COLUP: The Cautious Optimistic Lazy Update Protocol[*]

Luis Irún-Briz, Francesc D. Muñoz-Escoí, Josep M. Bernabéu-Aubán
Instituto Tecnológico de Informática
Universidad Politécnica de Valencia, 46071 Valencia, Spain
Email: {lirun,fmunyoz,josep}@iti.upv.es

Abstract

Lazy update protocols have proven to have an undesirable behavior in respect to the abortion rate in scenarios with high degree of access conflicts. In this paper, we present the problem of the abortion rate in such protocols from the statistical perspective, in order to provide an expression that predict the probability of an object to be out of date during the execution of a transaction.

It is also presented a pseudo-optimistic algorithm that makes use of this expression, allowing the system to reduce the abortion rate, and maintaining stable, or even improve, the number of processed transactions per second.

The proposal is validated by means of an empirical study of the behavior of the algorithm and the expression. Finally, an adaptative technique to adjust the algorithm behavior is also described, in order to optimize its improvements.

Keywords: distributed databases, concurrency control, database replication

1 Introduction

Fault tolerance and performance enhancement are two of the most important advantages of distributed systems. To achieve this, replication has been proven to be a powerful and versatile approach. In the area of distributed databases, consistency requirements introduce a new parameter in the problem, and the proposed solutions used to be centered in it.

In distributed databases, a set of algorithms for concurrency control [1] (known as *update everywhere* protocols) tries to maximize the use of every node in a replicated system, by means of the execution of each transaction in a unique node in the system. This makes possible for every node in the system to execute a different transaction at the same time, but

[*]This work has been partially supported by the EU grant IST-1999-20997 and the Spanish grant TIC99-0280-C02-01.

it becomes necessary, once the transaction is locally committed, to propagate its changes to the rest of the system. These techniques provide an improvement of the performance, but make it necessary to introduce any kind of consistency control in the system.

Optimistic consistency protocols allow a transaction to use every object needed without any control, maintaining a *timestamp* [2] (or *version number*) for each accessed object. When a transaction tries to be committed, then it becomes necessary to reach a consensus with the other involved nodes about the absence of conflicts for the objects accessed by the transaction. This consensus is commonly performed as a unique ”voting algorithm” at commit time. If any node in the system holds in its local database a more recent version of any object accessed by the transaction, then the transaction must be aborted. The main disadvantage of optimistic techniques consists of the increase of the probability for a transaction to be aborted in commit time.

On optimistic approaches, the update propagation is used to be done using the ”constant interaction” approach, performing all the propagation at a time. A transaction trying to commit must ensure that every object accessed in its execution has the adequate version, aborting the transaction if the consensus cannot be achieved.

When a transaction succeeds its commit phase, it must propagate its changes. If this propagation is completely performed inside the commit phase, then the update propagation is usually named *eager update propagation*. On the other hand, it is possible, in order to reduce the overhead in the system, to delay this propagation beyond the commit phase [3].

If this is done, the consensus made at commit time about the object versions will also refer to the check that every object accessed by the transaction held an updated value in the local database when it occur, making it necessary for the consistency protocol to abort a potentially higher number of transactions.

The use of laziness [4] can improve the performance in certain scenarios, because it makes possible to broadcast a lower number of updates when read operations are mainly local. In such scenarios, each node in the system trends to access to a concrete set of objects. So, these objects are often modified by transactions executed in this node, and there are a lower number of foreign transactions (i.e. executed in other nodes) that access to these objects.

The conjunction of optimistic and lazy approaches increases the transaction abortion rate, and makes such systems unusable when the system load grows up, because the higher number of object-collisions will produce a higher number of aborts. Moreover, this growth of the abort number will degrade the performance of the system, because the time spent in aborted transactions becomes useless [4]. This inconvenience of lazy and optimistic approaches can be attenuated by means of a more careful design of the concurrency protocol.

This paper provides an statistical study of the increase in the abortion rate caused by the use of lazy replication, and uses these results to make a proposal of a new algorithm to solve the concurrency control in a distributed transactional system (e.g. distributed databases), making use of the lazy paradigm, and maintaining the optimistic approach in a certain degree. In addition, we present in this paper the results of a simulation of the algorithm, showing the adequacy of our approach, and providing an exhaustive description of the behavior of our model.

