Lightweight Reflection for Middleware-based Database Replication

J. Salas*, R. Jiménez-Peris*, M. Patiño-Martínez*
Universidad Politecnica de Madrid (UPM)
Madrid, Spain
{jsalas, rjimenez, mpatino}@fi.upm.es

B. Kemme
McGill University
Montreal, Quebec, Canada
kemme@cs.mcgill.ca

Abstract

Middleware-based database replication approaches have emerged in the last few years as an alternative to traditional database replication implemented within the database kernel. A middleware approach enables third party vendors to provide high availability solutions, a growing practice nowadays in the software industry. However, middleware solutions often lack scalability and exhibit a number of consistency and performance issues. The reason is that in most cases the middleware has to handle the database as a black box, and hence, cannot take advantage of the many optimizations implemented in the database kernel. Thus, middleware solutions often reimplement key functionality but cannot achieve the same efficiency as a kernel implementation. Reflection has been proposed during the last decade as a fruitful paradigm to separate non-functional aspects from functional ones, simplifying software development and maintenance whilst fostering reuse. However, fully reflective databases are not feasible due to the high cost of reflection. Our claim is that by exposing some minimal database functionality through a lightweight reflective interface, efficient and scalable middleware database replication can be attained. In this paper we explore a wide variety of such lightweight reflective interfaces and discuss what kind of replication algorithms they enable. We also discuss implementation alternatives for some of these interfaces and evaluate their performance.

Keywords: database replication, reflection, middleware, fault-tolerant distributed systems.

*This work has been partially funded by the European Adapt project, the Spanish Ministry of Education and Science (MEC) under grant #TIN2004-07474-C02-01, and the Madrid Regional Research Council (CAM) under grant #0505/TIC/000285.

1 Introduction

Database replication is a topic that has attracted a lot of research during the last years. There are two main reasons: on the one hand databases are frequently the bottleneck of more complex systems such as multi-tier architectures that need higher throughput; on the other hand, databases store critical information that should remain highly available. Database replication has been employed to address these issues. A critical issue of database replication is how to keep the copies consistent when updates occur. That is, whenever a transaction updates data records, these updates have to be performed at all replicas. Traditional approaches have studied how to implement replication within the database, what we term the *white box* approach [11, 10, 22, 23]. However, the white box approach has a number of shortcomings. Firstly, it requires access to source code. This means that only the database vendor will be able to implement them. Secondly, it is typically tightly integrated with the implementation of the regular database functionality, in order to take advantage of the many optimizations performed within the database kernel. However, this approach results in the creation of inter-dependencies between the replication code and the other database modules, and is hard to maintain in a continuously evolving codebase.

Recent research has focused on how to perform replication outside the database [2, 36, 12, 5, 4, 15, 31, 33, 35, 26], typically as a middleware layer. However, nearly none of them is truly a *black-box* approach in which the database is used exclusively based on its user interface since this would lead to very simplistic and inefficient replication mechanisms. Instead, in the simplest form, they require some information from the application. In the most basic form, they parse incoming SQL statements in order to determine the tables accessed by an operation [12]. This allows to perform simple concurrency control at the middleware layer. More stringent, they might require the first command within a transaction to indicate whether the transaction is read-only or an update transaction [35], or to indicate the tables that are going to be accessed by the transaction [5, 4, 32]. Nevertheless, many do not require any additional functionality from the database system itself. However, this can lead to inefficiencies. For instance, it requires update operations or even entire update transactions to be executed at all replicas which results in limited scalability. We term this approach *symmetric processing*. An alternative is *asymmetric processing* of updates that consists in executing an update transaction at any of the replicas and then propagate and apply only the updated tuples at the remaining replicas. [20] shows that the asymmetric approach results in a dramatic increase of scalability even for workloads with large percentages (80% and above) of update transactions. However, retrieving the writesets and applying them at remote sites is non-trivial and is most efficient if the database system provides some support. We believe that even further replication efficiency could be achieved, if the database exposed even more functionality to the middleware layer.

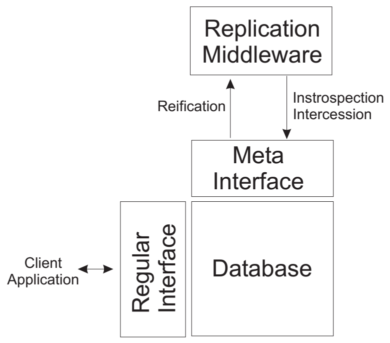


Figure 1. A reflective database

Thus, in this paper we try to combine the advantages of white and black box approaches by resorting to computational reflection [28]. The essence of computational reflection lies in the capability of a system to reason about itself and its behaviour, and act upon it. A reflective system is structured around a representation of itself or *meta-model*. This meta-model might provide different abstractions of the underlying system with different levels of detail. A reflective system is split into: a base-level (the database in Figure 1), where the regular computation takes place, and a meta-level, where the system reasons about this regular computation and extends it (the replication middleware in Figure 1). *Reification* is the process by which changes in the base-model are reflected in the meta-model. *Introspection* is the mechanism that enables the meta-model to interrogate the base-model about its structure and/or behavior. *Intercession* is the mechanism through which the meta-model can change the state and/or behavior of the base-model.

We use reflection to expose some functionality of the database resulting in what we term a *gray box approach* to database replication (Fig. 1). Unlike previous reflective approaches for middleware [25, 9], our goal is not to propose a full-fledged metamodel of a database, since the inherent overheads are prohibitive and incompatible with the high performance requirements of databases. Our aim is to identify a set of composable, minimal and lightweight reflective interfaces that enable to attain high performance replication at the middleware level. We start by a set of basic interfaces which are essential for the practicality of the approach. Functionality provided by such interfaces is already implicitly used by existing protocols. From there, we reason about more advanced functionality which we have identified of being beneficial for database replication. These advanced interfaces allow us to obtain the combined benefits of black and white box approaches. That is, on the one hand we achieve separation of concerns by having separate components for the regular database functionality and the replication code; on the other hand, we can perform sophisticated optimizations in our replication middleware.

In the following, Section 2 gives an overview of techniques used in database replication. Section 3 presents two replication algorithms that will serve as examples to discuss our needs for reflection. Section 4 provides a series of interesting reflective interfaces. Section 5 presents the implementation and evaluation of some specific reflective interface. Section 6 presents related work and Section 7 concludes the paper.

