GORDA: An Open Architecture for Database Replication

A. Correia Jr. U. Minho — J. Pereira U. Minho — L. Rodrigues U. Lisboa — N. Carvalho U. Lisboa — R. Oliveira U. Minho — S. Guedes U. Lisboa

Abstract

Although database replication has been a standard feature in database management systems for a long time, third party solutions have been enjoying an increasing popularity. These solutions often rely on the use of group communication primitives to offer various forms of synchronous replication with reduced overhead. Unfortunately, the lack of native support 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 middleware architecture and programming interface for replication, such that different replication strategies can be efficiently implemented on top of any compliant database in a cost-effective manner. The contribution is two-fold. First we identify key functionality for representative replication protocols. Then we propose a middleware interface to expose its functionality and evaluate its implementation in both Apache Derby and PostgreSQL.

1. Introduction

Database replication has been a standard feature in database management systems (DBMS) for a long time, even in entry level products. Specifically, asynchronous or lazy propagation of updates provides a simple yet efficient way of increasing performance and data availability [9].

On the other hand, support for synchronous replication has been confined to high-end offerings. This has motivated the emergence of third-party solutions, such as the Sequoia replication and clustering middleware (formerly ObjectWeb C-JDBC) [6]. This type of solutions has been enjoying an increasing popularity, since it provides much of the advantages of synchronous replication and clustering without requiring the migration of legacy databases to different database management systems.

Furthermore, recent research in database replication based on group communication has proposed novel algorithms that are able achieve strong replica consistence without the overhead of traditional synchronous replication [11, 18, 14, 26, 19].

The lack of native support for third party replication requires those solutions to either modify the database server or to develop a server wrapper in middleware. However, the modification of the database server is very hard to maintain and port, and, in many cases, simply impossible due to unavailability of source code. A middleware wrapper, which implements replication and redirects requests to the actual underlying DBMS, represents a large development effort and introduces a non-negligible performance overhead.

This paper addresses the balance between portability and performance by proposing a novel middleware architecture and a programming interface for replication, such that different replication strategies can be implemented on top of any compliant database in a cost-effective manner. Instead of relying on client interfaces, we propose a reflective interface to transaction processing that supports a wide range of replication protocols.

The contribution of the paper is therefore two-fold. Firstly, we identify the key functionalities required by the most representative replication protocols in use today. Then we propose a middleware interface to expose its functionality. Finally we discuss the implementation of this interface in both Apache Derby 10.2 [2] and PostgreSQL 8.1 [21].

The work reported here is being developed in the context of an EU funded research project, GORDA (Open Replication of DAtabases),[1] 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, EMIC Networks Oy, and MySQL AB.

The rest of this paper is structured as follows. Section 2 discusses existing replication protocols and implementation strategies. Section 3 identifies a list of interface requirements and their impact on the implementation strategy. Section 4 introduces the GORDA architecture and interfaces. The application to several replication protocols is presented in Section 5. Section 6 discusses the implementation in two

[1] `http://gorda.di.uminho.pt/`

different systems. Section 7 concludes the paper.

2. Background

In this section, we survey some of the the most relevant replication strategies and then we address different architectures to implement them in concrete systems. This brief analysis is useful to motivate the replication interface proposed in the paper.

2.1. Target Approaches

Primary-Backup In the primary-backup approach to replication, also called passive replication [16], 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 DBMS and third-party offers. An example is the Slony-I package for PostgreSQL [22].

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 later 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-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.

State-machine The state-machine approach, also called active replication [16], 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 [6] and PGCluster [20] 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.

Certification Based 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 [11, 18, 14, 26].

Multiple variants of the certification based approach have been proposed. Here we briefly describe an approach providing snapshot-isolation [14, 26]. 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 [11, 18]. Although this might have some impact in performance [10], it is desirable for DBMS in which the native serializability criteria 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.

2.2. Implementations

Multiple architectures have been used to interface replication with DBMS. We discuss the main categories in the following paragraphs:

Replication implemented as a normal client. In this approach, the replication protocol connects to each DBMS using client interfaces, e.g. JDBC. This makes it portable and can be very efficient, namely, when code resides within the server using a server side client interface. This implementation strategy is however confined to the asynchronous primary-backup approach, as client interfaces don't provide the functionality required for synchronous replication. An example of an application of this approach is Slony-I [22], which provides asynchronous primary-backup replication of PostgreSQL. Typically, these solutions resource to installing triggers in the underlying DBMS in order to update meta-information and gather write-sets. The management of the DBMS lifecycle can be addressed by wrapping the database server initialization, without intercepting client requests.

