GORDA: An Open Architecture for Database Replication*

Alfrânio Correia Jr.
University of Minho

José Pereira
University of Minho

Luís Rodrigues
University of Lisboa

Nuno Carvalho
University of Lisboa

Ricardo Vilaça
University of Minho

Rui Oliveira
University of Minho

Susana Guedes
University of Lisboa

1 Introduction

Database replication has been a common feature in database management systems (DBMSs) for a long time. In particular, asynchronous or lazy propagation of updates provides a simple yet efficient way of increasing performance and data availability [5] and is widely available across the DBMS product spectrum. High end systems additionally offer sophisticated conflict resolution and data propagation options as well as, synchronous replication based on distributed locking and two-phase commit protocols.

There has however been a growing interest in third party replication solutions. Namely, recent research in database replication based on group communication has led to novel algorithms that achieve strong replica consistence without the overhead of traditional synchronous replication [6, 13]. The large demand for 3-tier systems and Web applications, with read-intensive database workloads, has increased the interest on clustering middleware such as Sequoia (formerly ObjectWeb C-JDBC) [3] for cost-effective scalability and higher availability. Finally, replication has also been proposed as the means to seamlessly combine special purpose query processing abilities from multiple DBMSs [9].

The lack of native support from database vendors for third party replication forces proponents of those solutions to either modify the database server or to

*Parts of this report were published in the Proceedings of the 6th IEEE Internacional Symposium on Network Computing and Applications (NCA '07), Boston, MA, USA. 2007.

develop, in middleware, a server wrapper that intercepts client requests. Unfortunately, the modification of the database server is hard to maintain and port, and, in many cases, simply impossible due to unavailability of source code. On the other hand, a middleware wrapper, which implements replication and redirects requests to the actual underlying DBMS, represents a large development effort and introduces an additional communication step and thus some performance overhead.

We address this problem with the GORDA architecture and programming interface (GAPI), that enables different replication strategies to be implemented once and deployed in multiple DBMSs. This is achieved by proposing a reflective interface to transaction processing instead of relying on client interfaces or ad-hoc server extensions. The proposed approach is thus cost-effective, in enabling reuse of replication protocols or components in multiple DBMSs, as well as potentially efficient, as it allows close coupling with DBMS internals.

The contribution of the paper is therefore as follows. First we propose a reflective model of transaction processing. Then we implement and evaluate the proposed architecture in three representative DBMS architectures, namely, Apache Derby 10.2 [1], PostgreSQL 8.1 [10] and Sequoia 2.9 [3]. These implementations illustrate a spectrum of different ways to implement the proposed interface.

The rest of this paper is structured as follows. Section 2 discusses implementation strategies and reflection facilities in DBMSs. Section 3 introduces the GORDA architecture and programming interface (GAPI). Section 4 discusses implementation strategies and tradeoffs. Section 5 concludes the paper.

2 Background

This section surveys different architectures to implement replication in concrete systems as well as existing reflection facilities that have been proposed for DBMSs.

2.1 Implementation of Replication

Multiple architectures have been used to interface replication protocols with DBMSs. In the following, we discuss their main categories.

Replication implemented as a normal client. In this approach, both the application and the replication protocol interact with the DBMS independently and exclusively through client interfaces, e.g. JDBC. This makes the replication protocol portable and can be very efficient, specially when the code resides within the server using a server side client interface. This strategy is however very limited as the replication protocol is confined to propagate the updates performed by application initiated transactions without any control over their execution. As a consequence,

the inability to suspend a third party initiated transaction and synchronously update the database replicas only allows to perform asynchronous replication. An example of this approach is Slony-I [11], which provides asynchronous replication of PostgreSQL.

Replication implemented as a server wrapper. These implementations rely on a wrapper to the database server that intercepts all client requests by sitting between clients and the server. An example of an application of this approach is Sequoia [3]. The middleware layer presents itself to clients as a virtual database. Compared to the previous approach, implemented as regular DBMS client, this solution offers improved functionality, as it is able to intercept, parse, delay, modify, and finally route statements to target database servers. Nonetheless, it imposes additional overhead, as it duplicates some of the work of the database server.