2 Taxonomy of Database Replication Protocols

We classify replication protocols across several dimensions, which extends the criteria defined in previous taxonomies [16, 39]:

- *when to propagate changes*: In *eager* protocols updates or the changes of a transaction (also called *writesets*) are propagated as part of the original transaction. In contrast, with *lazy replication*, updates are propagated as a separate transaction.
- *where to execute update transactions: Primary copy replication* requires that update transactions are executed at a given site (the primary). The primary propagates the changes to the secondary replicas which only accept read-only transactions from their local clients. In order to schedule read-only transactions to secondaries and updates to the primary, some systems require the application program to tag transactions as read-only or not [35]. If update transactions can be executed at any replica, the replication protocol follows an *update everywhere* approach.
- *number of messages:* This parameter considers the *number of messages* per transaction. Some protocols use a constant number of messages, while others require a linear number of messages depending on the number of update operations within the transaction.
- *coordination protocol:* Some replica control mechanisms require a coordination protocol among the replicas in order to terminate the transaction (*voting termination*). In others, each replica decides itself about the outcome of a transaction given the information it has received so far (*non-voting*).
- *correctness criteria:* The correctness criteria typically implemented is *1-copy-serializability* (1CS) [8]. *Serializability* guarantees that the concurrent execution of transactions is equivalent to a serial execution. 1CS guarantees that a replicated execution is equivalent to a serial execution over a non-replicated database. Centralized database systems usually offer apart of serializability more relaxed forms of isolation, such as the ANSI isolation levels or snapshot isolation [7]. Snapshot isolation is based on multiversion optimistic concurrency control as implemented by Oracle, Microsoft SQLServer (the just released Yukon), and PostgreSQL. Accordingly, 1-copy-snapshot-isolation [26] or generalized snapshot isolation [14] define what it means to provide snapshot isolation in a replicated system.
- *concurrency control:* Replica control can be combined with both optimistic and pessimistic concurrency control. A *pessimistic* approach restricts concurrency to enforce consistency at all replicas. For instance, a protocol could execute all update transactions sequentially to ensure the same state at all replicas. A protocol can increase concurrency in the replicated database by having some a priori knowledge about the data objects that are going to be modified. With this, transactions that access different objects can be executed in parallel and those that access the same objects are executed sequentially. An object can be on the granularity of a table, a tuple or a conflict class. Conflict classes are partitions of the data, and have to be defined by the application in advance. This facility is available in most commercial databases (Oracle, DB2, Sybase). One may think that having knowledge about conflict classes of a transaction is unrealistic. However, in many cases a database is accessed via well-defined application software packages which can be parsed to detect the update patterns. Note, however, that conflict classes and tables are typically of quite coarse granularity. Hence, some transactions might be executed serially (because accessing the same table/conflict class) although they do not conflict on the tuple level. *Optimistic* approaches submit potentially conflicting transactions in parallel. Then, after transaction execution a validation phase is run which checks if there was a conflict among the transaction being validated and those that run concurrently and already validated. If this is the case, the validating transaction has to abort.
- *update processing:* As mentioned above, there are two ways of processing update transactions: symmetric or asymmetric processing. With symmetric processing each update transaction is submitted and fully executed at all replicas. On the other hand, in an asymmetric protocol update transactions are executed at a single replica and only their changes (*writeset*) are propagated to the rest of the replicas. Some approaches lie in between [12, 5] being symmetric at the statement level, but asymmetric at the transaction level. That is, if an update transactions contains both update and query statements, the query statements are executed at one replica while update statements are executed at all replicas.
- *transaction restrictions:* Another interesting dimension is related to the constraints set on

the kind of transactions that can be replicated. Some protocols only allow single statement transactions (known as auto-commit mode in JDBC) [2]. Other protocols allow several statements within a transaction but they have to be known at the beginning of the transaction. This can be implemented using stored procedures (or prepared statements in JDBC) [24, 33, 3]. The more general protocols do not have any restriction on the number of statements a transaction contains [22, 40, 26, 14, 35, 12].

3 Two Basic Database Replication Protocols

The goal of this section is to give an intuition of how a typical database replication protocol looks like. We first look at very basic protocols. From there, we discuss the use of reflective interfaces, and the requirement for further reflective capabilities in order to be able to enhance the basic protocols in different aspects. One of the protocols is pessimistic and the other optimistic. Both protocols are eager, update-everywhere protocols and provide one-copy serializability. They use group communication systems [13] that provide a total order multicast (all messages are delivered to all replicas in the same order). For simplicity of description, we ignore reliability of message delivery in this paper.

The pessimistic replication protocol performs symmetric processing of updates and assumes single statement transactions or multiple statement transactions sent in a single request. In this protocol, a client sends its requests to one of the replicas. Read-only transactions are processed locally. However, update transactions are multicast in total order to all the replicas and processed at each of them sequentially. This protocol is executed at all the replicas and can be summarized as follows (which resembles the one proposed by [2]):

```
I.   Upon receiving a request for the execution of an
     update transaction from the client: multicast the
     request to all replicas with total order.
II.  Upon delivering an update transaction request:
     enqueue the request in a FIFO queue.
III. Once a request is the first in the queue: submit
     transaction for execution.
IV.  Once a transaction finishes execution: remove it
     from queue. If local return response to client.
```

This basic protocol only needs minimal reflective support. We discuss later how this basic protocol can be enhanced to improve performance and functionality, and how these improvements require extensions to the reflective interface.

The optimistic protocol is a simplified version of [34] and performs asymmetric processing of updates. For simplicity of description we assume that each request is one transaction (multiple request transactions could be handled in the same way). A client submits a request to one of the replicas where it is executed locally. If the transaction is read-only the reply is returned to the client immediately. Otherwise, a distributed validation is needed that checks whether 1-copy serializability has been preserved. If not, the validating transaction is aborted.

```
I.   Upon receiving a transaction request from the
     client: execute it locally.
II.  Upon completing the execution:
       1. If read-only: return response to client.
       2. If update transaction: extract the read ($RS$)
          and writeset ($WS$) and multicast them in total
          order.
III. Upon delivering $RS$ and $WS$ of transaction $tv$:
     validate it with transactions $tc$ that committed af-
     ter $tv$ started.
       1. If $WS(tc) \cap RS(tv) \neq \emptyset$: if local, abort $tv$ and
          return abort to client (it should have read $tc$'s
          update but might have read an earlier version),
          otherwise ignore $tv$
       2. If $WS(tc) \cap RS(tv) = \emptyset$: if $tv$ not local, apply
          the writeset of $tv$ and commit. If local, commit
          $tv$ and return commit to user.
```

4 Reflective Interfaces for Database Replication

In this section we study which reflective interfaces can be exhibited by databases to enable the implementation of the protocols presented in the previous section at the middleware level. Additionally, we also discuss how these protocols can be enhanced and which extensions to the reflective interface these enhancements require. The interfaces we present range from the well-known request interception to novel concepts such as reflective concurrency control.