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 [6]. The middleware layer presents itself to clients as a virtual database. Compared to the primary backup protocols, implemented as regular DBMS clients, this solution offers improved functionality, as 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 effort 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 write sets.

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 such as the Postgres-R prototypes [11, 26]. Given that it is implemented the the DBMS kernel, the replication protocol as an easy access to control information such as read and write sets, 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 multiple database servers but also as the implementation evolves.

Replication using proprietary interfaces. Database servers that natively support asynchronous primary-backup replication, usually do this using a well defined and published, although proprietary, interface. This allows some customization and integration with third party products, but is of little use when implementing recent innovations based on group communication. An exception is the Oracle Streams interface [25], which is based on existing standards, although still confined to asynchronous propagation of updates.

3. Requirements for a Replication Interface

We now identify a list of requirements that must be satisfied by an interface to support the implementation of replication protocols. These requirements have been extracted from our experience implementing not only the protocols above but also some other alternatives [23, 24, 17] that we have omitted due to lack of space.

(#1) Lifecycle Events: Mechanisms to observe and control the life cycle of a DBMS site, namely, when a site is started, recovers from logs, and put on-line to clients. This is required for proper recovery in coordination with remote sites, as a local log might have to be complemented by a remote log or even a state-transfer.

(#2) Object and Transaction Meta-Information: Mechanisms to record and retrieve protocol specific meta-information associated both with database objects (i.e. tables, tuples) and with transactions. For instance, global object identifiers or timestamps are needed by multiple protocols. For recovery purposes, updated to such meta-information in the context of user transactions that create or modify the corresponding objects.

(#3) Statement Inspection: Interception of statements submitted by clients either in a textual format or structured as parsed tree. This is required in state-machine replication to disseminate statements to replicas,as well as in most circumstances to handle DDL statements when update propagation relies on triggers to collect the write-set. Observation of statements is also useful to determine coarse-grained conflict classes to optimize state-machine replication.

(#4) Statement Modification: Modification or cancellation of statements, either in a textual format or structured as parsed tree. This is required to remove non-deterministic operations in state-machine replication. It is also required in clustering and partial replication to sub-set queries on each database partition or to cope with incompatible SQL dialects in heterogeneous environments.

(#5) Write-set Extraction: Capturing updates done to a database in a format that can be transfered and applied remotely. Certification-based approaches require that this is done within transactional boundaries, while for asynchronous primary-backup it is enough to periodically poll logs to recover modifications. A major issue is whether DDL statements are supported, which is not possible with a trigger based implementation.

(#6) Read-set Extraction: Capturing identifiers of objects read, in a format that can be transfered and used remotely for certification. This is required for certification-based approaches when one wants to achieve one-copy equivalence of the native concurrency control mechanisms in databases that use locking.

(#7) Efficient Write-set Injection: Although write-sets can always be applied using a regular client interface, it is hard to satisfy both correctness requirements to apply updates in a pre-defined order with performance. This often requires that updates are combined and scheduled beforehand to be applied in parallel [3].

(#8) Transactional Events: Observe transactional events such as transaction begin, rollback or commit. This is required in order to maintain version numbers as used by certification-based approaches. It can also ease the implementation of efficient parallel update application by allowing a predictable commit order to be established.

(#9) Commit Validation: Interception and validation of a request to commit an on-going transaction. The replication protocol needs to make sure that the transaction is able to commit successfully unless it explicitly rolls it back or a local fault occurs. Although this is similar to two-phase commit mechanisms associated with PREPARE statement, it is simpler and more efficient as it does not require durability. It is required for certification based approaches.

(#10) Predictable Deadlock Handling: Deterministic deadlock resolution mechanism, that can be controlled by middleware. This is required by certification based mechanism to ensure that already certified transactions are not aborted by locally executing transactions to solve deadlocks.

(#11) Result-Set Injection: Replace result-sets returned to clients. This is required by clustering and partial replication mechanisms to reconcile results from multiple database fragments. It is also required by certification-based approaches that ship statements to remote replicas for improved efficiency.

(#12) Runtime Model: An uniform runtime model for portable replication middleware components. Mainly, this is concerned with concurrency model which is different in each DBMS kernel. This is relevant as replication components can be installed in the database server itself and thus benefit from tight coupling with transaction execution kernel.

