GORDA: An Open Architecture for Database Replication*
Extended Report

Alfrnio Correia Jr.
University of Minho

Jos Pereira
University of Minho

Lus Rodrigues
University of Lisboa

Nuno Carvalho
University of Lisboa

Ricardo Vilaa
University of Minho

Rui Oliveira
University of Minho

Susana Guedes
University of Lisboa

Abstract

Recently, third party solutions for database replication have been enjoying an increasing popularity. Such proposals address a diversity of user requirements, namely preventing conflicting updates without the overhead of synchronous replication; clustering for scalability and availability; and heterogeneous replicas for specialized queries. Unfortunately, the lack of native support from database vendors for third party replication forces implementors to either modify the database server, restricting portability, or to develop a middleware wrapper, which causes a performance overhead. This paper addresses this problem with a novel architecture and programming interface for replication, such that different strategies can be efficiently implemented on

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

any compliant database management system in a cost-effective manner. The contribution is two-fold. First we propose a reflective model of transaction processing and explain how it can be used to achieve replication. Then we implement the proposed architecture in Apache Derby, PostgreSQL, and Sequoia and evaluate the PostgreSQL implementation with the TPC-W industry standard benchmark.

1 Introduction

Database replication has been a common feature in database management systems (DBMS) for a long time. In particular, asynchronous or lazy propagation of updates provides a simple yet efficient way of increasing performance and data availability [6] and is widely available accross 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 proposed novel algorithms that achieve strong replica consistence without the overhead of traditional synchronous replication [8, 14, 10, 22]. The large demand for 3-tier systems and Web applications, with read-intensive database workloads, have increased the interest on clustering middleware such as Sequoia (formerly ObjectWeb C-JDBC) [4] 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 DBMS [15].

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 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[1] architecture and program-

[1]The work reported here is being developed in the context of an EU funded research

ming interface (GAPI), that enables different replication strategies to be implemented once and deployed in multiple database management systems. This is achieved by proposing a reflective interface to transaction processing instead of relying on client interfaces or ad-hoc interface 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 and illustrate how it can be used to implement several representative replication strategies. Then we implement and evaluate the proposed architecture in three representative DBMS architectures, namely, Apache Derby 10.2 [1], PostgreSQL 8.1 [17], and Sequoia 2.9 [4]. These implementations illustrate a spectrum of different ways to implement the proposed interface. Finally we experimentally evaluate the performance of the proposed approach.

The rest of this paper is structured as follows. Section 2 discusses implementation strategies and reflection facilities in database management systems. Section 3 introduces the GORDA architecture and programming interface (GAPI). The application to several replication protocols is presented in Section 4. Section 5 discusses implementation strategies and trafeoffs. Section 6 evaluates the resulting performance and Section 7 concludes the paper.

2 Background

In this section, we survey different architectures to implement replication in concrete systems as well as existing reflection facilities that have been proposed for database management systems.

project, GORDA (Open Replication of DAtabases, `http://gorda.di.uminho.pt`), that intends to foster database replication as a means to address the challenges of trust, integration, performance, and cost in current database systems underlying the information society. The GORDA project has a mix of academic and industrial partners, including U. do Minho, U. della Svizzera Italiana, U. de Lisboa, INRIA Rhône-Alpes, Continuent Oy, and MySQL AB.

2.1 Implementation of Replication

Multiple architectures have been used to interface replication protocols with DBMS. 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 the 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 [18], which provides asynchronous replication of PostgreSQL. Typically, these solutions resort to installing triggers in the underlying DBMS in order to update meta-information and gather updates.

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 [4]. 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. The development of such infrastructure represents also a large undertaking, and prevents clients to connect directly to database servers using native privileged interfaces. It also has to rely on triggers, installed in the underlying DBMS to capture relevant control information such as updates.

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 [8, 22]. 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 [9]. 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 database management systems 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 commited 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 [12].

Custom interfaces to extract write-sets for different replication strategies can also be regarded as reflective interfaces [21]. These are however tightly coupled to the semantics of specific replication protocols, i.e., on how buffering is performed and how they are synchronized with transaction commit events, or to specific implementation strategies, i.e. assuming server wrapper. Their usage as a base for portable replication implementations is thus limited.