4.1 Reflective Database Connection

Clients open a database connection by means of a DB connectivity component (e.g. JDBC or

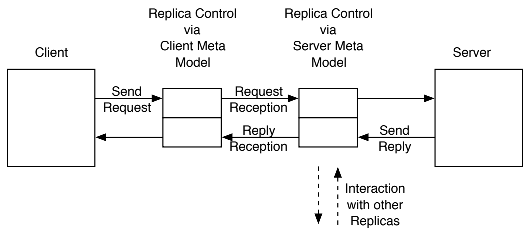

Client
Replica Control via Client Meta Model
Replica Control via Server Meta Model
Server
Send Request
Request Reception
Reply Reception
Send Reply
Interaction with other Replicas

Figure 2. Reflective Database Connection

ODBC), which runs at the client side, and submit transactions through it. The DB connectivity component forwards the connection request and the transactions to the DB server that processes them. The DB server then returns the corresponding replies that the connectivity component relays to the client. Finally, when the client is done, it closes the connection. Since database functionality is split into a client and server part, we have a base model, meta-level and meta-model at both the client and server side. The client base-level is the client connectivity component. The database base-level is the connection handler. The well-known request interception reflective technique can be applied at the connectivity component and the connection handler to implement replication as a middleware layer.

4.1.1 Basic Algorithm

Figure 2 shows how reflective support is required from the DB connectivity component and the connection handler to enable the pessimistic basic algorithm presented in the previous section. First, since the client is not aware of the replication, the connection request should be intercepted by the meta-model. This provides the first hook to insert the replication logic (*SendRequest*). The connection request will be reified and at the meta-model the connection will be performed by first executing a replica discovery protocol (e.g. by means of IP-multicast as in Middle-R [26]) to discover the available replicas. Then, one is chosen and the connection is established with it. This result will be reified (*ReplyReception*) to the client meta-level so that it can keep track of the replica to which it is connected. The standard client request (transaction requests) and the responses from the server can be intercepted in the same way, allowing further actions of the replication protocol. For the basic protocol no special actions are needed, and request and response are simply forwarded. Let us now examine what is required at the server side. A request that is received by one server replica should be reified (*RequestReception*) to the server meta-level. If it is a connection request the replica has to register the client. If it is a transaction request, it needs to be multicast to all replicas in total order where it will trigger transaction execution at the base-level of all servers as a form of intercession. When the request processing is completed at the server base-level the result is reified (*SendReply*) to the corresponding meta-level. This is a further hook at which the replication algorithm can apply the replication logic. For instance, for the response to a transaction request, only the local replica has to return the result to the client.

When looking at the optimistic algorithm, a simple reflective mechanism at the database connection level is not enough. Hence, we defer the discussion of this protocol to the next sections.

4.1.2 Fault-tolerance

However, a reflective database connection can enhance our pessimistic protocol (and in a similar way the optimistic protocol), by providing the right hooks to integrate fault-tolerance. A client should stay connected to the replicated system even if a replica fails. Ideally, the client itself is not even aware of any failures but experiences an uninterrupted service. Reflection at the connection level can achieve this. As mentioned before, when a client wants to connect to the system, the client meta-level can detect existing replicas and connect to any of them. In a similar way, when the replica to which the client is connected to fails, the failure needs to be reified to the client meta-level. Then, the meta-level can automatically reconnect to a different replica without the client noticing. For that the client meta-model has to do extra actions when intercepting standard transaction requests and their responses. A possible execution can be outlined as follows. When it receives a request it tags it with a unique identifier and caches it locally before forwarding the tagged request to the server. If the meta-model receives a failure exception or times out when waiting for a response it can reconnect to another replica and resubmit the request with the same identifier. The server meta-model of each replica keeps track of the last request and its response for each client using the request identifier. Hence, when it intercepts a request, it first checks whether it is a resubmission. If yes, it returns immediately the result. Otherwise, it is multicast and executed at all replicas as described above. At least-once execution is provided by letting the client meta-level resubmit outstanding request. At-most once is guaranteed by detecting duplicate submissions. Note that the server meta-level has to remove the request identifier from the request before the request is forwarded to the base-level server in order to keep the regular interface unmodified.

4.2 Reflective Requests

In the previous section, it was argued that without information about the content of the requests the replication logic was forced to use a read all write all symmetric approach executing all requests sequentially at all sites. That is, no concurrency control is performed at the middleware level. In order to enable more efficient replication protocols at the middleware level it is necessary to have an additional meta-interface. This meta-interface will enable to perform introspection on the request and might also provide access to application-dependent knowledge on the transaction access pattern. Therefore, this meta-interface can offer information about transaction requests with different levels of detail: 1) It can classify the transaction as read-only or update; 2) It can provide information about the tables that are going to be accessed by the transaction and in which mode, read or update; 3) It can determine the conflict classes (application defined) to be accessed by the transaction.

Level 1 is offered, for instance, by JDBC drivers through `SetConnectionToReadOnly`. This information is exploited by replication middlewares such as [2] and [35]. It is particularly helpful in the case of primary copy replication. In this case, the replication protocol can redirect update transactions to the base level of the primary server and read-only transactions to any other replica. Levels 2 and 3 are used to implement concurrency control at the middleware level. Level 2 can be easily achieved through a SQL parser run at the client (or server) metamodel as has been done in [21]. Level 3 requires a meta-interface so that application programmers can define conflict classes and transactions can be attached the set of conflict classes they access. Level 3 can be exploited by conflict aware schedulers such as in [32, 19, 5, 4, 12]. Middle-R [33] has an implementation of such interface. If levels 2 or 3 are available, then our basic pessimistic protocol can be extended. Instead of having one FIFO queue, there can be queues for each table or conflict class and requests are appended to the queues of tables/classes they access. Hence, transactions that do not conflict can be submitted concurrently to the base-level.

4.3 Reflective Transactions

For the optimistic and asymmetric protocol presented in the previous section, however, reflective connections and requests are not yet enough. Asymmetric replication requires to retrieve and apply writesets. While the optimistic concurrency control could be achieved through reflective requests, the coarse conflict granularity achieved at this level (on a table or conflict-class basis) is likely to lead to many aborts. Thus, in order for optimistic concurrency control to be attractive, conflicts should be detected on the tuple level. For this purpose we need reflective transactions.

