CORAL, A SOFTWARE SYSTEM FOR VENDOR-NEUTRAL ACCESS TO RELATIONAL DATABASES

I. Papadopoulos, R. Chytracek, D. Düllmann, G. Govi, IT Department, CERN, Geneva, Switzerland
Y. Shapiro, ATLAS Database Group, PH Department, CERN, Geneva, Switzerland
Z. Xie, Princeton University, Princeton, New Jersey 08544 USA

Abstract

The COmmon Relational Abstraction Layer (CORAL)[1] is a C++ software system,developed within the context of the LCG persistency framework, which provides vendor-neutral software access to relational databases with defined semantics. The SQL-free public interfaces ensure the encapsulation of all the differences that one may find among the various RDBMS flavours in terms of SQL syntax and data types. CORAL has been developed following a component architecture where the various RDBMS-specific implementations of the interfaces are loaded as plugin libraries at run-time whenever required. The system addresses the needs related to the distributed deployment of relational data by providing hooks for client-side monitoring, database service indirection and application-level connection pooling.

INTRODUCTION

CORAL is the set of software deliverables of the "Database Access and Distribution" work package of the POOL project[2]. The development of libraries for the vendor-independent database access, collectively referred to as the Relational Abstraction Layer (RAL), started within POOL in spring 2004. At that time the main requirement that needed to be met was the creation of an insulation layer that would allow the development of software components responsible for accessing data in RDBMS without the knowledge of the subtle differences among the various technology-specific solutions. Since then RAL was adopted by the relational components of POOL (File Catalog, Collections), the COOL (Conditions Database) project[3], and several components of the software frameworks of the LHC experiments.

In spring 2005 a formal review took place of the existing RAL API with feedback from the direct clients of RAL, within and outside POOL. During the review new emerging use cases have been considered that are related to the database deployment and distrubution issues. The outcome was the design of an improved version of RAL, which is now being developed and packaged independently of the rest of the POOL components under the new name CORAL (COmmon Relational Abstraction Layer).

PROJECT SCOPE

The primary goal of CORAL is to provide functionality for accessing data in relational databases using a C++ API, free of SQL commands and types, shielding the user from the technology-specific APIs. The need for the maximal SQL insulation from the client software can be demonstrated simply showing the SQL statements for two technologies (Oracle[5] and MySQL[6]), which are used mostly in HEP, in rather simple tasks:

1. Creation of a table:
   - MySQL syntax:
     `CREATE TABLE T_t`
     `(I BIGINT, X DOUBLE)`
   - Oracle syntax:
     `CREATE TABLE "T_t"`
     `(I NUMBER(20), X BINARY_DOUBLE)`
2. Query fetching only the first rows of the result set:
   - MySQL syntax:
     `SELECT X FROM T_t`
     `ORDER BY I LIMIT 5`
   - Oracle syntax:
     `SELECT * FROM`
     `(SELECT X FROM "T_t" ORDER BY I)`
     `WHERE ROWNUM < 6`

The CORAL approach is to present an identical API for such cases, which allows the development of software components that can be used without any code modification or conditional constructs against multiple relational technologies.

CORAL is expected to be used by applications running on a grid-enabled and distributed environment. It is therefore defining the necessary interfaces for database service indirection, multiple authentication mechanisms (certificate- and user/password pair-based), client-side connection pooling and client-side monitoring.

CORAL is not a general purpose C++ connectivity library, such as ODBC, JDBC, the python or perl DBI. It is neither a system to accomodate all access patterns to relational data. Its scope is restricted to serve primarily the use cases relevant to the data handling and analysis of the LHC experiments.

ARCHITECTURAL CHOICES

The architecture of CORAL is component-based. The overall system consists of a set of abstract interfaces and several implementation components which have no dependency on each other. Data and control flow only through the abstract interfaces in a user application.