To this end, the following section describes a basic optimistic lazy protocol for the concurrency control, and discusses its behavior in terms of transactions per second, abortion rate, etc. Section 3 performs an analysis of the abortion rate in such protocols, and provides an statistical expression of this rate. In section 4 a modification of the basic algorithm is presented, introducing the results presented in section 3 in order to improve the behavior of the abortion rate. An exhaustive study of our proposal is shown in section 5, where the simulated system is detailed, and the obtained results are described. Section 6 compares this protocol with other systems. Finally, section 7 gives some conclusions.

2 The Optimistic Lazy Protocol

The basic consistency protocol presented in this section is a variation of one of the consistency protocols currently implemented in the GlobData Project[5] (described in [6, 7]).

In the GlobData project, a software platform is used to support database replication. This platform, is called COPLA, and provides an object-oriented view of a network of relational DBMSs. The COPLA architecture is structured in three different layers, which interact using CORBA interfaces, enabling the placement of each layer in a different machine. Thus, multiple applications (running in different nodes) may access the same database replica using its local ”*COPLA Manager*”. The COPLA Manager of each node propagates all updates locally applied to other database replicas using its replication management components.

The basic optimistic Lazy Protocol implemented in GlobData is named ”*Lazy Object Multicast Protocol*” (LOMP). It will be described in 2.2; further clarifications are presented in the subsections thereafter.

2.1 Node Roles

Considering a given session that tries to commit, the nodes involved in its execution may have four different roles:

- *Active node*. The node where the COPLA Manager that has directly served the session's execution is placed.
- *Owner node*. For a concrete object, the node where this object was created. During

the consensus process performed at commit time of a session, the owner of an object will be asked to allow this session to complete the commit. Thus, it is the manager for the *access confirmation requests* sent by the active nodes at commit time. The management of these access confirmation requests is similar to lock management, but at commit time. These requests are detailed in section 2.2.

We will denote that a node N_k owns an object o_i with the expression $N_k = own(o_i)$.

- *Synchronous nodes*. If one of our goals is to provide fault tolerance, it becomes necessary a set of nodes that provides guarantees about the last version written for a certain object. So, for each session that is committing, the owner of each written object must multicast this update to a set of synchronous nodes, within the commit phase.

  We will denote that a node N_k is a synchronous replica of an object o_i with the expression $N_k \in S(o_i)$.

- *Asynchronous nodes*. For an object, all the other nodes that have a COPLA Manager replicating the database. In these nodes, the session updates will be eventually received.

  We will denote that a node N_k is an asynchronous replica of an object o_i with the expression $N_k \in A(o_i)$. Note that: $own(o_i) \in S(o_i), \text{ and } A(o_i) \cap S(o_i) = \emptyset$.

2.2 Protocol

As described above, the GlobData-LOMP consistency protocol multicasts object updates to all synchronous nodes when a session is allowed to commit. Consistency conflicts among sessions are resolved with an optimistic approach, using object versions. To this end, the protocol uses some meta-data tables in the database where the current object versions can be stored.

This protocol can be classified [1] as *optimistic constant interaction*, performing an *update everywhere with lazy propagation*, and with *voting termination*. A complete description can be found in [8].

3 Statistical Analysis of the Abortion Rate

In this section, a set of expressions describing the abortion rate will be presented. In the presentation, we will use a set of basic assumptions about the nodes in the system, the sessions executed, and the objects accessed by the sessions.

3.1 Assumptions for the Model

These assumptions are the following:

- There are K COPLA managers running in the system. Each one can be considered as a ”node” $N_{k=1..K}$.
- Each node in the system manages a complete replica of the database. This database contains N different objects.
- A session S can be written as a tuple $S = [R(S), W(S)]$ where:
  - $R(S)$ is the set of objects read by the session S. It is also named ”readset of S”. $R = \{r_i\}_{i=1..|R|}$
  - $W(S)$ is the set of object written by the session S (or ”writeset of S”). $W = \{w_i\}_{i=1..|W|}$
- We assume that $W(S) \subseteq R(S)$. And the objects contained in $R(S)$ and $W(S)$ can be expressed as tuples: $o_i = [id(o_i), ver(o_i), val(o_i), t(o_i), ut(o_i)]$ where:
  - $id(o_i)$ is a unique identifier for the object. The identifier includes the owner node (the node where the object was originated), a sequential number established within the context of each node, and other information used to calculate conflicts.
  - $ver(o_i)$ is the version number of the accessed object.
  - $val(o_i)$ is the value read (or written) by the session for the object.
  - $t(o_i)$ is the local time the object was accessed at.
  - $ut(o_i)$ is the local time the object was more recently updated at.