(#13) Configuration Storage: Replication protocols usually need also to keep meta-information that is updated only outside user transactions. This is not an issue unless the choice is the target database server itself and is to be used during recovery. This requires an extra step in the server life cycle that exposes such information while user databases are still off-line being recovered.

4. Reflective Interface

One key idea advocated in this paper is that the functionality required for supporting multiple replication protocols in an efficient manner should be provided through a common interface, regardless of the specific implementation approach. For instance, several replication protocols require the interception of the transaction commit: they should be able to perform such task using the same interface, regardless of being implemented as a client wrapper or by patching the database server.

To materialize this idea we propose the use of a common interface able to reflect abstract transaction processing concepts as objects in the target data model and programming language. In the GORDA project, this interface is simple called the *reflector*. This sort of approach [12, 13] has previously been shown to be a sound foundation for configurable and extendable software in a wide range of application areas, ranging from programming languages to distributed systems. The same idea is already being used in DBMS for different purposes, namely, for representing meta-information as relational tables and intercepting update operations with triggers. The idea that reflection can be used to enable self-tuning DBMS has also been advocated in [15].

In contrast with most previous uses of reflection in DBMS, our target data model is not the relational model. Instead, transaction processing concepts are reflected as objects in the Java programming language. This is justified

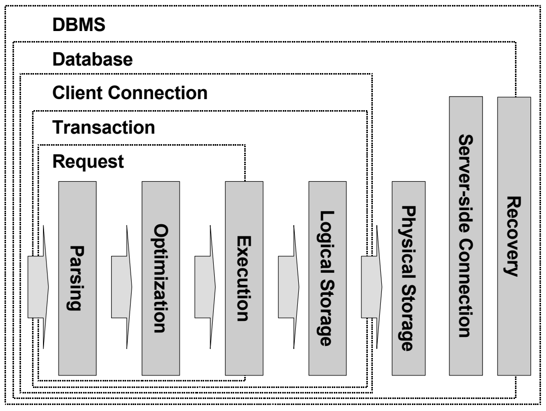

Figure 1. Reflector architecture.

as the proposed interface is not aimed at client application developers, but instead at those developing middleware for distributed systems. The same rationale underlies Oracle Streams [25] usage of the a messaging interface.

Integration with existing distributed systems middleware is achieved by reusing whenever possible interfaces and patterns from J2EE, specifically, event handling from JavaBeans and relational data manipulation from JDBC. Note however that, although being rendered in Java, the proposed interfaces are not Java specific in any way. One could easily translate them to languages in the same family such as C# or C++ and other middleware platforms that share the same design patterns [1].

A major feature of this interface is to separate *what* is being done, which is exposed as a global multi-stage pipeline, from on behalf of *whom* it is being done. This is captured by nested contexts which are associated with all events exposed. The rest of the section highlights this distinction and then discusses runtime and configuration issues.

4.1. Processing Stages

A major issue in implementing the requirements listed in Section 3 is that replication protocols interact with transaction processing at different levels of abstraction and therefore, need to interact with different sub-systems within the database server. Most of the reflector is therefore dedicated to exposing transaction processing as a pipeline corresponding to logical layers in a database management system.

As it will become clear later in this paper, different replication strategies make use of different portions of the reflector interface. To have an operational system one needs to ensure that the database exports the parts required by the replication protocol that one wants to deploy. Therefore, it is not required that every stage of the reflector interface is available at every implementation: depending on the replication protocol and on the properties of the underlying DBMS, the implementation of some stages may become optional.

Figure 1 shows an overview of the considered processing stages. In general, the replication protocol can be notified when data advances from one stage to the next. At that time, it can inspect and modify such data structure, and even delay or suppress it.

In detail, the *Parsing* stage parses raw statements received thus producing a parse tree. It allows inspecting and modifying statements in a textual format. It allows also delaying or even completely suppressing statements. The parse tree is transformed by the *Optimization* stage according to various optimization criteria, heuristics and statistics to an execution plan. This provides to replication protocols mostly the same functionality as the previous stage, although on a different data structure, which eases some operations. The *Execution* stage executes the plan and produces read set, write set, lock-grabbed set and lock-blocked set. The *Logical Storage* stage deals with mapping from logical objects to physical storage. In detail, it allows intercepting and injecting write-sets. Finally, the *Physical Storage* stage deals mostly with synchronization of commit requests.