4.3.1 Writesets

In order to be able to perform asymmetric replication, the meta-model needs to be able to obtain a writeset of a transaction from the base-level and apply a writeset. The first request requires reification, the second intercession.

The writeset contains the updated/inserted/removed tuples identified through the primary key. The meta-interface can adopt three different forms. The first form provides the writeset as a black box. In this case, the writeset can only be used to propagate changes and apply them at a different replica through the meta-interface. The second form consists in a reflective writeset providing an introspection interface itself. This introspection interface enables to analyze the content of the writeset, for instance, in order to identify the primary keys of the updated tuples. This is needed for our optimistic protocol to detect conflicts. The third form offers a textual representation of the writeset. Typically the textual representation will be an SQL update statement identifying the updated tuples with a primary key. Though, other textual representations are possible, like for instance, XML.With this, replication could be across heterogeneous databases since one could retrieve a writeset from a PostgreSQL database and apply it to an Oracle instance.

The writeset meta-interface can be implemented efficiently. At the time updates are physically processed at the database, the updates can be recorded in a memory buffer. The meta-interface just provides access to this buffer. A binary meta-interface for PostgreSQL is used in [19, 21], an introspective writeset meta-interface exists for PostgreSQL 7.2 [26], and we have just completed a textual one (as a SQL statement) for MySQL.

If the database itself does not provide writeset functionality, it can be implemented through different means such as triggers [35]. This approach has the advantage of not requiring the modification of the database code. However, it has the shortcoming of the high cost incurred by triggers. Typically, each update of a data item triggers the insert of writeset information into a special table. Retrieving the writeset means reading what a specific transaction has inserted into this special table. Thus, each update of a tuple in the original transaction leads to two updates when writeset functionality should be provided.

4.3.2 Readsets

In order to perform the validation of our optimistic replication algorithm, we also need the readset information. The readset meta-interface is similar to the writeset one. The main difference is that for the readset only the primary key of the read tuples is needed. The implementation is also relatively simple. Whenever a physical read takes place, the primary key is recorded on a buffer. However, the practicality of the approach is very questionable. Writesets are usually small while readset can become very large, e.g., if complex join operations are used. Thus propagating readsets might be prohibitive. Also, performing validation on readsets can be very expensive. Therefore, although the readset can be supported at the meta-interface, in general, very few protocols use it.

4.4 Reflective Log

The log of a database registers the undo and redo records to guarantee transaction atomicity. Redo records are in fact a form of a writeset. A reflective log provides an introspection interface that enables to analyze the records written to it by each transaction. Although conceptually similar to the writeset approach presented above, there is a very important difference. Writesets with the previous meta-interface are obtained before transaction completion, whilst in a reflective log only the writeset of committed transactions is usually accessible. This difference prevents the use of the reflective log approach for eager replication. This is particularly true if the writeset is needed for conflict detection which has to happen before transaction commit in eager protocols. Another implication of reflective logs is that access to the log is done by reading the log file. Since the log file is typically stored in a separated disk for efficiency reasons (so the head is always on the right track), the access by the replication middleware to the log file will reduce the benefits of this optimization by occasioning head movements.

Reflective logs are already provided by some commercial databases such as Microsoft SQLServer, Oracle, and IBM DB2. This reflective interface is usually known as log sniffer/reader/mining. Log sniffers are usually used for lazy replication in which updates are propagated as separate transactions.

4.5 Reflective Concurrency Control

The last reflective meta-interface we explore is the one related to concurrency control. The first possibility that one can consider is to have a full-fledge meta-model of concurrency control. For a locking concurrency control every lock request would be reified to the meta-model giving it opportunity to keep track of actual conflicts and wait-for relationships. Lock releases due to transaction abortion or commitment would be reified as well, so the meta-model can keep the information up-to-date. This meta-model would be very powerful but unfortunately it is very expensive, since a very high number of lock requests and releases take place during a transaction. Hence, this approach is inefficient and complex to implement. Therefore, it is necessary to resort to slim meta-models with lower overheads.

For this, one has to understand what is really needed by the middleware level. In asymmetric schemes a transaction is first executed at one replica and then at the others. Two transactions that are executed locally at one replica are typically scheduled by the concurrency control mechanism of this local base-level database system. However, conflicts between transactions that are local at different replicas can only be detected optimistically when the writeset of one of the transaction arrives at the other replica. Typically, the transaction whose writeset arrives first (e.g., in total order) may commit, the other has to abort. This means, a local transaction should abort when a conflicting writeset arrives. For this to happen, however, the middleware (1) needs to know about the conflict and (2) be able to enforce an abort. [26] discusses how these things are difficult to achieve in a black box approach, slowing down the replication mechanism. Two simple meta-level interface functions could simplify the problem.

4.5.1 Conflict Reification/Introspection

A minimum interface could provide the meta-model information about blocked transaction. One possibility could be reify the blocking of transactions (a callback mechanism) such that when a SQL request is made to the database and the transaction becomes blocked on a lock request the base-level automatically informs the meta-level about this blocking and the transaction that caused the block (i.e., the one holding a conflicting lock). This enables the meta-level to detect whether a writeset is blocked on a local transaction. Alternatively to a reification mechanism, the meta-level might use introspection via a get-blocked-transactions method to retrieve information about all blocked transactions. This method can be easily implemented in PostreSQL 7.2 in which there is a virtual view table with the transactions blocked awaiting for a lock release and a SELECT statement could be use for this purpose.

4.5.2 Indirect Abort

A second mechanism that is needed is to enable the abort of a transaction at random times. Usually, a client cannot enforce the abort of a transaction in the middle of execution of an operation. Instead, clients can usually only submit abort requests when the database expects input from the client, i.e., when it is not currently executing a request on behalf of the client. However, in the case described above, the replication protocol might need to abort a local transaction at any time. A meta-interface offering such indirect abort would provide a powerful intercession mechanism to the meta-model.

4.5.3 Lock Release Intercession

A transaction usually releases all locks at commit/abort time. Different lock implementations use different mechanisms to grant the released locks to waiting transactions. Lock requests could be waiting in a FIFO or other priority queue. Alternatively, they could be all waken up and given an equal chance to be the next to get the lock granted. However, the replication protocol might like to have its own preference of whom to give a lock, for instance, to guarantee the same locking order at all replicas. Hence, any intercession mechanism such as giving access to the priority queue or allowing the meta-level to decide in which order waiting transactions should be waken up will be useful.

