Project no. 004758

GORDA

Open Replication Of Databases

Specific Targeted Research Project

Software and Services

Middleware Mapping Report

GORDA Deliverable D4.2

Due date of deliverable: 2005/10/31
Actual submission date: 2005/09/12

Start date of project: 1 October 2004 Duration: 36 Months

INRIA

Revision draft 0.1

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Abstract

This document describes the mapping of the GORDA Architecture and Programming Interfaces to a generic middleware layer that can be used with any database management system that exposes the standard JDBC call-level interface. It aims at providing a description of a middleware implementation, thus reflecting current experience and roadmap for ongoing work on prototype D4.4.

IMPORTANTE NOTICE:

Current draft version aims only at establishing feasibility of the mapping and supporting on-going work on the definition of the Preliminary Architecture and Programming Interfaces to be presented as Deliverable D2.1. Therefore it does not describe final implementation decisions.

Contents

1 Scope

1.1 Objectives

The middleware mapping of the GORDA Architecture and Programming Interfaces (GAPI) has the following primary goals:

- Provide a compatibility layer to deploy GORDA protocols on a wide range of standard database management systems
- Provide full compatibility with the JDBC client interface.

The middleware mapping has the following secondary goals:

- Test the feasibility of implementing a large subset of the GAPI by relying only on a standard client interface JDBC.
- Leverage as much as possible existing open-source components.

1.2 Components

The architecture of the GORDA Middleware Mapping is depicted in Figure 1. Given the assumption that the target DBMS cannot be modified, a reflector cannot be directly attached to it. Instead, a Virtual DBMS is setup and the client is directed to such Virtual DBMS by means of a specific JDBC driver.

The Virtual DBMS is itself implemented in the Controller module and implements the part of the processing pipeline, which includes partial parsing, optimization, concurrency control and scheduling. The rest of the processing pipeline is implemented by the original DBMS which is used by means of a standard JDBC driver.

The reflector interfaces are therefore implemented by the components of the Virtual DBMS. Replication and group communication components are installed within the same controller process and make use of the GORDA Programming Interface thus made available.

As shown in Figure 2, this architecture allows all contexts to be implemented and also several stages of the pipeline. The next section describes the limitations of such implementation.

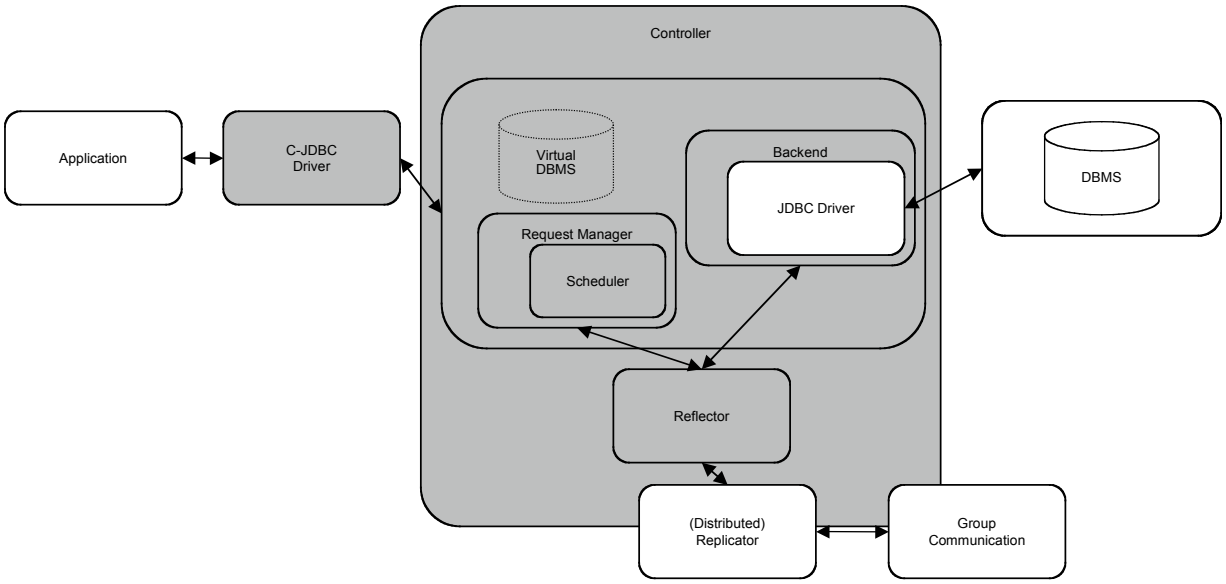

Figure 1- Mapping architecture.