Replication implemented as a server patch. This solution requires changes to the underlying database server. This approach has been used to implement certification-based replication protocols such as the Postgres-R prototypes [6, 13]. Given that it is implemented in the DBMS kernel, the replication protocol has an easy access to control information such as read and written tuple sets, transaction lifecycle events, etc. It has however the disadvantage of requiring access to the database server source code. It also imposes a significant obstacle to portability, not only to the multiple database servers but also as the implementation evolves.

Replication using custom interfaces. Database servers that natively support asynchronous replication usually do so using a well defined and published, although proprietary, interface. This allows some customization and integration with third party products when asynchronous propagation is desired but of little use otherwise. An example of the approach is the Oracle Streams interface [2] which is based on existing standards and confined to asynchronous propagation of updates.

2.2 Reflection and Database Management Systems

Reflective systems allow a computation to be inspected or altered by interacting with a representation of itself, i.e. its own reflection [7]. For instance, in an object oriented system, invocations of methods on objects can be reflected as objects in order to be inspected and manipulated. To be distinguished from ordinary or *base objects*, reflected objects are usually called *meta-objects*. It has been shown that reflection is key to extensible systems that can evolve.

Reflective capabilities have been offered by DBMSs for a long time, in which computation on the relational model is reflected back to relation entities. The most prevalent are *triggers*, which allow update operations to be intercepted by user defined procedures, and *log mining*, which expose committed transactions as read-only relational tables. Both have been used to collect updates for the purpose of

replication. The usage of such reflective features for self-tuning has been proposed in [8].

3 Reflective Architecture and Interfaces

Target Reflection Domain. Existing reflective facilities in DBMSs are targeted at application programmers using a relational model. Its domain is therefore the relational model itself. With it, one can intercept operations that modify relations by inserting, updating, or deleting tuples, observe the tuples being changed and then enforce referential integrity by vetoing the operation (all at the meta-level) or by issuing additional relational operations (base-level).

In contrast, a replication protocol is concerned with details that are not visible in the relational model, such as modifying query text to remove non-determinism or the precise scheduling of updates to achieve a given isolation level. For instance, one may be interested in intercepting a statement as it is submitted, whose text can be inspected, modified (meta-level) and then re-executed, locally or remotely, within some transactional context (base-level).

Therefore, a more expressive target domain is required. We select an object-oriented concurrent programming environment. Specifically, we use the Java platform, but any similar language would do. The fact that a series of activities (e.g. parsing) is taking place on behalf of a transaction is reflected as a transaction object, which can be used to inspect the transaction (e.g. wait for it to commit) or to act on it (e.g. force a rollback).

Meta-level code can register to be notified when specific events occur. For instance, when a transaction commits a notification is issued, containing a reference to the corresponding transaction object (meta-level). Actually, handling notifications is the way that meta-level code dynamically acquires references to meta-objects describing the on-going computation.

Processing Stages. The usefulness of the meta-level interface depends on what is exposed as meta-objects. If a very fine granularity is chosen, the interface cannot be easily mapped to different DBMSs and the resulting performance overhead is likely to be high. On the other hand, if a very large granularity is chosen, the interface may expose too little to be useful.

Therefore, we abstract transaction processing as a pipeline as it is commonly accepted [4] (Fig. 1). In detail, the *Parsing* stage parses raw statements received thus producing a parse tree. The parse tree is transformed by the *Optimization* stage according to various optimization criteria, heuristics and statistics into an execution plan. The *Execution* stage executes the plan and produces object-sets. The *Logical Storage* stage deals with mapping from logical objects to physical storage. Finally,

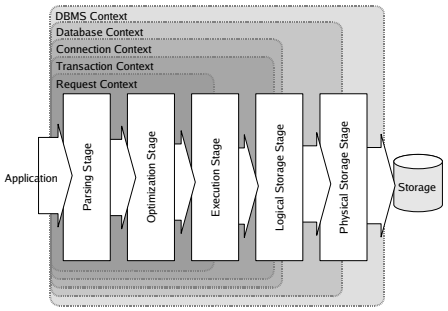

Figure 1: Major meta-level interfaces: processing stages and contexts.

the *Physical Storage* stage deals with block input/output and synchronization.