4.5.4 Priority Transactions

Another option to enforce an execution order on the base-level would be a form of indicating a priority level to a transaction. The simplest solution consists in providing a simple extension of the writeset interface that forces the database to apply the writeset in spite of existing conflicting locks. In case that a tuple contained in the writeset is blocked by a different transaction, the database aborts this transaction giving priority to the transaction installing the writeset. This could allow the replication middleware to enforce the same serialization order at all replicas. More advanced, transactions could be given different priority levels. Then the base-level database would abort a transaction if it has a lock that is needed by a transaction with a higher priority. In this case, local transaction could be given the lowest priority, and writesets a higher priority.

5 Evaluation

In this section we aim at providing an evaluation of the cost of the reflective writeset functionality, since it has shown to have a tremendous effect on performance [20]. We consider a wide number of mechanisms to capture the writeset. Our first two implementations are true extensions of the database kernel, and they capture the writeset either in binary or in SQL textual form. Furthermore, we have implemented a trigger based writeset capture, and a log based writeset capture, both of them return the writeset in SQL text format. The binary writeset capture has been implemented in PostgreSQL and was used by Middle-R [19, 33]. The SQL text writeset capture has been implemented in MySQL. The trigger and log based writeset capture were implemented in a commercial database [1].

For applying writesets at remote replicas, we have two implementations. One uses the binary

[1]The license does not allow to benchmark the database naming it.

writeset provided by the binary writeset capture service, the other requires as input a writeset with SQL statements (as provided by the SQL text writeset capture service and the trigger and log based captures).

Since the different reflective approaches were implemented in different databases we show the results separately for each database. We compare the performance of a regular transaction execution without writeset capture with the performance of the same database with writeset capture enabled. This allows us to evaluate the overhead associated with the writeset capture mechanism.

A similar setup is used for evaluating the costs of applying a writeset. In here, we compare the cost of executing an update transaction with the cost of just applying the writeset of the transaction. This enables to measure what is gained by performing asymmetric processing of updates.

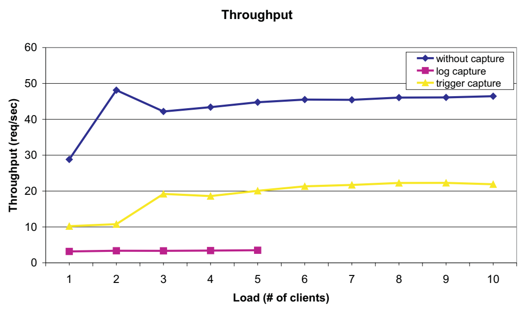
Finally, to show the effect of the different alternatives to capture and apply writesets we derive analytically the scalability for different workloads and number of replicas.

5.1 The Cost of Reflective Writeset Capture

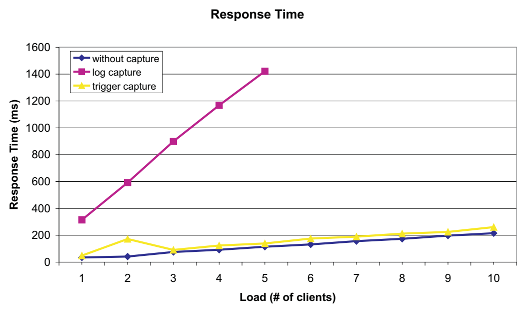
Let us first look at the results of the reflective mechanisms that are typically available in commercial database systems, namely log and trigger based reflection. We have implemented both mechanisms in the same commercial database and compared it to the baseline in this database. The baseline is the execution of a transaction without any capture enabled. The baseline therefore measures the performance of regular transaction execution. For this kind of transaction the maximum throughput that can be obtained is about 45-50 transactions per second. Its response time lies between 35 ms under low load and 205 ms under high load.

The writeset capture via the log mining facilities exhibits a very bad performance. The throughput drops to 3.5 transactions per second what means a drop of over 90%. The response time also worsens in the same line with a sharp increase from 300 to 1400 ms. What is more, it collapses for a number of clients beyond 5. The main reason for this bad behaviour is that the capture of the writeset using log mining creates a very high contention in the log converting it into a bottleneck.

When capturing the writeset via triggers the behaviour is quite different. The throughput suffers considerably and is basically halved, although not



(a) Throughput



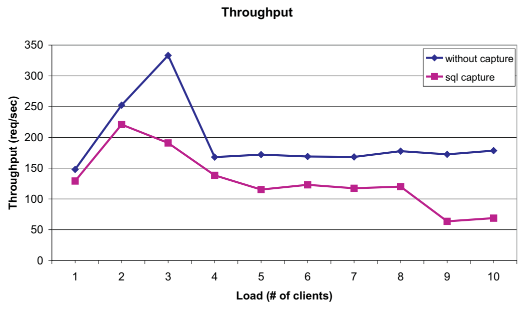
(b) Response time

Figure 3. Log Mining and Trigger Writeset Capture for a Commercial DB

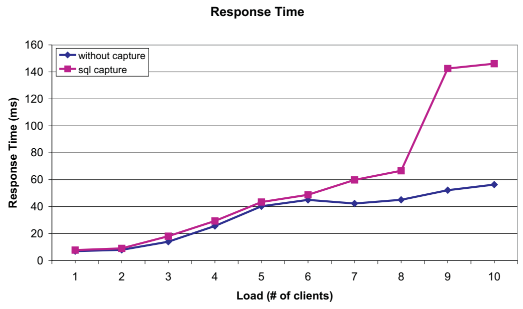
as much as with log mining. The throughput loss is around 55%. This cost for capturing the writeset is very high and as we will see later will result in a reduced scalability. Regarding the response time, it behaves very well, and the transaction latency is not significantly affected by the trigger capture.

The SQL writeset capture was implemented within the MySQL kernel. The SQL writeset capture has a more moderate cost, resulting in an affordable burden. With respect the maximum attainable throughput, the peak for the regular MySQL without capture was over 330 tps, whilst with the SQL writeset capture enabled the peak went down to 220, that means a throughput loss of 33%. If we consider the throughput with 4 clients or beyond, the loss is smaller. MySQL without capture stabilizes in 170 tps, whilst the SQL capture version goes down to 120 tps. The drop in throughput in this case is slightly smaller, around 30%. When the load is close to the saturation point of MySQL, the SQL capture version collapses dropping to a marginal throughput of 60 tps.