Extensive research has also been done on aspect oriented databases, which builds on reflection [19]. This effort is however not useful to replication as if focus on reflecting static aspects such as integrity constraints, and not dynamic aspects such as requests and transactions. It is used, for instance, to allow the database schema to evolve while maintaining backwards compatibility with existing applications.

3 Reflective Architecture and Interfaces

In this section we outline the GORDA architecture and programming interface, as well as the underlying rationale. Implementation issues are discussed in Section 5.

Target Reflection Domain Existing reflective facilities in database management systems are targetted 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 nondeterministim 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 reexecuted, 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 aquires 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 [5] (Fig. 1). In detail, the *Parsing* stage parses raw state-

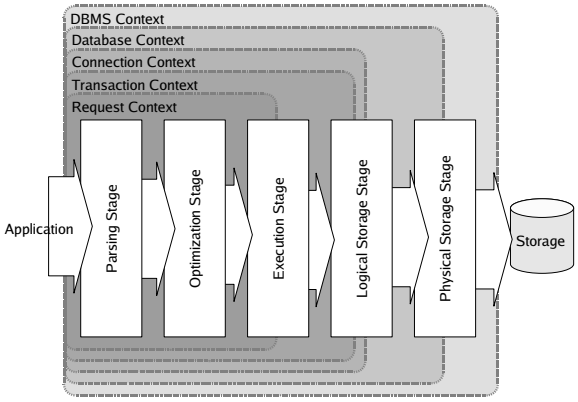

Figure 1: Major meta-level interfaces: processing stages and contexts.

ments received thus producing a parse tree. The parse tree is transformed by the *Optimization* stage according to various optimization criteria, heuristics and statistics to 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, 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 [21]. 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 [4] 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.

Furthermore, replication protocols can attach an arbitrary object to each context. This allows context information to be extended as required by each replication protocol. As an example, when handling an event fired by the first stage of the pipeline, signaling the arrival of a statement in textual format, the replication protocol gets a reference to the enclosing transaction context. It can then attach additional information to that context. Later, when handling an event signaling the readiness of parts of the write-set, the replication protocol follows the reference to the same transaction context to retrieve the information previously placed there.

Base-level and Meta-level Calls An advantage of reflection is that base- and meta-level code can be freely mixed, as there is no inherent difference between base- and meta-objects. This happens also in the proposed interface, albeit with some limitations.

In detail, a direct call to meta-level code can be forced by the application programmer by registering it as a native procedure and then using the CALL SQL statement. This causes a call to the meta-level code to be issued from the base-level code within the *Execute* stage. The target procedure can then retrieve a pointer to the enclosing *Request* context and thus to all relevant meta-interfaces. The reason for allowing this only from the *Execute* stage

is simplicity, as this is inherently supported by any DBMS, and does not seem to impact generality. A second reason is that this is where the pipeline can be reentered, should the meta-level procedure need to callback into the base-level.

Meta-level code can callback into base level in two different situations. The first is within a direct call from base-level to issue statements in an existing enclosing request context. This can be achieved using the JDBC client interface by looking up the "jdbc:default:connection" driver, as is usually done in Java procedures. The second option is to use the enclosing *Database* context to open a new base-level connection to the database. The reason for allowing base-level to use the JDBC interface is again simplicity, as this avoids the need to have interfaces that build contexts and inject external data into internal structures. This may however have an impact on performance, and is thus the subject of future work as discussed in Section 7.

A second issue when considering base-level calls is whether these also get reflected. The proposed option is to disable reflection on a case-by-case basis by invoking an operation on context meta-objects. Therefore, meta-level code can disable reflection for a given request, a transaction, a specific connection or even an entire database. Actually this can be used on any context meta-object and thus for perfomance optimization. For one, consider a replication protocol, that is notified that a connection will only issue read-only operations, and thus ceases monitoring them.

A third issue is how base-level calls issued by meta-level code interact with regular transaction processing regarding concurrency control. Namely, how are conflitcs that require rollback resolved, namely, in multi-version concurrency control where the first commiter wins or, more generally, when resolving deadlocks. The proposed interface solves this by ensuring that transactions issued by the meta-level do not abort in face of conflicts with regular base-level transactions. Given that replication code running at the meta-level has a precise control on which base-level transactions are scheduled, and thus can prevent conflicts among those, has been sufficient to solve all considered use cases. The simplicity of the solution means that implementation within the DBMS resulted in a small set of localized changes.