3.2 Analysis

We can define the probability for a session S to be aborted as:

$$PA(S) = 1 - (PC_{conc}(S) \cdot PC_{outd}(S))$$

- being $PC_{conc}(S)$ the probability for S to conclude without concurrency conflicts.
- and $PC_{outd}(S)$ the probability for S to conclude without ”outdate conflicts”.

The goal of this section is to determine the value of $PC_{outd}(S)$, in order to predict the influence our LOMP has into the abortion rate in the system.

To this end, we can calculate this probability in terms of the probability of a session to conclude with conflicts produced by the access to outdated objects ($PA_{outd}(S)$):

$$PC_{outd}(S) = PC_{outd}(r_i)^{nr}$$

taking nr as the number (in mean) of objects read by a session,

$$PC_{outd}(S) = PC_{outd}(r_i)^{\frac{\sum_k nr_k}{K}} \tag{1}$$

moreover, $PC_{outd}(r_i)$ is the probability for an object r_i to have an updated version in the instant the session accesses it. This probability can be expressed in terms of the probability for an object to be accessed in an outdated version ($PA_{outd}(r_i)$) as:

$$PC_{outd}(r_i) = 1 - PA_{outd}(r_i)$$

Now, let's see the causes of these conflicts: we took r_i as an asynchronous object in the active node that has not been updated since $ut(r_i)$; the outdated time for r_i satisfies $\delta(r_i) = t(r_i) - ut(r_i)$; it can be seen that $PA_{outd}(r_i)$ depends on the number of sessions that write r_i having the chance to commit during $\delta(r_i)$. Let PA_{T,r_i} be the probability for another concurrent session T (that has success in its commit) to finalize with $r_i \in W(T)$. Then,

$$PA_{outd}(r_i) = (PA_{T,r_i})^C$$

where C depends on the number of write-sessions that can be committed in the system during $\delta(r_i)$ ...

$$PA_{outd}(r_i) = (PA_{T,r_i})^{\sum_k wtps_k \times \delta(r_i)} \tag{2}$$

Now, we can rewrite PA_{T,r_i} as $1 - PC_{T,r_i}$, being PC_{T,r_i} the probability for a concurrent write-session T (that has success in its commit) to finalize with $r_i \notin W(T)$. That is:

$$PC_{T,r_i} = P[r_i \notin W(T)]$$

if $W(T) = \{w_1, w_2, \ldots w_{nw(T)}\}$, then in mean, $PC_{T,r_i} = \left(P[r_i \neq w_{j\in\{1..nw\}}]\right)^{nw}$ taking nw as the mean of $|W(T)|$ for every write-session in the system.

$$PC_{T,r_i} = \left(P[r_i \neq w_{j\in\{1..nw\}}]\right)^{\frac{\sum_k nw_k}{K}} \tag{3}$$

The next step consists of the calculation of $P[r_i \neq w_{j\in\{1..nw\}}]$. To do this, we must observe the number of objects in the database (N). The probability that an accessed object is a given one is $\frac{1}{N}$, thus:

$$P[r_i \neq w_{j\in\{1..nw\}}] = 1 - \frac{1}{N}$$

Finally, the complete expression for PA_{outd} can be rewritten as follows:

$$PA_{outd}(r_i) = \left(1 - \left(1 - \frac{1}{N}\right)^{\frac{\sum_k nw_k}{K}}\right)^{\sum_k wtps_k \times \delta(r_i)} \qquad (4)$$

This expression provides a basic calculation of the probability for an object access to cause the abortion of the session by an out-of-date access.

The expression can be calculated with a few parameters. Only nw_k and $wtps_k$ must be collected in the nodes of the system in order to obtain the expression. Thus, it becomes possible for a node to estimate the convenience for an object to be locally updated before being accessed by a session. This estimation will be performed with a certain degree of accuracy, depending on the ”freshness” of the values of nw_k and $wtps_k$ the node has. The way the expression can be used, and an adequate mechanism for the propagation of these parameters will be presented in section 4.

4 COLUP: The Cautious Optimistic Lazy Update Protocol

An expression for the probability for an object to cause the abortion of a session has been presented in section 3. This expression can be used to predict the convenience for a session to ensure that an object that is asynchronously updated has a recent version in the local database.

To this end, it becomes necessary to introduce a new request in the basic algorithm. An active node for a session will send an ”Update request” to the owner of an object, in order to get an updated version of such object.

This update request message can be sent from an active node to an owner node along the session execution, in order to maintain updated (in a certain degree) the objects being about to be accessed by the session.