The response time of MySQL without capture

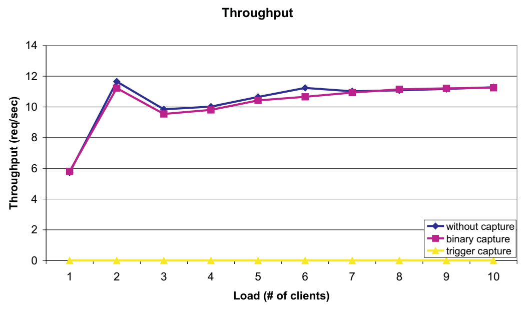


(a) Throughput

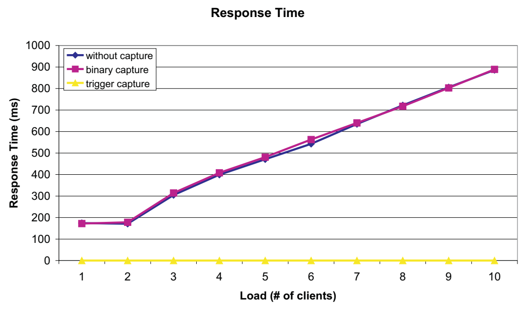


(b) Response time

Figure 4. SQL Writeset Capture for MySQL



(a) Throughput



(b) Response time

Figure 5. Binary Writeset Capture for PostgreSQL

starts at around 8 ms and stabilizes at around 50 ms with high loads. The SQL writeset capture performs reasonably well. With mild loads up to 6 clients the response time remains very close to the one of regular MySQL. When going close to the saturation point the response time starts to increase rapidly reaching 140 ms for high loads.

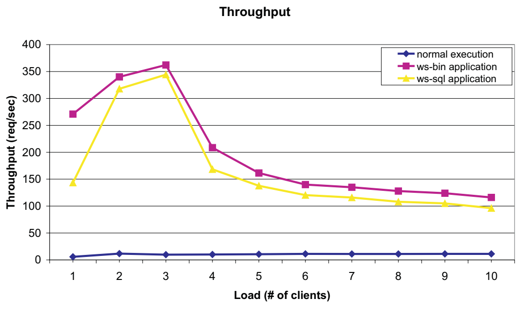
The binary writeset capture has been implemented in PostgreSQL. The throughput of PostgreSQL without capture was around 11 tps. When enabling the binary capture, the throughput remained almost the same without any significant impact. With respect to response time PostgreSQL had a response time of 200 ms with up to two simultaneous clients, and it increased linearly with the number of clients till 900 ms. The binary writeset capture showed the same behaviour as the version without capture. This means that the binary writeset capture resulted in a negligible cost, what is the base to attain a high scalability. Although the version of PostgreSQL used in this experiment achieved generally a very low throughput we do not expect the binary writeset capture to behave worse for higher throughputs since it is a very local task which is not affected by more concurrency in the system.

In summary, trigger and log-based reflection turned out to be too heavyweight with an unacceptable cost in the case of log mining and a very high cost in the case of triggers. The reflective services implemented in MySQL and PostgreSQL have shown to be very lightweight with a quite affordable cost, the binary writeset capture exhibiting an extremely low overhead.

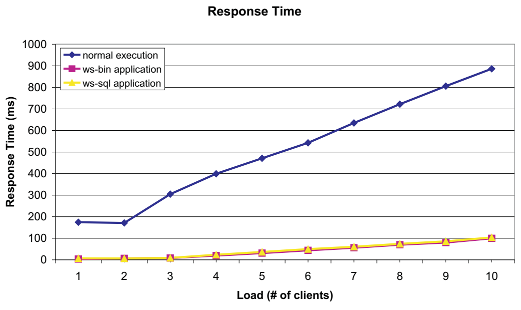
5.2 The Gain of Reflective Writeset Application

In here we evaluate the gains of applying writesets compared to executing the entire transaction. We evaluate two approaches, namely applying binary writesets and SQL writesets. SQL writesets are obtained by most of the capture mechanisms evaluated in the previous section. Binary writesets are only captured with the binary writeset reflective service. We use PostgreSQL for this evaluation since it is the only one in which we have implemented both kinds of writeset application.

Figure 6 shows the performance of fully executing transactions in PostgreSQL, and applying the



(a) Throughput



(b) Response time

Figure 6. SQL and Binary Writeset Application for PostgreSQL

SQL writesets, and the binary writesets. Applying writesets achieves higher throughputs than executing normal transactions. The application of SQL writesets reaches a peak of 345 tps and the application of binary writesets a peak of 362 tps. These peaks are reached when there are around 3 clients submitting writesets concurrently. When there are more clients, the achievable throughput lies between 170-100 tps for SQL writesets, and between 210-115 tps for binary writesets. This means that binary writesets can obtain a throughput between 5-20% higher than SQL writesets. Furthermore, applying writesets can achieve throughputs that are 100-350 times higher than executing the update transactions.

With regard response time, a similar situation is found. The response time for full execution of update transactions in PostgreSQL lies between 170 and 900 ms. Again in stark contrast, the writet applications show a much better performance with response times lying between 7 and 100 ms. That is, a reduction of a 90-95% is obtained in the response time.

5.3 Analytical Scalability of Different Reflective Approaches

In the two previous sections we have evaluated the relative cost of capturing and applying the writeset with different reflective mechanisms. In this section we extend the analytical model for database replication scalability presented in [20] to consider the cost of capturing the writesets, since in that analytical model that cost was considered negligible (what is accurate for binary writesets) and only took into account the ratio between the application of the writesets and the full transaction execution.

Let L be the total processing capacity of the system, i.e., the maximum number of transactions per time unit that can be handled by the aggregation of all sites. Let $L_w = w \cdot L$ and $L_r = (1 - w) \cdot L$ be the load created by update and read transactions, being w the proportion of update transactions in the load. Let t be the processing capacity of a single site in terms of transactions per time unit. t and L exhibit the following relation:

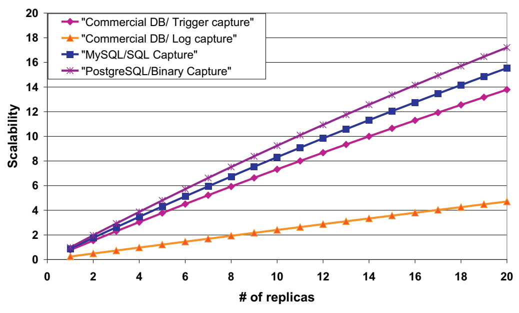
$$t = P_w \cdot L_w + P_r \cdot L_r = L \cdot (w \cdot P_w + (1-w) \cdot P_r)$$

Where P_r and P_w are the probabilities of executing a read and a write operation.