Design Patterns The design of meta-level interfaces leverages patterns that have proven useful in object oriented middleware. The first is the faade, which allows inspection of diverse data structures through a common inter-

face. A very well known example is the ResultSet, which allows results to be stored in a DBMS native format. The alternative is the potentially expensive conversion to a common format such as XML. The proposed architecture suggests using this for most of the data that is conveyed between processing stages (e.g. object sets).

The second is the inversion-of-control pattern, which eases deployment of software components. In detail, meta-objects such as transactions are exposed to an object container, which is configured with replication components. The container is then responsible for injecting the required meta-objects into each replication component during initialization.

The third pattern is the container managed concurrency. The container implementation schedules event notifications according to performance and correctness criteria. For instance, by ensuring that no two transactions commit notifications are issued concurrently, implicitly exposes a commit order. Notification of available write-sets of two different transactions can be issued concurrently.

4 Case Studies

This section describes how the reflector interface is used to implement state-machine, primary-backup and certification-based replication protocols and also how it might be used to delevop plugins such as tracers and debuggers.

4.1 Primary-Backup

Overview In the primary-backup approach to replication, also called passive replication [13], update transactions are executed at a single master site under the control of local concurrency control mechanisms. Updates are then captured and propagated to other sites. Asynchronous primary-backup is the standard replication in most DBMSs and third-party offers. An example is the Slony-I package for PostgreSQL [18]. Implementations of the primary-backup approach differ whether propagation occurs synchronously within the boundaries of the transaction or, most likely, is deferred and done asynchronously. The latter provides optimum performance when synchronous update is not required, as multiple updates can be batched and sent in the background. It also tolerates extended periods of disconnected operation. The main advantage of this approach is that it can easily cope with non-

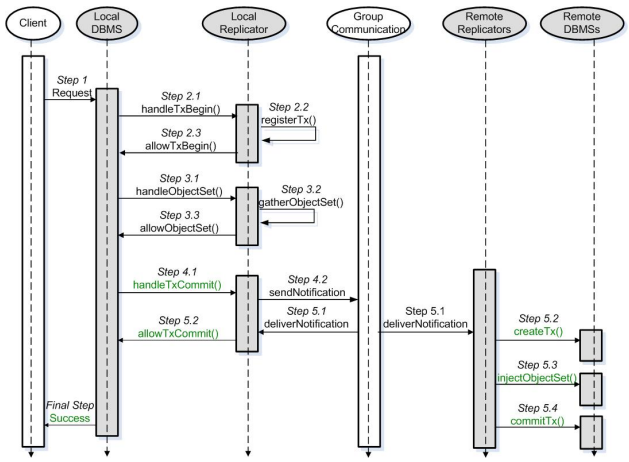

Figure 2: Primary-backup replication.

deterministic servers. A major drawback is that all updates are centralized at the primary and little scalability is gained, even if read-only transactions may execute at the backups. It can only be extended to multi-master by partitioning data or defining reconciliation rules for conflicting updates. The Primary-Backup protocol has a primary replica where all transactions that update the database are executed. Updates are either disseminated in transaction's boundaries (i.e., synchronous replication) or periodically propagated to other replicas in background (i.e., asynchronous replication).

Reflector Components Used Synchronous primary-backup replication requires the component that reflects the Transaction context to capture the moment where the transaction starts executing, commits, or rollbacks at the primary. It will also need the object set provided by the Execution stage to extract the write set of a transaction from the primary and insert it at the backup replicas.

Replicator Execution The execution of a primary-backup replicator is depicted in Figure 2. We start by describing the synchronous variant. It consists of the following steps. *Step 1:* Clients send their requests to the primary replica; *Step 2:* When a transaction begins, the replicator at the primary is notified, registers information about this event, and allows the primary replica to proceed; *Step 3:* Right after processing a SQL command

the database notifies the replicator through the Execution stage component sending an *ObjectSet*. Roughly, the ObjectSet provides an interface to iterate on a statement's result set (e.g.,write set). Specifically, in this case, it is used to retrieve statement's updates which are immediately stored in a in-memory structure with all other updates from the same transaction context; *Step 4:* When a transaction is ready to commit, the transaction context component notifies the replicator of the primary. The replicator atomically broadcasts the gathered updates to all backup replicas (this broadcast should be *uniform* [3]); *Step 5:* The write set is received at all replicas. On the primary, the replicator allows the transaction to commit. On the backups, the replicator injects the changes in the DBMS; *Final Step:* After the transaction execution, the primary replica replies to the client. An asynchronous variant of the algorithm can be achieved by postponing Step 4 (and, consequently, Step 5) for a tunable amount of time.