This technique will reduce the probability of abortion caused by the accesses to outdated objects within the sessions. This improvement is based in the outdated time, shown as $\delta(o_i)$ in the expression for $PA_{outd}(o_i)$.

4.1 Modification of the LOM Protocol

The way the COPLA manager decides to send or not an update request for an object that is about to be accessed is quite simple:

1. During the life of a session, an access to an object o_i is requested by the application. This access is intercepted by the COPLA manager of the active node, and then the

probability of being outdated is calculated for the object o_i. If the active node is a synchronous (or owner) node for o_i, this probability is 0. This is because the protocol ensures that incoming updates of the object will abort every transaction conflicting with the updated object.

In the active node is an asynchronous node for o_i, then the COPLA manager will use the expression of $PA_{outd}(o_i)$. To perform this calculation, it is needed $\delta(o_i)$: the time elapsed from the last update received for o_i.

2. If this probability exceeds a certain threshold T_c, then the COPLA manager sends the update request to the owner of o_i. If the threshold is not reached, the protocol continues as described in section 2.2. Section 5 will present an empirical approximation to determine an optimum value for the threshold T_c.

3. In other case, after the COPLA manager allows the session to continue, it waits for the update request response. This response indicates whether the local version of o_i is updated, or outdated (then, the response contains the newest version for the object). If the local version must be updated, then the update is applied in the local database, and the update time is also written down.

4. Once the COPLA manager has ensured that o_i is updated, the required access to the object can continue.

By forcing to update an object before the session accesses it, the COPLA manager decreases the value of δ_i, to the length of the session. Thus, the chance for a session to success the commit phase is higher.

This technique implies that every update performed in the asynchronous nodes of an object must include a timestamp of the instant the update is performed at. Note that it is not necessary to use global time, but it is only necessary the local time for each node.

5 Results

We have validated the algorithm presented above by implementing a simulation of the system. In this simulation, we have implemented nodes that concurrently serve sessions, accessing to different objects of a distributed database.

We have also modeled the concurrency control, and the lazy update propagation protocol used by LOMP.

5.1 Assumptions

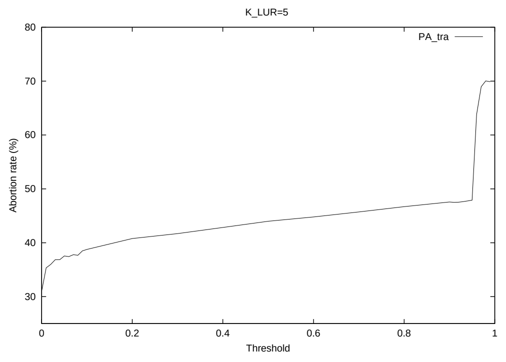
The assumptions for the implementation of the simulation are compatible with the ones taken for the model calculation, and the values have been established to increase the number of conflicts produced by the transactions executed in the system:

- There are 4 nodes in the system, each holding a full replica of the database, that contains 20 objects. Each node executes transactions, accessing the database.
- For every object, a local replica holds the value of the object, and the version corresponding to the last modification of the object.
- There are three types of transactions:
  1. ”read-only” transactions: reads three objects.
  2. ”read-write” transactions: reads three objects, then writes these objects.
  3. ”read&read-write” transactions: reads six objects, then writes three of them.
- The probability for a transaction to be of these types is 0.2, 0.4, and 0.4 respectively.
- The model supports the locality of the access by means of the probability for an accessed object to be owned by the node where the transaction is started. For read-only transactions, this is 1/4, (as the system contains 4 nodes, this models no locality for read-only transactions). For read-write transactions, and read&read-write transactions, the probability is 3/4.
- The cost of each operation is shown in time units (t.u.):
  - Read operation in a local database: $LR = 0.01\ t.u.$
  - Write operation in a local database: $LW = 0.02\ t.u.$
  - Cost of a local-update request: $LUR = K_{LUR} \times LR, K_{LUR} = 5$
  - Cost of a confirmation request: $CR = 0.6 \times LUR$
- In order to provide a complete study of the algorithm, subsection 5.5 will include different executions of the simulation, with values for the constant $K_{LUR} \in [1..10]$.
- The simulation time has been fixated at 1,422 t.u., discarding the first 2 t.u. as stabilization time for the simulation. This allows to start up to 60,000 transactions.