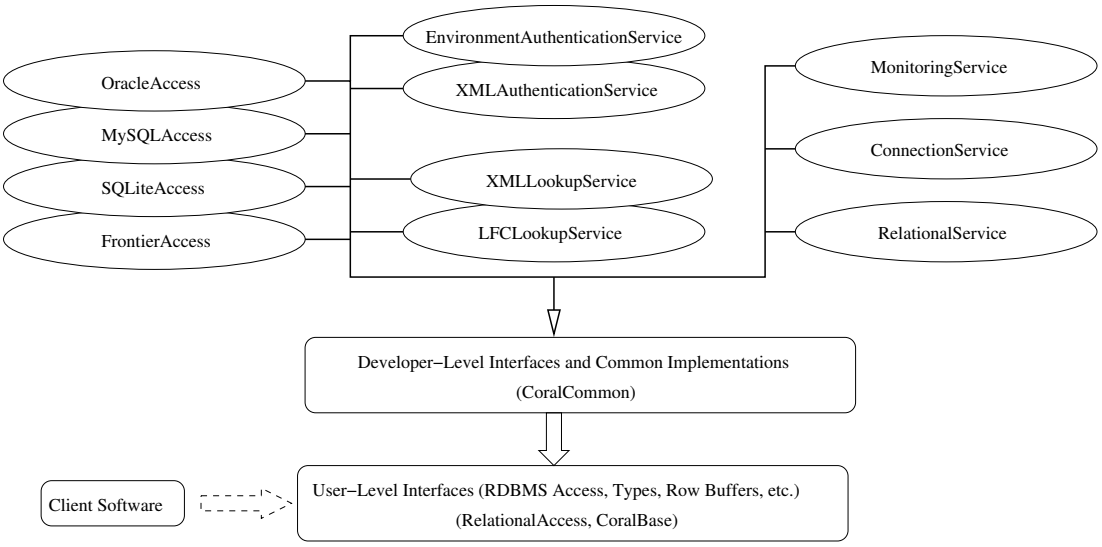

Figure 1: Component architecture of the CORAL framework.

The technology insulation is achieved through the set of the abstract interfaces; a client component depends only on them. The functionality expressed by each interface is minimal but complete. The component architecture and the fact that the functionality expressed by each interface is minimal but complete, enable the parallel and independent development of the various CORAL components, as well as additional and user-provided implementations of the same interfaces. They also facilitate efficient unit and integration testing.

The static view of the CORAL architecture is shown in Fig. 1. The realization of the component architecture has been based on the plugin management and the component framework of the SEAL[4] project. The latter allows the construction of *context hierarchies* onto which the various implementation components are hooked. In this configuration if a component requires the functionality of a particular interface, it simply queries its context tree for an available implementation.

FUNCTIONALITY OF THE PUBLIC API

A user may connect to a database schema by prividing a contact string with the following format:
technology_protocol://hostname:port_number/schema_name
The technology may take values such as *oracle*, *mysql*, *sqlite*, etc. The protocol is optional and used only on embedded technologies such as SQLite[7]. It may take values such as *file*, *http*, etc.

The loaded implementation of the *IRelationalService* interface parses the connection string and searches for available plugins providing the RDBMS functionality for the particular technology. No authentication credentials appear in the connection string. These are provided by the loaded implementation of the *IAuthenticationService* interface to the RDBMS plugin whenever an authentication is attempted. This feature ensures that the location of the actual data, as this is defined by the contact string, is fully decoupled from the authentication parameters and mechanism that are required for accessing them.

A user may enable client-side monitoring for a stated verbosity level. In this case the RDBMS plugin searches for a loaded implementation of the *IMonitoringService* interface and registers information such as begin and end of user sessions, signaling of transaction boundaries and the response times of the various SQL statements that are issued.

Some RDBMS implementations that are used for read-only purposes be based on a web-cache server for the fast delivery of the query results. With the use of the *IWebCacheControl* interface a user may fully control the caching policies in order to ensure consistency of the data that are retrieved.

Alternatively to the format specified above, a user may choose a free format which indicates a logical name for a database service. In this case a query to an implementation of the *ILookupService* interface has to be issued in order to obtain the corresponding connection string that corresponds to an actual service. The best way of doing so is to use an implementation of the *IConnectionService* interface which coordinates the control flow among the above components and provides application-level connection pooling.

After having retrieved a valid handle to a database schema the CORAL API provides the user with a large subset of the functionality that is possible through generic SQL and RDBMS-specific connectivity APIs:

- Schema definition and manipulation operations, such as creating, droping and redefining tables and views, as well as defining indices, keys and constraints. It is also possible to fully describe a given schema and its elements.

  Columns are defined by specifying the corresponding C++ types (eg. *int*, *float*, *double*, *string*) and

some storage hints such as the allocated size for string variables. The translation into the actual SQL types that are eventually used is performed by the RDBMS-specific plugin which is employed for the technology in use.