4.2 State-machine

Overview The state-machine approach, also called active replication [13], is a decentralized replication technique. Consistency is achieved by starting all replicas with the same initial state and, subsequently, receiving and processing the same exact sequence of client requests. Examples of this approach are provided by the Sequoia [4] and PGCluster [16] middleware packages. The main advantage of this approach is its simplicity and failure transparency, since if a replica fails the requests are still processed by the others. It also trivially handles Data Definition Language (DDL) statements without any special requirements. On the other hand, the state machine operates correctly only under the assumption that requests are processed in a deterministic way, i.e., when provided with the same sequence of requests, replicas produce the same sequence of output and have the same final state. To start with, this requires that the original SQL command is rewritten to remove non-deterministic expressions and functions such as *now()*. A second source of non-determinism is scheduling of concurrently executing conflicting transactions, namely, the order by which locks are acquired is hard to predict. To overcome this problem, it is common to have an external global scheduler that manages which SQL commands can be concurrently processed without undermining the determinism requirement. This introduces additional complexity and may overly restrict concurrency in update-intensive workloads. The state-machine protocol requires that all replicas receive and process the

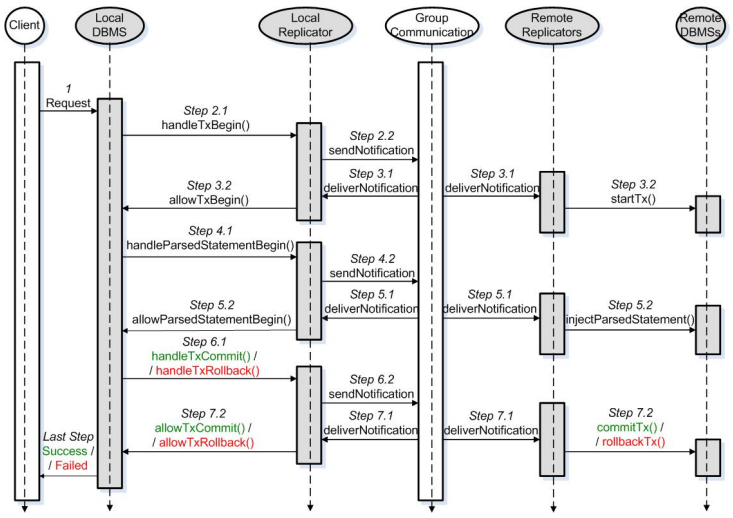

Figure 3: State-machine replication using.

same sequence of client requests producing a deterministic outcome. To accomplish this, we need to intercept client requests before it is processed enforcing deterministic executions. Specifically, begin, commit and rollback commands, implicitly or explicitly sent, and every SQL command should be intercepted. One possible solution is depicted in Figure 3.

Reflector Components Used State-machine replication requires the use of the Transaction context component and Parsing Stage component. On one hand, the transaction component is used to capture the moment where the transaction starts to execute, commits, or rollbacks at one replica. On the other hand, the Parsing Stage component is used to capture and start the execution of transaction statements.

Replicator Execution The execution of a state-machine replicator is depicted in Figure 3. It consists of the following steps. *Step 1:* Clients send their requests to one of the replicas. This replica is called the *delegate* replica; *Step 2:* Using the Transaction component the replicator at the delegate replica is notified of the beginning of the transaction. The replicator uses a totally ordered atomic broadcast to propagate this notification to all other replicas; *Step 3:* All replicators receive the notification in the same order. The transaction is started in remote replicas and resumed in the delegate