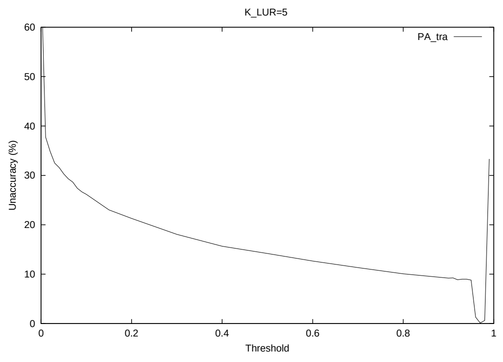
Due to the characteristics of the expression shown in section 3 the algorithm becomes very sensible for low values of the established threshold. To relax this, we have applied an escalation to the basic expression of $PA_{outd}(o_i)$. The goal of this escalation is to distribute the values provided by $PA_{outd}(o_i)$ in a more homogeneous way. The scaled function $PA_{tra}(o_i)$ just expands the main range of values taken by $PA_{outd}(o_i)$ into a wider scale, and compacts the residual range. The rest of the section studies each parameter of the protocol using the normalized expression (PA_{tra}).

5.2 Abortion Rate

The main goal of the COLUP algorithm consists of the reduction of the abortion rate. This reduction is achieved by the adjustment of the value of a threshold. When the probability for an object to be outdated ($PA(o_i)$) is greater than this threshold, an update request is sent to the owner of the object. This reduces the probability of abortion of the transaction, because it is reduced the time the objects are out of date in each node.



(a) Abortion rate



(b) Inaccuracy

Figure 1: Evolution of the protocol for different thresholds

Figure 1(a) shows the evolution of the abortion rate for different thresholds.

In one extreme, when the threshold is fixated to 0, the protocol will try to update every accessed object. This causes the lower abortion ratio.

In this case, the behavior of the protocol is similar to an eager protocol. In such algorithms, every object update is always propagated to each node in the system. The disadvantage of this approach appears when the same object is updated several times before another node performs an access to it. In contrast, in the case of COLUP with $Threshold = 0$, the disadvantage consists of the unnecessity of performing update requests when the accessed object is already up-to-date.

On the other hand, when a $Threshold = 1$ is fixated, the protocol behavior is the same than a pure lazy protocol: no update is performed, unless a transaction is aborted.

5.3 Accuracy of the Prediction

When the expression exceeds the established threshold for an object, and an update request is sent, it is possible for the response of this request to contain the same version for the requested object (e.g. when the object was, in fact, up to date). We name this situation ”Inaccurate prediction”.

The more accurate the predictions are, the less overhead the algorithm introduces in the system. This accuracy of the predictions will be given by the fixated threshold: higher values for the threshold should provide more accurate predictions, but, as seen in the previous section, the abortion rate suffers an increase.

The figure 1(b) shows the evolution of the inaccuracy of the prediction, for different values of the threshold. The figure shows a gradual decrease in the inaccuracy when the threshold is increased. But, for higher values of the threshold, the number of update requests is very low, and the corpus of the experiment is not valid.

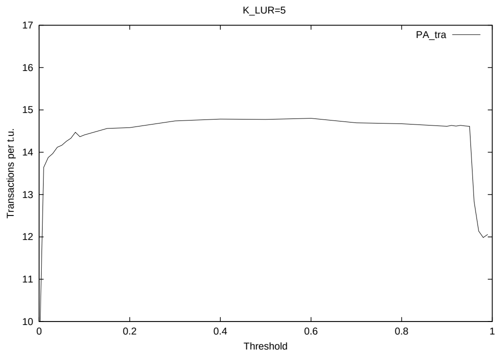
In general, it can be observed that higher values for the threshold increase the accuracy of the prediction.

5.4 Performance

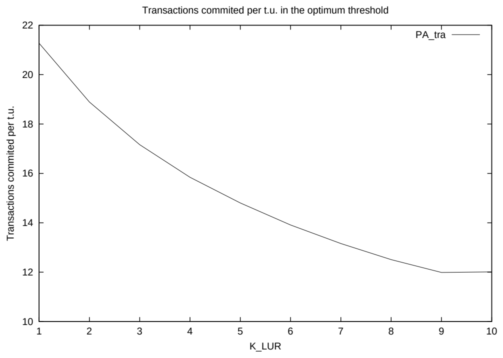
In the middle point, it can be found a balance between accuracy and abortion rate. In this point, the time spent in aborted transactions will be low, and the overhead introduced by the updates will be useful in the most of the cases. Then, the number of transactions committed per second should be maximized, keeping low the abortion rate.

