log << coral::Warning << "Could not enable SQL trace" <<
coral::MessageStream::endmsg;
      }
    }
  }

  // Enable automatic plan capture (Oracle 11g)
  static const char* capture_on = ::getenv(
"CORAL_ORA_CAPTURE_SQL_PLAN_BASELINES" );
  if ( capture_on )
  {
    std::string sqlStatement = "ALTER SESSION SET
OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES=TRUE";
    coral::MessageStream log( domainProperties().service()->name() );
    log << coral::Info << "Enable automatic plan capture by " <<
sqlStatement << coral::MessageStream::endmsg;
    coral::OracleAccess::OracleStatement statement( *this, schemaName(),
sqlStatement );
    if ( ! statement.execute( coral::AttributeList() ) )
    {
      log << coral::Warning << "Could not enable automatic plan capture" <<
coral::MessageStream::endmsg;
    }
  }

  // Task #10775 (performance optimization for data dictionary queries)
  // Special settings for special CERN accounts (nightly and release tests)
  static const char* selectsys_on = ::getenv(
"CORAL_ORA_SELECT_FROM_SYS_TABLES" );
  if ( !selectsys_on )
  {
    std::string userNameUp = m_userName;
    std::transform( userNameUp.begin(), userNameUp.end(),
userNameUp.begin(), (int (*)(int)) std::toupper );
    if ( userNameUp != "AVALASSI" &&
         userNameUp != "LCG_COOL" &&
```

```
          userNameUp != "LCG_COOL_NIGHTLY" &&
          userNameUp != "LCG_CORAL_NIGHTLY" &&
          userNameUp != "LCG_POOL_NIGHTLY" )
      {
        cannotSelectAnyTable();
      }
      //std::cout << std::endl << "USER: " << userName << std::endl;
    }
    // Very important to not miss this
    static_cast<coral::IDevSession&>(m_session).validateSession();
}
```

Transaction.h

in coral/OracleAccess/src

```
#ifndef ORACLEACCESS_TRANSACTION_H
#define ORACLEACCESS_TRANSACTION_H

#include "RelationalAccess/ITransaction.h"
#include "CoralBase/boost_thread_headers.h"
#include "QueryMgr.h"

struct OCITrans;

namespace coral {

  namespace OracleAccess {

    class SessionProperties;
    class ITransactionObserver;

    /**
     * Class Transaction
     * Implementation of the ITransaction interface for the OracleAccess
plugin
     */
    class Transaction : virtual public coral::ITransaction
    {
    public:
      /// Constructor
      Transaction( SessionProperties& properties,
                   ITransactionObserver& observer,
                   TransactionStatusInstance& tsi,
                   bool isSerializable = true );

      /// Destructor
      virtual ~Transaction();

      /**
       * Starts a new transaction.
       * In case of failure a TransactionNotStartedException is thrown.
       *
       */
      void start( bool readOnly = false );

      /**
       * Commits the transaction.
       * In case of failure a TransactionNotCommittedException is thrown.
       */
      void commit();

      /**
       * Aborts and rolls back a transaction.
       */
      void rollback();

      /**
       * Returns the status of the transaction (if it is active or not)
       */
      bool isActive() const;

      /**
       * Returns the mode of the transaction (if it is read-only or not)
       */
```

```
      bool isReadOnly() const;

      /**
       * Returns the mode of the transaction (if it is serializable or not)
       */
      bool isSerializable() const
      {
        return m_isSerializable;
      }

    private:
      /// Allocates the necessary handles
      void allocateHandles();

      /// Releases the handles
      void releaseHandles();

    private:
      /// The session properties
      SessionProperties&    m_sessionProperties;

      /// The transaction observer
      ITransactionObserver& m_observer;

      /// Flag indicating whether read-only transactions should be skipped
      bool m_isSerializable;

      /// The OCI transaction handle
      OCITrans*             m_transactionHandle;