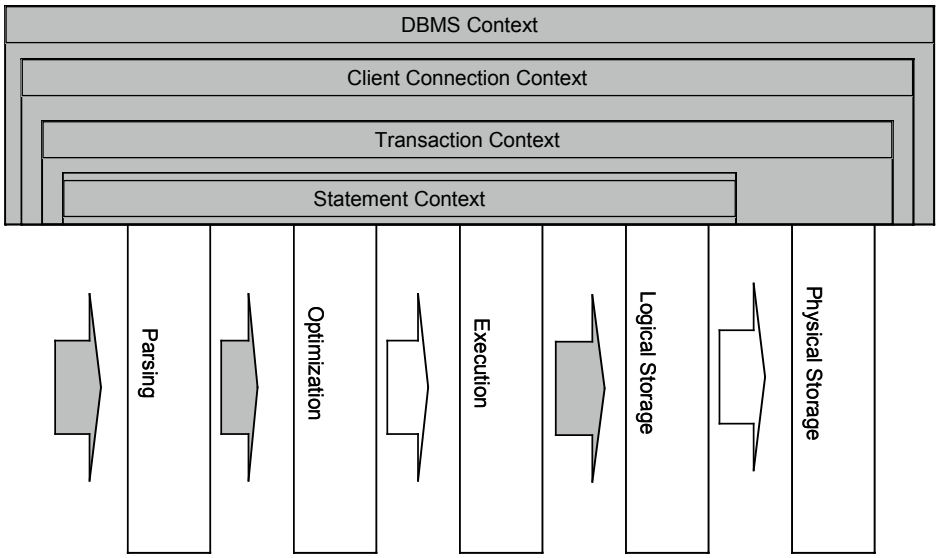

Figure 2 - Mapped interfaces.

2 Implementation

The GORDA Middleware Mapping prototype builds on C-JDBC components and infrastructure. C-JDBC is an open-source packaged developed by project partners which provides a complete and thoroughly tested JDBC interceptor for database clustering based on RAIDb protocols. In this section we describe each of the major components and how the functionality of the reflector is implemented.

2.1 Driver

The C-JDBC driver is a hybrid type 3 and type 4 JDBC driver and it implements the JDBC 2.0 specification. All processing that can be performed locally is implemented inside the C-JDBC driver like in a type 4 JDBC driver. For example, when a SQL statement has been executed on a database backend, the result set is serialized into a C-JDBC driver ResultSet that contains the logic to process the results. Once the ResultSet is sent back to the driver, the client can browse the results locally.

All database dependent calls are forwarded to the C-JDBC controller that issues them on the database native driver like a type 3 JDBC driver. SQL statement executions are the only calls that are completely forwarded to the backend databases.

2.2 Controller

The controller exports virtual databases to users. A virtual database has a virtual name that matches the database name used in the client application. Virtual login names and passwords match also the ones used by the client.

The request manager is a major component of the controller that implements pass-through logic and schedules requests on the backend.

When a request comes from a C-JDBC driver, it is routed to the request manager associated to the virtual database. The request is sent to its native driver through a connection manager that can perform connection pooling. The ResultSet returned by the native driver is transformed into a serializable ResultSet that is returned to the client by means of the C-JDBC driver.

2.3 Statement capture

Parsing is done within the controller and thus repeated later in the real DBMS. Injecting parsed statements means converting them back to SQL text and applying them with JDBC.

2.4 Transaction capture

Transaction boundaries are determined by examining statements received from the client that start or conclude a transaction. The results from execution on the backend are also observed to determine the end of the transaction.

2.5 Read-set extraction

Read-set extraction is done only with very large granularity (table level) by traversing the parsed statement.

2.6 Write-set extraction

Write-set extraction is currently performed with a very large granularity (table level) by traversing the parsed statement. This approach is also of limited usefulness when considering stored procedures, where the SQL statement does not contain table names.

Current work focus on obtaining write-sets with fine granularity by setting up update, delete and notify trigger in the underlying database as has been implemented in the Middle-R and MADIS research prototypes. This approach does not have the limitations of observing parsed statements regarding extraction of write-sets when stored procedures are involved.

2.7 Applier

Application of updates is initiated by the request manager. In detail, the *scheduler* is responsible for ordering the requests for execution. Consecutive write queries may be aggregated in a batch update so that they perform better. Once the request scheduler processing is done, the requests are sequentially ordered and then submitted to the backend driver.