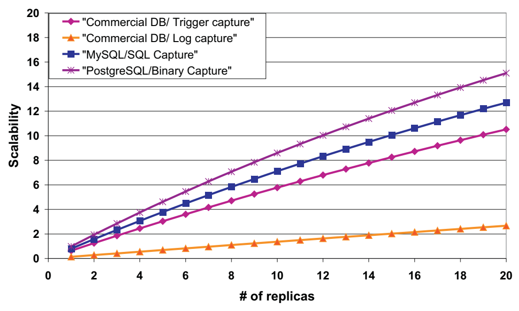
The global scalability of a replicated system is given by the total processing capacity of the entire system (L) divided by the processing capacity of one site (t):

$$so = \frac{L}{t} = \frac{1}{w \cdot P_w + (1-w) \cdot P_r}$$

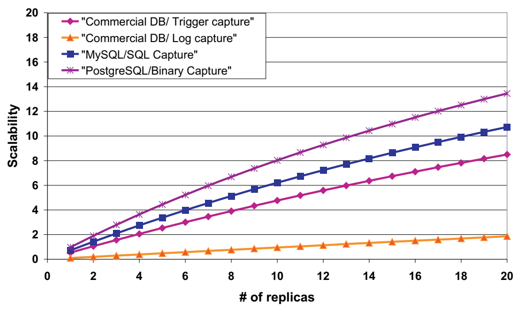
Taking into account that writes are processed asymmetrically the probability for a site to execute a write transaction can be split into: $P_w = P_w^L + P_w^R$, where P_w^L is the probability of being the site fully executing the write transaction (local site for the transaction) and P_w^R is the probability of applying the writeset of an update transaction (remote site for the transaction). Then, we need to distinguish the cost of fully executing an update transaction, which is considered to be 1, the cost of fully executing the transaction including capturing the writeset by means of a particular mechanism, which we denote as *writeset capture overhead* or wco, and the cost of applying the writeset, which we denote as *writeset application overhead* or wao. As seen before, for asymmetric systems, $wco \geq 1$ and $0 < wao \leq 1$. In contrast, $wco = wao = 1$ represents a symmetric system.



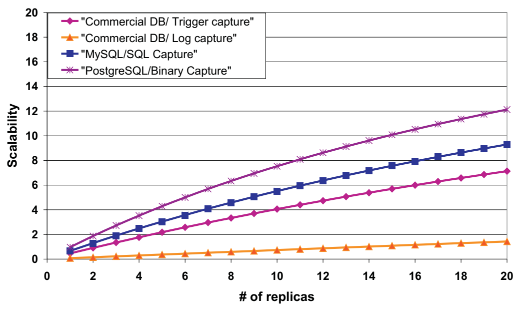
(a) w = 0.25



(b) w = 0.50



(a) w = 0.75



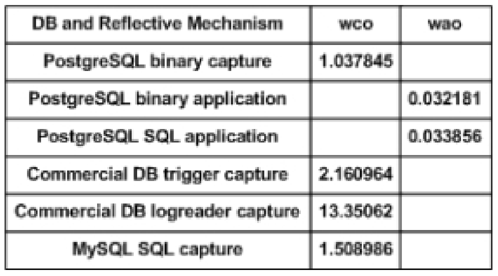
(b) w = 1

Figure 7. Scalability for the Different Reflective Writeset Handling Mechanisms

With this the scalability, sc of a replicated system is:

$$sc = \frac{L}{t} = \frac{1}{w \cdot wco \cdot P_w^L + w \cdot wao \cdot P_w^R + (1-w) \cdot P_r}$$

We can know feed the analytical model with different values of wco and wao extracted from the experimental evaluation performed in the previous sections. wco is obtained as the ratio between the maximum throughput with writeset capture enabled and the maximum throughput for regular transaction execution. Similarly, wao is computed as the ratio between the maximum throughputs of writeset application and regular transaction execution. The computation of these values is made for each implemented reflective writeset mechanism in the different DBs. The so obtained computed values of wao and wco are summarized in Figure 8.



DB and Reflective Mechanism	wco	wao
PostgreSQL binary capture	1.037845	
PostgreSQL binary application		0.032181
PostgreSQL SQL application		0.033856
Commercial DB trigger capture	2.160964	
Commercial DB logreader capture	13.35062	
MySQL SQL capture	1.508986	

Figure 8. Empirical values of writeset capture and application overheads

In Figure 7 we can find the scalability of the different approaches if the percentage of write operations is 25%, 50%, 75% and 100%. The graph shows in the y-axis the relative power of the replicated system compared to a non-replicated system, that is, how many times the throughput of a replicated system multiplies the throughput of a centralized non-replicated system. For instance, a value of 10 in the y-axis, means that the maximum throughput is 10 times the one of a single non-replicated site. The x-axis shows the number of replicas. Since all the graphs are relative, it does not matter that the curves might belong to different databases.

The first observation is that the higher the value of w (percentage of update transactions) the more noticeable the difference among the different approaches. This is intuitive since, the more updates, the more impact has how efficiently they are handled. When comparing all the approaches, it becomes clear that the log mining approach is not an alternative for database replication. Trigger-based writeset capture pays the cost of a heavy weight reflective mechanism. However, it has the advantage that it can be implemented as database application, and hence, does not require changes to the database kernel code. Finally, the two reflective services implemented within the database kernel have the best scalability. Their scalability is very competitive. In the case of SQL capture, the scalability is somewhat lower, since capturing the writeset requires some additional processing for generating the SQL statements. Secondly, there is also the slightly higher cost for applying the writeset that has also some impact on scalability.

If we compare the binary writeset service approach with the others, we can see that for 20 replicas, it provides 10-23% more scalability than the SQL writeset service approach. With respect to the trigger approach, it has 20-41% better scalability. Finally, it beats the log mining approach with an enhancement in scalability of 73-88%.

6 Related Work

Reflection has become a popular paradigm to introduce non-functional concerns with a clean architecture without tangling the code of the regular functionality. In the last decade a number of approaches have been taken to introduce reflection in middleware such as OpenORB [9] and DynamicTAO [25] to disentangle the implementation of nonfunctional cross-cutting concerns from the implementation of the functional aspects. Some new component-based middlewares have been designed from the very beginning to provide reflective components that can be composed into new reflective components [30].

Reflection to introduce transactional semantics has been explored by some researchers. Early approaches relied simply on inheritance (without reflection) to provide flexible transactional semantics [37]. [6] extends a legacy TP-monitor with transactional reflective capabilities to implement advanced transaction models at the meta-level. [41] exploits a reflective Java for introducing transactionality in a declarative fashion for component-based systems.