      /// Flag indicating whether a transaction is read-only
      bool m_isReadOnly;

      /// The mutex lock
      mutable boost::mutex m_mutex;

      /// Shared transaction status information (not owned)
      TransactionStatusInstance& m_tsi;

    };

  }

}

#endif
```

Transaction.cpp

in coral/OracleAccess/src

```
#ifdef WIN32
#include <WTypes.h> // AV 17.04.2008 include this first on Win (bug #35683)
#endif

#include "Transaction.h"
#include "SessionProperties.h"
#include "DomainProperties.h"
#include "OracleErrorHandler.h"
#include "ITransactionObserver.h"
#include "QueryMgr.h"

#include "CoralCommon/MonitoringEventDescription.h"
#include "CoralCommon/SimpleTimer.h"

#include "CoralCommon/IDevConnection.h"

#include "RelationalAccess/IMonitoringService.h"
#include "RelationalAccess/SessionException.h"

#include "CoralBase/MessageStream.h"
#include "CoralKernel/Service.h"

#include "oci.h"
```

```
coral::OracleAccess::Transaction::Transaction(
coral::OracleAccess::SessionProperties& properties,
                                               ITransactionObserver&
observer,
                                               TransactionStatusInstance&
tsi,
                                               bool isSerializable )
: m_sessionProperties( properties )
, m_observer( observer )
, m_isSerializable( isSerializable )
, m_transactionHandle( 0 )
, m_isReadOnly( false )
, m_mutex()
, m_tsi( tsi )
{
}


coral::OracleAccess::Transaction::~Transaction()
{
  if ( this->isActive() ) this->rollback();
}


void
coral::OracleAccess::Transaction::start( bool readOnly )
{
  if ( m_sessionProperties.wasConnectionLost())
  {

    const ConnectionProperties& cproperties =
m_sessionProperties.connectionProperties();
    // AK: small hack
    ///AK: FIXME: access to connection from transaction?

const_cast<ConnectionProperties&>(cproperties).getConnection().invalidateAll
();

    if(m_sessionProperties.restartConnectionWithTimeOut())
    {
      m_sessionProperties.restartSession();
    }
  }
  else if(m_sessionProperties.wasSessionLost())
  {
    m_sessionProperties.restartSession();
  }

  if ( this->isActive() ) {
    coral::MessageStream log(
m_sessionProperties.domainProperties().service()->name() );
    log << coral::Warning << "A transaction is already active" <<
coral::MessageStream::endmsg;
    return;
  }

  boost::mutex::scoped_lock lock(m_mutex);

  if ( ! readOnly && m_sessionProperties.isReadOnly() )
  {
    throw coral::InvalidOperationInReadOnlyModeException(
m_sessionProperties.domainProperties().service()->name(),

"ITransaction::start" );
  }

  this->allocateHandles();
  if ( !readOnly || m_isSerializable ) // SKIP READONLY TRANSACTIONS
  {
    sword status = OCITransStart(
m_sessionProperties.serviceContextHandle(),
                                 m_sessionProperties.errorHandle(),
                                 0,
                                 OCI_TRANS_NEW | ( readOnly ?
OCI_TRANS_READONLY : 0 ) );
```

```
    if ( status != OCI_SUCCESS ) {
      coral::MessageStream log(
m_sessionProperties.domainProperties().service()->name() );
      coral::OracleAccess::OracleErrorHandler errorHandler(
m_sessionProperties.errorHandle() );
      errorHandler.handleCase( status, "start transaction" );
      if ( errorHandler.isError() ) {
        log << coral::Error << errorHandler.message() <<
coral::MessageStream::endmsg;
        this->releaseHandles();
        throw coral::TransactionNotStartedException(
m_sessionProperties.domainProperties().service()->name() );
      }
      log << coral::Warning << errorHandler.message() <<
coral::MessageStream::endmsg;
    }
  }

