Project no. 004758

GORDA

Open Replication Of Databases

Specific Targeted Research Project

Software and Services

In-core Mapping Report

GORDA Deliverable D4.1

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 the open-source PostgreSQL database management system. It aims at providing an example of an in-core implementation, thus reflecting current experience and roadmap for ongoing work on prototype D4.5.

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.

Chapter 1

Scope

This Chapter describes the scope of the PostgreSQL prototype, regarding its objectives and the components of the GORDA Architecture and Programming Interfaces actually being implemented.

1.1 Objectives

The in-core mapping of the GORDA Architecture and Programming Interfaces (GAPI) to the open source PostgreSQL database management has two primary goals:

- Allow the implementation of a wide range of group communication based database replication protocols thus requiring an extensive mapping of the GAPI, in particular, in the storage phase.
- Allow full compatibility with all PostgreSQL clients interfaces and tools.

The PostgreSQL mapping has also two secondary goals:

- Demonstrate the feasibility of effciently extracting read-sets.
- Modifications to the core PostgreSQL source code although unavoidable, should be as little intrusive as possible in order to facilitate dissemination by proposing patches for acceptance by the PostgreSQL open source community.

1.2 Components

The mapping to PostgreSQL is outlined in Figure 1.1. It assumes an additional Container process that holds replication and communication components as well as part of the abstract reflector component. PostgreSQL and the container communicate by two means: a plain BSD socket using a simple protocol designed specifically for this case; and through a JDBC connection.

Communications between applications and PostgreSQL are performed using a legacy driver provided by the database vendor/supplier thus ensuring compatibility with all existing clients and tools.

As show in Figure 1.2 the current mapping of the GORDA Programming Interfaces includes all contexts and the two main stages.

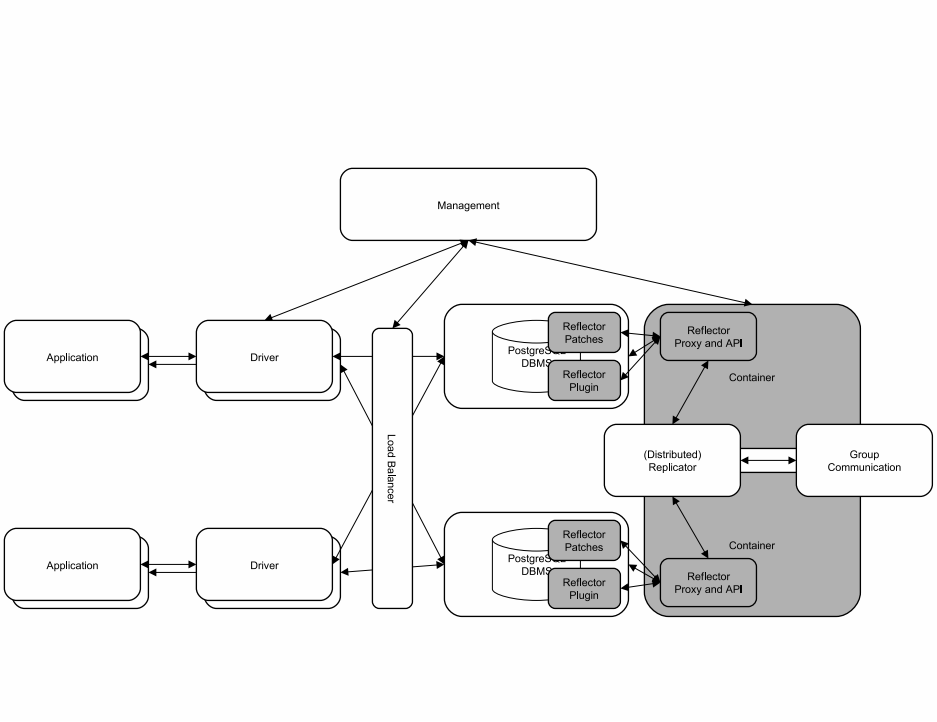

Figure 1.1: Mapping architecture.

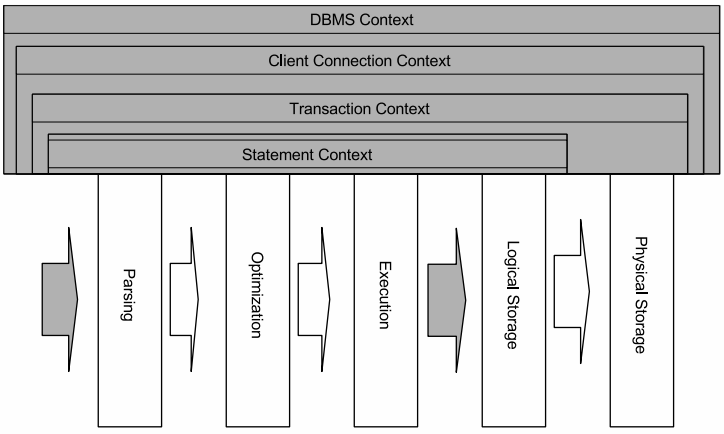

Figure 1.2: Mapped interfaces.

Chapter 2

Implementation

This part of the document describes the elements of the in-core solution. It starts by describing the container and the reflector links. Right after, it describes the functionalities provided by the plugins and patches inside the PostgreSQL, namely the ability to capture transactions, statements, read and write operations. Then, it describes how the reflector proxy handles remote and local updates.