- Manipulating data in tables such as inserting, modifying and deleting rows. Operations with BLOB types are facilitated through a *Blob* C++ type provided by CORAL.

  The API supports the execution of such operations in bunches, where only the input data change between subsequent iterations. This minimizes the total roundtrips to the database server and the consumption of its resources. These result to an improvement of the overall performance of the application.

- Issuing queries involving one or more tables or views, allowing for row ordering, application of set operations, inclusion of sub-queries and the limiting the rows in the result set. Client-side row caching can be enabled to minimize the total roundtrips to the server.

The CORAL interfaces have been designed such that the RDBMS-specific optimizations or standard "best" practices in RDBMS programming are handled internally by the implementation plugins. Typical examples include the use of *database bind variables* and the efficient use of *server-side cursors*.

IMPLEMENTATION COMPONENTS

Every CORAL release provides:

- Four RDBMS-specific implementations for the interface set related to the functionality of accessing data in a relational database:
  - An implementation to access Oracle databases, based on the Oracle Call Interface (OCI) version 10.2. It is the implementation which complies to the full semantics of the CORAL API.
  - An implementation to access MySQL databases, based on the native client library version 5.0. It is foreseen to be used wherever lower administration resources are available.
  - An implementation to access SQLite files, based on the client library version 3.2. This solution
  - An implementation to access FronTier[8] servers. This is best suited for accessing read-only data from an Oracle database, that are updated infrequently.

  All of the underlying external packages are C libraries. This fact has the benefit that the migration to a new version of the C++ compiler can be done without having to rely on the availability of these libraries for the new compiler version.

- Two implementations of the interface set related to the retrieval of authentication credentials: one based on XML files and another on environment variables.

- An implementation of the interface set responsible for peforming the necessary technology dispatching given a connection string and the available RDBMS-specific implementations found at run time.

- An XML-based implementation of the interface set responsible for peforming logical to physical database service lookup operations.

  An implementation based on LFC[9] is currently under development.

- A simple implementation of the interface set responsible for registering and reporting monitoring events. It is provided mainly in order to serve as an example for demonstrating the implementation of the IMonitoringService interface. The software teams of the LHC experiments are expected to provide implementations which are based on their specific monitoring systems.

- An implementation of the interface set responsible for the client-side connection pooling and the overall system configuration.

DEVELOPMENT AND RELEASE PROCEDURES

The development of CORAL is done incrementally based on individual component tags submitted by the developers. A collection of tags which is validated through a series of integration tests becomes a release candidate.

Frequent internal releases are used for early validation by the experiment software integrators and for providing a reference release for incremental development for the CORAL developers.

Every component provides the corresponding documentation. During the release procedure, the documentation fragments from all packages are automatically assembled to produce the CORAL User Guide and web documentation.

Every public release is installed under afs and binaries are provided for all the standard LCG platforms. Currently these comprise slc3_ia32_gcc323, slc3_ia32_gcc344, slc3_amd64_gcc344, slc4_ia32_gcc345, slc4_amd64_gcc345, win32_vc71 and the corresponding debug versions.

OUTLOOK

As it has already been mentioned CORAL is primarily used for the implementation of the relational components of POOL, such as the Relational File Catalog, Relational Collections and Relational Storage Manager[10], as well

as COOL. The experiments have picked CORAL both indirectly through the use of POOL and COOL, but also directly, especially in on-line applications where there is a strong requirement for minimizing the stack of external software dependencies.

CORAL will continue to be evolving in response to new functional requirements from its main users. However, the focus will be shifted towards deployment-related issues; highest priority will be given to ensuring that the software will contribute towards the success of the forthcoming service challenges of the LCG.

REFERENCES

[1] http://pool.cern.ch/coral/

[2] http://pool.cern.ch/

[3] http://lcgapp.cern.ch/project/CondDB/
A. Valassi et al., "COOL Development and Deployment - Status and Plans", Contribution #337, this conference.

[4] http://seal.cern.ch/

[5] http://www.oracle.com/

[6] http://www.mysql.com/

[7] http://www.sqlite.org/

[8] S. Kosyakov, et. al.,"Frontier: High Performance Database Access Using StandardWeb Components", Proceedings of the CHEP 04 Conference, Interlaken Switzerland, 27 September - 1 October, 2004.

[9] http://wiki.gridpp.ac.uk/wiki/LCG_File_Catalog

[10] G. Govi et al., "POOL Developments for Object Persistency into Relational Databases", Contribution #330, this conference.