replica; *Step 4:* The transaction is executed at the delegate replica; Every time a new command starts the replicator is notified through the Parsing Stage component of the reflector interface. Then the replicator verifies if its parsed statement does not have any expression or function (e.g., *now()*) that might lead to non-deterministic executions. If so, it changes the parsed statement in order to remove the non-determinism. The resulting (potentially altered) parsed statement is broadcast to all replicators; *Step 5:* The parsed statement is received at all replicators. Replicators must implement a deterministic scheduler: each replicator must ensure that no two concurrent conflicting parsed statements are handled to the underlying DBMS. If such conflict exists, the parsed-statement is kept on hold. Otherwise it is handled to the DBMS at all replicas through the parsing stage component. It is worth noting two points related to this strategy. First, with this approach deadlocks may happen and the replicator should resolve them. Second, if a statement would be used, as it provides access to a command as a string, the replicator would also need to parse it to extract information on tables. *Further steps:* Steps 4 and 5 above are repeated; *Step 6:* Using the Transaction context component the replicator at the delegate replica is notified when the transaction is about to commit or rollback. This notification is atomically broadcast to all replicators; *Step 7:* Upon receiving a commit or rollback notification, remote replicas execute the proper command and the delegate replica allows it to proceed; *Final step:* Once the processing is completed, the delegate replica replies to the client.

4.3 Certification Based

Overview Certification based approaches operate by letting transactions execute optimistically in a single replica and, at commit time, run a coordinated certification procedure to enforce global consistency. Typically, global coordination is achieved with the help of an atomic broadcast service, that establishes a global total order among concurrent transactions [8, 14, 10, 22]. Multiple variants of the certification based approach have been proposed. Here we briefly describe an approach providing snapshot-isolation [10, 22]. At the time a transaction is initiated, a replica is chosen to execute the transaction (usually, the closest replica to the client which is called the delegate replica). When a transaction intends to commit, its identification, database version read, and the write set are broadcast to all replicas in total order. Right after being delivered by the atomic broadcast protocol, all replicas

verify if the received transaction has the same version as the database. If so, it should commit. Otherwise, one needs to check if previously committed transactions do not conflict with it. There is no conflict if previously committed transactions have not updated the same items. If a conflict is detected, the transaction is aborted. Otherwise, it is committed. Since this procedure is deterministic and all replicas, including the delegate replica, receive transactions by the same order, all replicas reach the same decision about the outcome of the transaction. The delegate replica can now inform the client application about the final outcome of the transaction.

This can be extended to serializability by considering also the read-set and then detecting read-write conflicts during certification [8, 14]. Although this might have some impact in performance [7], it is desirable for DBMS in which the consistency criterion is similar. Note also that certification based approaches do not require the entire database operation to be deterministic: Only the certification phase has to be processed in a deterministic manner. Furthermore, they allow different update transactions to be executed concurrently in different replicas. If the number of conflicts is relatively small, certification based approaches can provide both fault-tolerance and scalability. Certification based approaches operate by letting a transaction to execute optimistically in a single replica and, at commit time, execute a coordinated certification procedure to enforce global consistency.

Reflector Components Used Given its similarity to the Primary-Backup approach, the Certification based replication requires the use of the same components, explicitly the Transaction context and Parsing Stage components.

Replicator Execution The execution of a certification-based replicator is depicted in Figure 4. It consists of the following steps. *Step 1-4:* Same as in the Primary-Backup solution presented before; *Step 5:* Upon receiving the write-set, each replica certifies the transaction and decides its outcome: commit or abort. If it is an abort, the delegate replica through the transaction context component cancels the commit and remote replicas discard it. If it is a commit, the delegate replica allows it to continue and remote replicas inject updates in the DBMS; *Final Step:* The delegate replica returns the response to the client.