[1] is a seminal paper on dependability through reflection. The paper takes advantage of reflection in an actor-based language to implement dependable protocols. [17] is also one of the early approaches to implement fault-tolerance exploiting reflection. This paper explores how to perform process replication in three different flavors, active, semiactive and passive, utilizing linguistic reflection in object oriented languages, that is, by means of a meta-object protocol. The use of MOPs to implement fault-tolerant CORBA systems has been studied in [29, 18]. More recently, reflective design patterns have been studied for implementing fault tolerance [27]. Another important topic that has been studied in the context of implementing fault-tolerance adoptive reflective approaches is what happens in complex systems such as a middleware on top of operating systems [38].

7 Conclusions

In this paper we have proposed a wide set of lightweight reflective mechanisms for databases that enable to perform replication at the middleware level. These mechanisms have explored all the main functionalities of the database, database connectivity, request handling, concurrency control and logging. Some of the reflective mechanisms are already widely used, others are quite novel and an efficient implementation would be very useful for middleware based replication. From there, a thorough comparison, both empirically and analytically, of different implementations for writeset capture and application has been performed, since this reflective mechanism has proven to have a high impact on the scalability of database replication. The main conclusion has been that the most promising reflective mechanisms are those that capture the writeset within the database kernel either in binary of SQL form.

References

[1] G. Agha, S. Frolund, R. Panwar, and D. Sturman. A Linguistic Framework for Dynamic Composition of Dependability Protocols. In *Proc. of DCCA-3*, 1993.

[2] Y. Amir and C. Tutu. From Total Order to Database Replication. In *ICDCS*, 2002.

[3] C. Amza, A. L. Cox, and W. Zwaenepoel. Scaling and Availability for Dynamic Content Web Sites, 2002.

[4] C. Amza, A. L. Cox, and W. Zwaenepoel. Conflict-aware scheduling for dynamic content applications. In *USITS*, 2003.

[5] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *Middleware*, 2003.

[6] R. S. Barga and C. Pu. A Reflective Framework for Implementing Extended Transactions. In S. Jajodia and L. Kerschberg, editors, *Advanced Transaction*

Models and Architectures, pages 63–89. Kluwer Academic Press, 1997.

[7] H. Berenson, P. Bernstein, J. Gray, et al. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

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

[9] G. S. Blair, G. Coulson, A. Andersen, et al. *IEEE Distributed Systems Online*, 6(2), 2001.

[10] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. In *ACM SIGMOD*, 1999.

[11] Y. Breitbart and H. F. Korth. Replication and consistency: Being lazy helps sometimes. In *ACM PODS*, 1997.

[12] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *USENIX*, 2004.

[13] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computer Surveys*, 33(4), 2001.

[14] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database replication using generalized snapshot isolation. In *SRDS*, 2005.

[15] S. Gancarski, H. Naacke, E. Pacitti, and P. Valduriez. Parallel Processing with Autonomous Databases in a Cluster System. In *Proc. of CoopIS/DOA/ODBASE*, pages 410–428, 2002.

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

[17] J-C. Fabre, V. Nicornette, T. Pérennou, R. J. Stroud, and Z. Wu. Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming. In *Proc. of FTCS*, 1995.

[18] J-C. Fabre and T. Pérennou. A Metaobject Architecture for Fault-Tolerant Distributed Systems: the FRIENDS Approach. *IEEE Transactions on Computers*, 47:78–95, 1998.

[19] R. Jiménez-Peris, M. Patiño-Martínez, and G. Alonso. Non-intrusive, parallel recovery of replicated data. In *IEEE Symp. on Reliable Distributed Systems (SRDS)*, 2002.

[20] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication. *ACM Transactions on Database Systems*, 28(3), 2003.

[21] J.M. Milan, R. Jiménez-Peris, M. Patiño-Martínez, and B. Kemme. Adaptive middleware for data replication. In *Middleware*, 2004.

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

[23] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM TODS*, 25(3), 2000.

[24] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *ICDCS*, 1999.

[25] F. Kon, M. Román, P. Liu, T. Yamane, L. C. Magalhaes, and R. H. Campbell. Monitoring, Security, and Dynamic Configuration with the DynamicTAO Reflective ORB. In *Middleware*, 2000.

[26] Y. Lin, B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. Middleware based data replication providing snapshot isolation. In *SIGMOD*, June 2005.

[27] L.L. Ferreira and C.M.F. Rubira. Reflective design patterns to implement fault tolerance. In *OOPSLA Workshop on Reflective Programming*, 1998.

[28] P. Maes. Concepts and Experiments in Computational Reflection. In *Proc. of Int. Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1987.

[29] M.O. Killijian, J-C. Fabre, J.C. Ruiz-Garcia, and S. Chiba. A Metaobject Protocol for Fault-Tolerant CORBA Applications. In *SRDS*, 1998.

[30] ObjectWeb. Fractal, http://fractal.objectweb.org.

[31] E. Pacitti, M. T. Özsu, and C. Coulon. Preventive multi-master replication in a cluster of autonomous databases. In *Euro-Par*, 2003.

[32] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable Replication in Database Clusters. In *Proc. of Distributed Computing Conf., DISC'00. Toledo, Spain*, volume LNCS 1914, pages 315–329, Oct. 2000.

[33] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. MIDDLE-R: Consistent database replication at the middleware level. *ACM Transactions on Computer Systems*, 23(4):375–423, Nov. 2005.

[34] F. Pedone, S. Frolund, R. Guerraoui, and A. Schiper. The Database State Machine Approach. *Distributed and Parallel Databases*, 14(1), 2003.

[35] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Middleware*, 2004.

[36] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. Strong Replication in the GlobData Middleware. In *Workshop on Dependable Middleware-Based Systems (part of DSN02)*, pages 503–510, 2002.

[37] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An Overview of Arjuna: A Programming System for Reliable Distributed Computing. *IEEE Software*, 8(1):63–73, Jan. 1991.

[38] F. Taiani, J-C. Fabre, and M-O. Killijian. Towards Implementing Multi-Layer Reflection for Fault-Tolerance. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN)*, San Francisco, June 2003.

[39] M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso. Database replication techniques: A three parameter classification. In *SRDS*, 2000.

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

[41] Z. Wu. Reflective Java and a reflective component-based transaction architecture. *In ACM OOPSLA'98 Workshop on Reflective Programming in Java and C++*, 1998.