Notice that the proposed model is based on classic transaction processing phases [8] augmented to produce not only write and read sets but also lock-grabbed and lock-blocked sets, thus exposing which locks were acquired and which locks are blocking the execution respectively. This information might be used by a consistency criterion that requires knowledge about locks, such as range locks, to ensure one-copy serializability [4].

Such stages are therefore not mutually exclusive. It is likely that different parts of the same transaction or even of the same statement are in different stages of the pipeline. As an example, some dirty blocks might be being flushed to disk at the physical level while at the same time, at the higher level, a new statement is being parsed.

All interfaces to data structures are designed such that can be implemented as thin facades [7] to the internal state of the DBMS. Thus, such interfaces allow manipulation of the internal state of the DBMS without forward and backward format conversions. Conversion to a DBMS independent representation, if necessary, should be achieved by an additional layer. For instance, to enable the propagation of the updates among different database vendors and architectures, a rendering of the write-set interface should store information using a portable representation.

Additional interfaces are provided for *Recovery*, which allows the replication protocol to alter local recovery by suppressing some transactions from the log or adding other received from remote replicas. Access to a *Server-side Connection* to the database is also provided, as client functionality is often desirable.

4.2. Processing Contexts

The processing pipeline provides a global view of the database management system. However, it is often useful that events from a single or from multiple levels are grouped together and handled within a common context. In detail, 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 feasible to determine on behalf of which transaction a specific disk block is being flushed by the physical stage.

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 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.

Furthermore, replication can attach an arbitrary object to each context. This allows context information to be extended as required by the each specific 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.

4.3. Runtime and Configuration

The implementation of a replication protocol using the reflector interface requires registering listeners to multiple events fired by reflector components. It is then important to discuss how these listeners can be configured and what are the concurrency semantics they imposed on protocol implementations.

Event listeners can be registered either as synchronous or asynchronous handlers. When registered as asynchronous, notification is queued but processing continues, thus imposing only a minimal overhead on normal transaction processing at the expense of not allowing data structures to be modified. When registered as a synchronous handler, processing halts and waits for confirmation or cancellation from the replication protocol. If this is done immediately, before the event handler finishes, overhead should still be very low. Synchronous event handlers can also defer confirmation and even alter state before restarting or canceling processing.

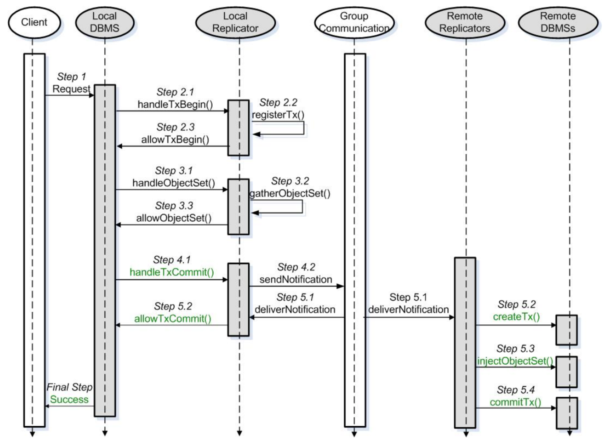

Figure 2. Primary-backup replication.

Event notification is performed on multiple threads. The reflector implementation thus manages active threads within the replication protocol according to the policy it seems adequate, usually, in close cooperation with the transaction processing kernel. However, synchronous event listeners with ordering constraints are not executed concurrently. Namely, transaction events are synchronized by the reflector such that they can be handled in a predictable order.

Finally, configuration interfaces are assumed to be out of the scope of the reflector specification. It is therefore assumed that reflector and replication components are managed by a container that provides adequate dependency resolution, namely, by resorting to a naming service. This simplifies reflector interfaces and allows a tight coupling with DBMS administration interfaces.

5. Case Studies

This section describes use cases of the reflector interface for renderings of state-machine, primary-backup and certification-based replication protocols.

5.1. Primary-Backup

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 to execute, 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* [5]).

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.