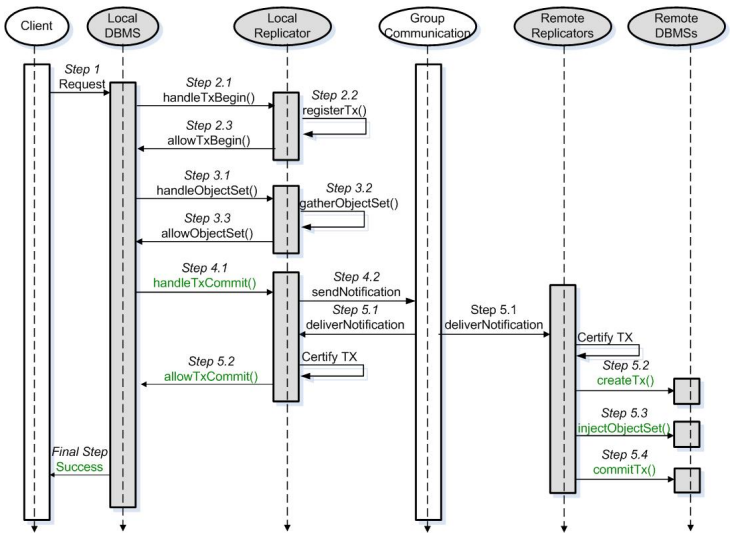

Figure 4: Certification-based replication.

5 Implementation

In this section we discuss 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 database management system 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 the upcoming Sun JDK 1.6 as JavaDB. It can either be embedded in applications or run as a standalone server. It uses locking to provide serializability. 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 contex objects and the parsing and execution objects, as well as calling between base-level and meta-level as described in Section 3. Therefore, it supports all use cases described in Section 4. 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 [17] is a fully featured database management system distributed under an open source license. Although written in C, it has been ported to multiple operating systems, and is included in most Linux distributions as well as in recent versions of Solaris. Commercial support and numerous third party add-ons are available from multiple vendors. Since version 7.0, it provides a multi-version concurrency control mechanism supporting snapshot isolation. The major issue in implementing 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 uses a single standalone Java virtual machine and inter-process communication. This imposes an inter-process remote procedure call overhead on all communication between base and meta-level. Furthermore, the prototype implementation of the GORDA interface in PostgreSQL 8.1 uses a hybrid approach. Instead of directly patching the reflector interface on the server, key functionality is added to existing client interfaces and as loadable modules. The proposed meta-level interface is then built on these. The two layer approach avoids introducing a large number of additional dependencies in the PostgreSQL code, most notably on the Java virtual machine. As an example, transaction events are obtained by implementing triggers on transaction begin and end. A loadable module is then provided to route such events to meta-objects in the external PL/J server. The current prototype exposes all contex objects and the parsing and execution objects, as well as calling between base-level and meta-level as described in Section 3. It avoids that meta-level operations are blocked by base-level operations simply by modifying the choice of the transactions to be terminated upon deadlock detection and write conflicts. Therefore, it supports all use cases described in Section 4. 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 [4] is a middleware package for database clustering built as a server wrapper. It is primarily targetted at obtaining replication or partitioning by configuring the controller with multiple backends, as well as improving availability by using several interconnected controllers. Nevertheless, when configured with a single controller and a single backend, Sequoia provides a state-of-the-art JDBC interceptor. 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 the 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 such statements at the parsing stage. It does also not avoid that base-level operations interfere with meta-level operations, and this cannot be implemented as described in the previous sections as one does not modify the backend DBMS. It is however possible to the clustering scheduler alread present in Sequoia to avoid concurrently scheduling base-level and meta-level operations to the backend, thus precluding conflicts. The effort required to implement was as follows: the size of the generic portion of Sequoia is 137238 lines, which includes the controller and the JDBC driver. Additionally, Sequoia contains 29373 lines implement plugable replication and partitioning strategies, that we don't use. In the controller, 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 three very different 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 server 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 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 state-of-the-art 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.

6 Performance

In this section we evaluate the performance of one prototype implementation of the proposed interface in PostgreSQL, which illustrates the performance of the GAPI implementation. The purpose of the evaluation is to assess the overhead introduced. It is important to evaluate also the overhead of the introduced changes when not in use, which if not negligible is a major obstacle to the adoption of the proposed architecture.

We use the workload generated by the industry standard TPC-W benchmark [11, 20]. TPC-W defines an Internet commerce environment that resembles real world, business oriented, transactional web applications. The relatively heavy weight transactions of TPC-W make CPU processing the bottleneck. In addition, we use only the Ordering Mix, which has 50% read-only transactions and 50% update transactions. The average size of an update transaction write-set in TPC-W is 275 bytes. We used 100 emulated browsers, that allows us to provide a realistic amount of concurrency, without overloading the server and thus the latency closely reflects processing overhead. If a very small concurrency level was used, concurrency bottlenecks would not be noticed. If a very large number of concurrent clients was used, latency would show mainly contention and not overhead.