  m_isReadOnly = readOnly;
  if ( m_sessionProperties.monitoringService() ) {
    m_sessionProperties.monitoringService()->record( "oracle://" +
m_sessionProperties.connectionString() + "/" +
m_sessionProperties.schemaName(),

coral::monitor::Transaction,
                                                     coral::monitor::Info,

monitoringEventDescription.sessionBegin(),
                                                     ( m_isReadOnly ?
monitoringEventDescription.transactionReadOnly() :
monitoringEventDescription.transactionUpdate() ) );
  }
  // Set the status information to used/active
  m_tsi->increase();
}


void
coral::OracleAccess::Transaction::commit()
{
  coral::MessageStream log(
m_sessionProperties.domainProperties().service()->name() );

  if ( ! ( this->isActive() ) ) {
    log << coral::Warning << "No active transaction to commit" <<
coral::MessageStream::endmsg;
    return;
  }
  // Set the status information to not used anymore/inactive
  m_tsi->decrease();

  boost::mutex::scoped_lock lock(m_mutex);

  coral::SimpleTimer timer;

  if ( m_sessionProperties.monitoringService() )
    timer.start();

  sword status = OCI_SUCCESS;
  if ( !m_isReadOnly || m_isSerializable ) // SKIP READONLY TRANSACTIONS
    status = OCITransCommit( m_sessionProperties.serviceContextHandle(),
                             m_sessionProperties.errorHandle(),
                             OCI_DEFAULT );

  if ( m_sessionProperties.monitoringService() )
  {
    timer.stop();
    double idleTimeSeconds = timer.total() * 1e-6;
    m_sessionProperties.monitoringService()->record( "oracle://" +
m_sessionProperties.connectionString() + "/" +
m_sessionProperties.schemaName(),

coral::monitor::Transaction,
                                                     coral::monitor::Time,

monitoringEventDescription.transactionCommit(),
                                                     idleTimeSeconds );
```

```
  }

  m_observer.reactOnEndOfTransaction();

  if ( status != OCI_SUCCESS ) {
    coral::OracleAccess::OracleErrorHandler errorHandler(
m_sessionProperties.errorHandle() );
    errorHandler.handleCase( status, "commit transaction" );
    log << coral::Error << errorHandler.message() <<
coral::MessageStream::endmsg;
    this->releaseHandles();
    throw coral::TransactionNotCommittedException(
m_sessionProperties.domainProperties().service()->name() );
  }

  this->releaseHandles();
}


void
coral::OracleAccess::Transaction::rollback()
{
  coral::MessageStream log(
m_sessionProperties.domainProperties().service()->name() );

  if ( ! ( this->isActive() ) ) {
    log << coral::Warning << "No active transaction to roll back" <<
coral::MessageStream::endmsg;
    return;
  }
  // Set the status information to not used anymore/inactive
  m_tsi->decrease();

  boost::mutex::scoped_lock lock(m_mutex);
  sword status = OCI_SUCCESS;
  if ( !m_isReadOnly || m_isSerializable ) // SKIP READONLY TRANSACTIONS
    status = OCITransRollback( m_sessionProperties.serviceContextHandle(),
                               m_sessionProperties.errorHandle(),
                               OCI_DEFAULT );

  if ( m_sessionProperties.monitoringService() ) {
    m_sessionProperties.monitoringService()->record( "oracle://" +
m_sessionProperties.connectionString() + "/" +
m_sessionProperties.schemaName(),

coral::monitor::Transaction,
                                                     coral::monitor::Info,

monitoringEventDescription.transactionRollback() );
  }

  m_observer.reactOnEndOfTransaction();

  if ( status != OCI_SUCCESS ) {
    coral::OracleAccess::OracleErrorHandler errorHandler(
m_sessionProperties.errorHandle() );
    errorHandler.handleCase( status, "roll back transaction" );
    log << coral::Error << errorHandler.message() <<
coral::MessageStream::endmsg;
  }

  this->releaseHandles();
}


void
coral::OracleAccess::Transaction::allocateHandles()
{
  coral::MessageStream log(
m_sessionProperties.domainProperties().service()->name() );