5.2. State-machine

The state-machine protocol requires that all replicas receive and process the same sequence of client requests producing a deterministic outcome. To accomplish this, we

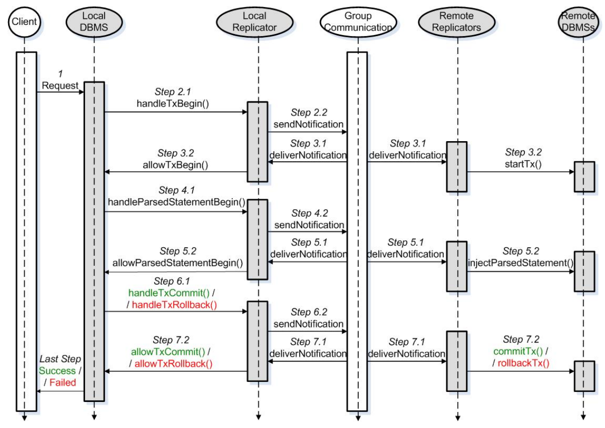
Client
Local DBMS
Local Replicator
Group Communication
Remote Replicators
Remote DBMSs
1 Request
Step 2.1 handleTxBegin()
Step 2.2 sendNotification
Step 3.1 deliverNotification
Step 3.1 deliverNotification
Step 3.2 allowTxBegin()
Step 3.2 startTx()
Step 4.1 handleParsedStatementBegin()
Step 4.2 sendNotification
Step 5.1 deliverNotification
Step 5.1 deliverNotification
Step 5.2 allowParsedStatementBegin()
Step 5.2 injectParsedStatement()
Step 6.1 handleTxCommit() / / handleTxRollback()
Step 6.2 sendNotification
Step 7.1 deliverNotification
Step 7.1 deliverNotification
Step 7.2 allowTxCommit() / / allowTxRollback()
Step 7.2 commitTx() / / rollbackTx()
Last Step Success / / Failed

Figure 3. State-machine replication using.

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

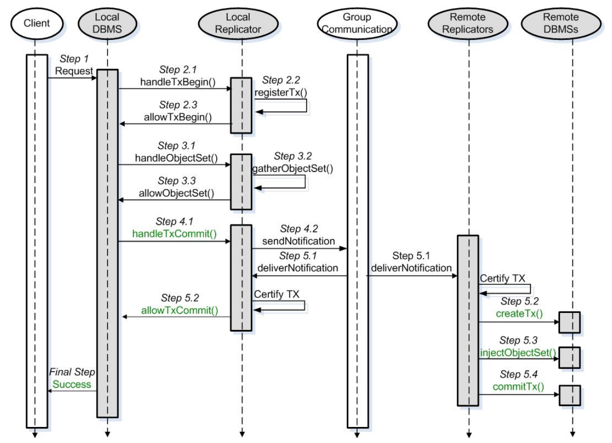

Figure 4. Certification-based replication.

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 such string 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.

5.3. Certification Based

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.

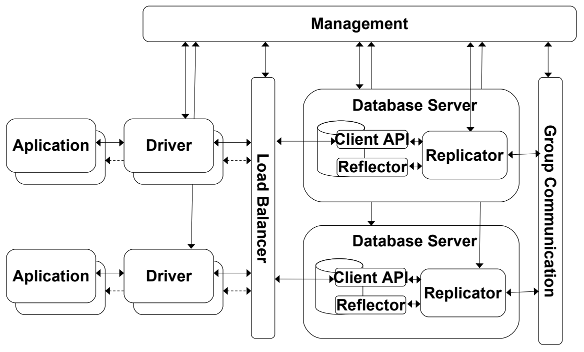

Figure 5. Generic GORDA architecture.

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.

6. Implementation

The general architecture of a replicated DBMS using the GORDA interface is presented in Figure 5. At each site, replication protocol components are tightly coupled to a local replica using the reflector interface, that subsumes a normal client interface, thus observing and controlling it. The communication between replication components depends on the specific protocol, which is likely to be a group communication toolkit [5].

Notice that different implementations may map architectural components to different process boundaries. In this section we discuss design decisions in two prototype implementations of the GORDA interfaces in Apache Derby and

PostgreSQL, namely, how the reflector is bound to the transaction processing kernel, and which sub-set of the interface was implemented.

6.1. Apache Derby 10.2

Apache Derby 10.2 [2] 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 in Apache Derby 10.2 is done by patching the server and aims at fully implementing all proposed processing stages and contexts. The reflector and replication protocol components are therefore embedded within the Apache Derby kernel. This approach is eased by the fact that Apache Derby is completely implemented in Java, using a compatible threading model.