In general, one wants to issue notifications at the meta-level whenever computation proceeds from one stage to the next. For instance, when write-sets are issued at the execution stage, a notification is issued such that they can be observed. The interface thus exposes meta-objects for each stage and for data that moves between them.

In contrast, previous approaches assume that reflection is achieved by wrapping the server and intercepting requests as they are issued by clients [12]. By choosing beforehand such implementation approach, one can only reflect computation at the first stage, i.e. with a very large granularity. Exposing further details requires rewriting large portions of DBMS functionality at the wrapper level. As an example, Sequoia [3] does additional parsing and scheduling stages at the middleware level.

Processing Contexts. The meta-interface exposed by the processing pipeline is complemented by nested context meta-objects, also shown in Fig. 1. These show on behalf of whom some operation is being performed. In detail, the *DBMS* and *Database* context interfaces expose metadata and allow notification of lifecycle events. *Connection* contexts reflect existing client connections to databases. They can be used to retrieve connection specific information, such as user authentication or the character set encoding used. The *Transaction* context is used to notify events related to a transaction such as its startup, commit or rollback. Synchronous event handlers available here are the key to the synchronous replication protocols presented in this document. Finally, to ease the manipulation of the requests within a connection to a database and the corresponding transactions one may use the *Request* context interface.

Events fired by processing stages refer to the directly enclosing context. Each

context has then a reference to the next enclosing context and can enumerate all enclosed contexts. This allows, for instance, to determine all connections to a database or which is the current active transaction in a specific connection. Notice that some contexts are not valid at the lowest abstraction levels. Namely, it is not possible to determine on behalf of which transaction a specific disk block is being flushed by the physical stage.

4 Implementation

This section discusses how the proposed architecture and interface is implemented in three different systems, namely, Apache Derby, PostgreSQL and Sequoia. These systems represent different tradeoffs and implementation decisions and are thus representative of what one should expect when implementing the GORDA architecture and programming interfaces, namely, in terms of lines of code required.

Apache Derby 10.2. Apache Derby 10.2 [1] is a fully featured DBMS with a small footprint developed by the Apache Foundation and distributed under an open source license. It is also distributed as IBM Cloudscape and in JDK 1.6 as JavaDB. It can either be embedded in applications or run as a standalone server. The prototype implementation of the GORDA interface takes advantage of Derby being natively implemented in Java to load meta-level components within the same JVM and thus closely coupled with the base-level components. Furthermore, Derby uses a different thread to service each client connection, thus making it possible that notifications to the meta-level are done by the same thread and thus reduce to a method invocation, which has negligible overhead. This is therefore the preferred implementation scenario. The current prototype exposes all context objects and the parsing and execution objects, as well as calling between base-level and meta-level. The effort required to implement such subset of the interface can roughly be estimated by the amount of lines changed in the original source tree as well as the amount of new code added: the size of Apache Derby is 514941 lines, where 29 files were changed by inserting 1250 lines and deleting 25 lines; 9464 lines of code were added in new files.

PostgreSQL 8.1. PostgreSQL 8.1 [10] is a fully featured DBMS distributed under an open source license and is written in C. The major issue in implementing the proposed architecture is the mismatch between its concurrency model and the multi-threaded meta-level runtime. PostgreSQL 8.1, as all previous versions, uses multiple single-threaded operating system processes for concurrency. This is masked by using the existing PL/J binding to Java, which provides a single standalone Java virtual machine and inter-process communication. Unfortunately, this approach imposes an inter-process remote procedure call overhead on all commu-

nication between base and meta-level. The current prototype exposes all context objects and the parsing and execution objects, as well as calling between base-level and meta-level. The effort required to implement was as follows. The size of PostgreSQL is 667586 lines and the PL/J package adds 7574 lines of C code and 16331 of Java code, where 21 files were changed by inserting 569 lines and deleting 152 lines. Additionally, 1346 lines of C code and 11512 lines of Java code were added in new files.