The following scenarios were tested: (*i*) *Unmodified DBMS* is the original DBMS, without any modification, serving as the baseline; (*ii*) *DBMS + patch* is the modified DBSM, as described in the previous section, but without any meta-level objects and thus with all reflection disabled. Ideally, this does not introduce any performance overhead; (*iii*) *DBMS + without write-set* is the modified DBMS with listeners registered for transactional events and statements. This means that each transaction generates at least 3 events and (*iv*) *DBSM + all listeners* is the modified DBMS with listeners registered

	Mean latency (ms)	Std. Dev.	# Samples
PostgreSQL	1.766	1.882	50464
PostgreSQL + patch	1.922	1.430	50574
PostgreSQL + without write-set	2.718	1.501	50528
PostgreSQL + all listeners	3.016	1.884	50554

Table 1: Benchmark results.

for transactional events, statements, and all modified tuples. This causes a variable number of meta-level events allowing the capture of all modifications.

The results are presented in Table 1. Looking at the results, when no meta-level objects are configured it is not possible to conclude that the proposed modifications introduce a big overhead, given the standard deviation observed. However, the impact of registering meta-level objects is noticeable, as this causes several round-trips to the external PL/J server process. This is most notable when collecting the write-set. It is however acceptable, especially as the PostgreSQL architecture makes it the worst case scenario for implementing the proposed interface.

7 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 development 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. The contribution is then as follows:

First, by explaining how a number of different replication algorithms fit our proposed model, we have shown its usefulness. These include the state-machine approach, the primary-backup approach (both synchronous and

asynchronous), certification based approach, as well as heterogeneous replicas. Second, by implementing the proposed interface on three different and representative architectures, the Apache Derby and PostgreSQL databases as well as the Sequoia server wrapper, we have shown that the approach is viable and cost-effective. Finally, by benchmarking the PostgreSQL prototype using the industry standard TPC-W benchmark, we have shown that the proposed approach results in very low overhead when implemented within the database server, even when the server's architecture does not match the proposed abstraction as happens with PostgreSQL.

Prototypes described in this paper are published as open source, can be downloaded from the GORDA project's home page, 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 there.

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 *Proc. VLDB Ws. Design, Implementation, and Deployment of Database Replication*, 2005.

[3] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33 - 4:427 – 469, 2001.

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

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

[6] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, 1996.

[7] A. Correia Jr., A. Sousa, L. Soares, J. Pereira, R. Oliveira, and F. Moura. Group-based replication of on-line transaction processing servers. In *Proc. IEEE/IFIP Latin-American Dependability Conf. (LADC'05)*, 2005.

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

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

[10] Y. Lin, B. Kemme, M. Patio-Martnez, and R. Jimenez-Peris. Middleware based data replication providing snapshot isolation. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, 2005.

[11] Mikko H. Lipasti. Tpc-w in java. http://www.ece.wisc.edu/ pharm/tpcw.shtml.

[12] P. Martin, W. Powley, and D. Benoit. Using reflection to introduce self-tuning technology into dbmss. In *IDEAS '04: Proceedings of the International Database Engineering and Applications Symposium (IDEAS'04)*, pages 429–438, Washington, DC, USA, 2004. IEEE Computer Society.

[13] S. Mullender. *Distributed Systems*. ACM Press, 1989.

[14] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. In *Journal of Distributed and Parallel Databases and Technology*, 2003.

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

[16] PostgreSQL. PGCluster version 1.3. http://pgcluster.projects.postgresql.org/, 2006.

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

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

[19] A. Rashid and P. Sawyer. A database evolution taxonomy for object oriented databases. *Journal of Software Maintenance ? Practice and Experience*, 17:93–141, 2005.

[20] Daniel A. Menasc Ronald Dodge JR and Daniel Barbar. Testing e-commerce site scalability with tpc-w. In *Proceedings of 2001 Computer Measurement Group Conference*, Orlando, FL, December 2001.

[21] J. Salas, R. Jimenez-Peris, M. Patino-Martinez, and B. Kemme. Lightweight reflection for middleware-based database replication. In *SRDS'06: Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS'06)*, pages 377–390, Washington, DC, USA, 2006. IEEE Computer Society.

[22] S. Wu and B. Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE'05)*, 2005.