The current prototype implements almost entirely the reflector interface with the exception of the *Execution* and *Physical Storage* stages, thus not allowing inspection and modification of execution plans or logger object sets. It also does not yet support extracting fine-grained read-sets, providing them only with table level granularity, and the Client Connection context. This is however sufficient to run the protocols described in Section 5.

6.2. PostgreSQL 8.1

PostgreSQL 8.1 [21] 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 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 reflector is then build 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, requirement #8 is satisfied by implementing triggers on transaction begin and end. A loadable module is then provided to route such events to the reflector interface.

The major issue in implementing the PostgreSQL 8.1 reflector is the mismatch between its concurrency model and the multi-threaded reflector runtime. PostgreSQL 8.1, as all previous versions, uses multiple single-threaded operating system processes for concurrency. This forces replication protocols to run also in a separate process, using an inter-process communication mechanism, as these depend on shared Java objects.

A similar dilemma was faced by developers of server side binding for Java. The PL/Java proposal uses a local Java virtual machine in each PostgreSQL process, thus precluding shared Java objects. The alternative PL/J proposal, uses a single standalone Java virtual machine and interprocess communication. A modification of PostgreSQL to run a single multithreaded backend has been discussed and is currently on the developers to-do list.

The PostgreSQL prototype therefore excludes the *Optimization*, *Execution* and *Physical Storage* stages in order to minimize overhead. This is however sufficient to run the protocols described in Section 5. Although not required for one-copy equivalence, as the native consistency criterion is snapshot isolation, the PostgreSQL prototype includes an experimental implementation of read-set extraction.

6.3. Others

The GORDA interface is also being implemented in MySQL and as DBMS independent server wrapper. Both pose interesting challenges. The MySQL implementation has to deal with multiple storage engines, thus complicating the implementation of the later stages of the pipeline.

The server wrapper builds on Sequoia (formerly C-JDBC) and attaches the reflector to the *virtual DBMS* at the middleware level. This implementation strategy restricts the scope of reflector interfaces that can be efficiently provided. Still, this it preserves the logical decoupling between the replication algorithms and the DBMS interface and re-use the same algorithms with both compliant and non-compliant DBMS's. The use of the reflector interface with non-compliant DBMS is interesting to foster adoption by widening the applicability of compliant replication protocols.

7. Conclusions

Recent developments in database replication and clustering have been placing new demands on DBMS interfaces. This is particularly true for replication approaches that rely on group communication to offer synchronous replication with a minimum overhead. 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 have addressed this problem by proposing an interface to attach replication protocols and database management systems. The interface has been designed to support a variety of DBMS systems, replication protocols and implementation strategies. It builds on the concept of reflection of transaction processing to the Java language at multiple levels of abstraction. When compared to other approaches to build configurable software, such as a component framework, reflective interfaces can more easily be mapped to existing implementations. This approach is also a good fit with reflective mechanisms already common in DBSM.

The approach is illustrated with the discussion of how multiple representative replication protocols have been implemented using the interface. It is also illustrated with the implementation discussion of the proposed interfaces in actual database management systems, namely, Apache Derby and PostgreSQL.

References

[1] D. Alur, J. Crupi, and D. Malk. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall and Sun Microsystems Press, 2001.

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

[3] N. Arora. Oracle Streams for near real time asynchronous replication. In *Proc. VLDB Ws. Design, Implementation, and Deployment of Database Replication*, 2005.

[4] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

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

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

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Desing Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

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

[9] 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.

[10] A. C. 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.

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

[12] G. Kiczales. Towards a new model of abstraction in the engineering of software. In *Proc. Intl. Ws. New Models for Software Architecture*, 1992.

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

[14] 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.

[15] P. Martin, W. Powley, and D. Benoit. Using reflection to introduce self-tuning technology into dbmss. In *Proc. Intl. Database Engineering and Applications Symp. (IDEAS'04)*, 2004.

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

[17] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Proc. Intl. Conf. on Distributed Computing (DISC)*, 2000.

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

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

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

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

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

[23] L. Rodrigues, H. Miranda., R. Almeida., J. Martins., and P. Vicente. The GlobData fault-tolerant replicated distributed object database. In *EURASIA-ICT*, 2002.

[24] A. Sousa, F. Pedone, R. Oliveira, and F. Moura. Partial replication in the database state machine. In *IEEE NCA*, 2001.

[25] M. Tumma. *Oracle Streams High Speed Replication and Data Sharing*. Rampant, 2004.

[26] 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.