Sequoia 2.9. Sequoia [3] is a middleware package for database clustering built as a server wrapper. It works by creating a virtual database at the middleware level, which reimplements part of the abstract transaction processing pipeline and delegates the rest to a backend database. The current prototype exposes all context objects and the parsing and execution objects, as well as calling from meta-level to base-level with a separate connection. It does not allow calling from base-level to meta-level, as execution runs in a separate process. It can however be implemented by directly intercepting statements at the parsing stage. The effort required to implement was as follows: the size of the generic portion of Sequoia is 137238 lines, which includes the server and the JDBC driver. Additionally, Sequoia contains 29373 lines which implement pluggable replication and partitioning strategies, that we don't use. In the server, 7 files of code were changed by inserting 180 lines and deleting 23 lines and 8625 lines of code were added in new files.

Discussion. The first interesting conclusion is that the proposed interfaces have been implemented in such scenarios with a consistently low intrusion in the original source code. This translates in low effort both when implementing it but also when maintaining the code when the DBMS evolves. Note also that a significant part of the additional code is shared, as it is the definition of the interfaces (6144 lines). There is also a firm belief that most of the rest of the code could also be shared, as it performs the same container and notification support functionality. This has not happened as each implementation was developed independently and concurrently. A second interesting conclusion is that the amount of code involved in developing a server-wrapper is in the same order of magnitude as a full-featured database (i.e. hundreds of Klines of code). It is further evidence that relying on this strategy is not cost-effective. In comparison, implementing the proposed interface involves 100 times less effort as measured in lines of code.

5 Conclusions

Recent developments in database replication and clustering have been placing new demands on DBMS interfaces. Current attempts to satisfy these demands, such as patching the database kernel or building complex wrappers, require a large de-

velopment effort in supporting code, cause avoidable performance overhead, and reduce the portability of replication middleware. Ultimately, that lack of appropriate interfaces to support third-party replication protocols is a serious obstacle to research and innovation in replicated databases. In this paper we address this issue by proposing a reflective architecture and interface that exposes transaction processing such that it can be observed and modified by external replication protocols. Instead of creating ad-hoc hooks for known replication protocols, we rely on an well known abstraction for transaction processing which should provide a solid foundation for extensibility. Prototypes described in this paper are published as open source, can be downloaded from the GORDA project's home page (`http://gorda.di.uminho.pt`), examined, and benchmarked. A modular replication framework that builds on the proposed architecture and thus runs on PostgreSQL, Apache Derby, or any DBMS wrapped by Sequoia, is also available, as well as an extended version of this paper with preliminary performance results.

References

[1] Apache DB Project. Apache Derby version 10.2. http://db.apache.org/derby/, 2006.

[2] N. Arora. Oracle Streams for near real time asynchronous replication. In *VLDB Conference WDIDDR*, 2005.

[3] Continuent. Sequoia version 2.9. http://sequoia.continuent.org, 2006.

[4] H. Garcia-Mollina, J. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[5] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *ACM SIGMOD*, 1996.

[6] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *VLDB Conference*, 2000.

[7] G. Kiczales. Beyond the black box: Open implementation. *IEEE Software*, 13:10–11, 1996.

[8] P. Martin, W. Powley, and D. Benoit. Using reflection to introduce self-tuning technology into dbmss. In *IEEE IDEAS*, 2004.

[9] C. Plattner, G. Alonso, and M. Özsu. Extending DBMSs with satellite databases. VLDB Journal, To appear.

[10] PostgreSQL Global Development Group. Postgresql version 8.1. http://www.postgresql.org/, 2006.

[11] PostgreSQL Global Development Group. Slony-I version 1.1.5. http://slony.info, 2006.

[12] J. Salas, R. Jimenez-Peris, M. Patino-Martinez, and B. Kemme. Lightweight reflection for middleware-based database replication. In *IEEE SRDS*, 2006.

[13] S. Wu and B. Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *IEEE ICDE*, 2005.