  // Allocating a new transaction handle
  void* temporaryPointer = 0;
  sword status = OCIHandleAlloc( m_sessionProperties.environmentHandle(),
                                 &temporaryPointer,
                                 OCI_HTYPE_TRANS, 0, 0 );
  if ( status != OCI_SUCCESS ) {
```

```
    throw coral::TransactionNotStartedException(
m_sessionProperties.domainProperties().service()->name(),
                                                "Failed to allocate a new
transaction handle" );
  }
  m_transactionHandle = static_cast< OCITrans* >( temporaryPointer );

  // Set the transaction attribute in the service context
  status = OCIAttrSet( m_sessionProperties.serviceContextHandle(),
OCI_HTYPE_SVCCTX,
                       m_transactionHandle, 0, OCI_ATTR_TRANS,
m_sessionProperties.errorHandle() );
  if ( status != OCI_SUCCESS ) {
    coral::OracleAccess::OracleErrorHandler errorHandler(
m_sessionProperties.errorHandle() );
    errorHandler.handleCase( status, "Setting the transaction attribute in
the service context" );
    if ( errorHandler.isError() ) {
      log << coral::Error << errorHandler.message() <<
coral::MessageStream::endmsg;
      OCIHandleFree( m_transactionHandle, OCI_HTYPE_TRANS );
      m_transactionHandle = 0;
      throw coral::TransactionNotStartedException(
m_sessionProperties.domainProperties().service()->name(),
                                                  "Failed to set the
transaction attribute in the service context" );
    }
    else {
      log << coral::Warning << errorHandler.message() <<
coral::MessageStream::endmsg;
    }
  }
}


void
coral::OracleAccess::Transaction::releaseHandles()
{

  // Release the transaction handle
  sword status = OCIHandleFree( m_transactionHandle, OCI_HTYPE_TRANS );
  if ( status != OCI_SUCCESS )
  {
    coral::OracleAccess::OracleErrorHandler errorHandler(
m_sessionProperties.errorHandle() );
    errorHandler.handleCase( status, "Releasing the transaction handle" );
    coral::MessageStream log(
m_sessionProperties.domainProperties().service()->name() );
    if ( errorHandler.isError() )
      log << coral::Error << errorHandler.message() <<
coral::MessageStream::endmsg;
    else
      log << coral::Warning << errorHandler.message() <<
coral::MessageStream::endmsg;
  }
  m_transactionHandle = 0;
  // Clear the transaction attribute in the service context (fix bug #61090)
  status = OCIAttrSet( m_sessionProperties.serviceContextHandle(),
                       OCI_HTYPE_SVCCTX,
                       m_transactionHandle, 0, OCI_ATTR_TRANS,
                       m_sessionProperties.errorHandle() );
  if ( status != OCI_SUCCESS )
  {
    coral::OracleAccess::OracleErrorHandler errorHandler(
m_sessionProperties.errorHandle() );
    errorHandler.handleCase( status, "Clearing the transaction attribute in
the service context" );
    coral::MessageStream log(
m_sessionProperties.domainProperties().service()->name() );
    if ( errorHandler.isError() )
      log << coral::Error << errorHandler.message() <<
coral::MessageStream::endmsg;
    else
      log << coral::Warning << errorHandler.message() <<
coral::MessageStream::endmsg;
  }
}
```

```
bool
coral::OracleAccess::Transaction::isActive() const
{
  boost::mutex::scoped_lock lock(m_mutex);
  return ( m_transactionHandle != 0 );
}


bool
coral::OracleAccess::Transaction::isReadOnly() const
{
  boost::mutex::scoped_lock lock(m_mutex);
  return m_isReadOnly;
}
```