2.1 Container

The container is implemented in Java and is configured by an XML file both regarding which capabilities are actually offered by the reflector as well as what replication and communication components are loaded. The reflector connects to the database and handles metadata related operations, receives transaction's information, statements and read and written data captured by the reflector's plugins and patches (called Reflector PostgreSQL).

The container handles its component's life-cycle, the number of threads that can simultaneously execute inside a component and delivers the events produced externally to the appropriate components. Specifically, it starts, pause, resume and stops the components whenever requested. If a component is declared as non-reentrant in the deployment file, the container must ensure that only one thread can be executing an instance at any time. Furthermore, it also delivers remote events generated by the reflector components in PostgreSQL.

2.2 Reflector link

The custom link protocol is provided as a library that can be used from plugins in triggers and directly from the patched source code. It defines a simple protocol designed specifically for exchanging data for the reflector. The link can be used for synchronization when the PostgreSQL process blocks waiting for data, thus blocking the corresponding transaction.

Figure 2.1 depicts the protocol during. In detail:

- Upon transaction begin, the PostgreSQL side identifies the current transaction and database version identifiers.
- For each statement, the container is notified and acknowledges it for execution.

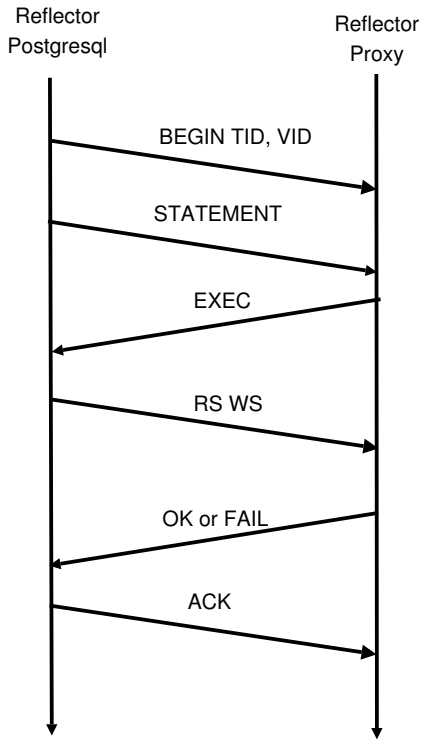

Figure 2.1: Socket Protocol

- When the transaction enters commiting stage, the read sets and write sets are sent. Execution is blocked until commit is confirmed or denyed.
- After commit has been confirmed, and the DBMS can ensure that the transaction is not aborted (although it may not be durable yet if the server crashes) the commit is acknowledged.

2.3 Statement capture

Statement capture is achieved by patching the PostgreSQL immediately after receiving the text from the client. Execution is suspended until acknowledged, implicitly or explicitly, through the reflector.

2.4 Transaction capture

Transaction capture is achieved with a collection of patches for transaction triggers. These patchs create hooks for the following set of events: transaction begin, commit, and rollback. They are implemented as global triggers, associated with a database instead of with a relation.[1]

During normal operation, whenever a transaction begins (implicitly or explicitly) the begin hook is called. The commit hook is called just after the client requests commit (implicitly or explicitly) and when the trigger function can still rollback the transaction. The rollback trigger is called whenever a transaction is aborted (implicitly or explicitly). The modifications in the abort procedure are necessary to release resources remotely and locally held.

[1] A preliminary version has been made available to the community in *http://gorda.di.uminho.pt/community/pgsqlhooks/*.

Surprisingly, the efforts to build such triggers are not so high because most DBMSs have non-standard APIs that provide similar functionalities. Although the motivation for this development was to support synchronous replication without intrusive modification of the PostgreSQL, it might also be useful to projects such as materialized views.

2.5 Write-set extraction

Th write sets are extracted into a memory relation by setting up trigger on update, insert and delete on relevant tables. Once the commit statement is set to execute, the collected partial write sets are assembled into the final write set and is sent to the reflector proxy by using the socket link.

2.6 Read-set extraction

The read-set extraction does not have triggers provided by the DBSMs and must be done by using a patch that may be turned on/off at compile time. Briefly, the read set extraction is developed using iterators inserted in the execution plan.[2] The iterators define extraction points where the read data is gathered and dumped into memory. Thus, during the execution, the database calls "the read set iterator" to retrieve a next tuple and the "read set iterator" calls the iterator below it to retrieve the tuple. So one, the tuples that passes through the "read set iterator" are dumped into memory.

2.7 Applier

The applier process is key to the performance of replication protocols built uppon the GAPI and must address the issues of parallel apply and transaction priority. The current prototype implementation uses standard JDBC connections to handle remote updates and simply acknowledge the local database by using the reflector socket for local transactions.

Upon applying remote transactions, the order determined by the replication protocol must be followed. If t preceeds t' and both update the same item i, the last value of i must be the one given by t'. Basically, the applier must guarantee that the remote transactions inside the DBMS do not have their update order switched by the database internal scheduler. To solve this issue, the applier process may use two different approaches. In the first one, it only submits remote updates after knowing for sure that either the previous is about to release its locks or that they do not access common tuples. The second approach considers transaction batching and groups remote updates into a batch transaction. Note that when batching transaction one cannot include local update transactions because, these are not delivered using JDBC.

Finnally, we must take into account that remote transactions should not wait for local executing transactions. If a remote transaction waits a local transaction to execute in order to proceed, the overall performance is reduced. Thus, the applier assumes that remote transactions have higher priority when compared to locally executing transactions and hence resources acquired by local transactions must be released and granted the remote transaction, whenever the former blocks the later.

[2] A technical report describing the process in detail is available in the Library section of the project web site.