In figure 2(a), the performance of the system is analyzed through the number of transactions served per time unit. It is shown how the number of transactions committed when the COLUP gives a pure lazy behavior (i.e. $Threshold = 1.0$) is higher than the performance obtained for a ”paranoiac COLUP” (i.e. $Threshold = 0.0$). In the middle point, it can be found a threshold maximizing the performance.

In our experiments, it has been proven that the evolution of the system for different thresholds evidences an optimum value, maximizing the performance of the system. In addition, the abortion rate when this threshold is used, can be kept below to a reasonable limit.



(a) Number of transactions committed per t.u. (c.t.p.s.) for different thresholds



(b) Evolution of c.t.p.s. in the optimum for different K_{LUR}

Figure 2: Productivity of the protocol

5.5 Optimum Threshold

In order to evaluate the behavior of the algorithm for different configurations, we have considered that the most relevant parameter is the cost introduced by the *LocalUpdate* messages. For higher values, the better option may be to avoid any local update, because the cost of this update will be higher than the time spent in the transaction if the the transaction is finally aborted.

To analyze this, we have repeated the simulation giving different values to this cost (i.e. making K_{LUR} take values from $[1..10]$). For each K_{LUR}, we have taken the optimum value of the thresholds, in terms of committed transactions per second. Figure 2(b) shows how the reduction of the cost of an update request (lower values of K_{LUR}) allows the protocol to improve the system with better performances.

6 Related Work

Pessimistic consistency control for distributed databases [9] uses the principle of ”locks” in order to avoid concurrent transactions to access to the same object in an inadequate mode. This approach minimizes the number of aborted transactions, but the overhead of the management of the locks degrades the performance of the system.

In the other hand, the traditional approach for optimistic consistency control was presented in [10, 11], and its main advantage consists on the reduction of the blocking time of the transactions, using ”timestamps” (or ”timestamps” [2]) as the basis for its implementation. The main disadvantage of optimistic consistency protocols consists of the increase in the abortion rate.

Lazy update protocols, introduced in [12], presents a new approach for the propagation of the updates, in contrast to the traditionally used ”eager replication”. Lazy update protocols takes advantage of the fact that an object can be written several times for different transactions before another transaction tries to read it.

Current work on consistency protocols for replicated databases can be found using either eager [13] or lazy protocols [3]. Pros and cons of both approaches are described in [4], and a good classification of consistency control protocols was presented in [1], according to three parameters: server architecture, server interaction, and transaction termination.

7 Conclusions

Lazy update protocols have not been widely exploited due to its excessive abortion rate on scenarios with high probability of access conflicts. Nevertheless, such protocols can provide important improvements in the performance of a distributed system, when the abortion rate can be kept low, and the locality of the accesses is appreciable.

We have presented an statistical study of the abortion rate (as disadvantage of lazy protocols), in order to provide an expression for the probability for an accessed object to be out of date ($PA_{outd}(o_i)$), and cause a further abortion of the accessing transaction.

We have also described a lazy update protocol, that uses this expression to obtain a good behavior in respect to the abortion rate, and even improving the performance of the system. In addition, the design of this protocol allows a variety of behaviors from a pure lazy protocol, to a ”paranoiac” behavior.

A complete empirical study of the protocol has been done, validating either the proposed expression and the predicted behavior of the protocol.

References

[1] M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso. Database replication techniques: A three parameter classification. In *Proc. of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, pages 206–217, October 2000.

[2] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, MA, EEUU, 1987.

[3] F. Ferrandina, T. Meyer, and R. Zicari. Implementing Lazy Database Updates for an Object Database System. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 261–272, Santiago, Chile, 1994.

[4] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, Montreal, Canada, 1996.

[5] Instituto Tecnológico de Informática. GlobData Web Site, 2002. Accessible in URL: *http://globdata.iti.es*.

[6] Luis Irún, Francesc Muñoz, Hendrik Decker, and Josep M. Bernabéu-Aubán. Copla: A platform for eager and lazy replication in networked databases. *International Conference on Enterprise Information Systems*, April 2003.

[7] Francesc Muñoz, Luis Irún, Pablo Galdámez, Josep Bernabéu, Jordi Bataller, and Mari-Carmen Bañul. Globdata: A platform for supporting multiple consistency modes. *Information Systems and Databases*, pages 137–143, September 2002.

[8] Francesc Muñoz, Luis Irún, Pablo Galdámez, José Bernabéu, Jordi Bataller, and Mari-Carmen Bañuls. Flexible management of consistency and availability of networked data replications. *Flexible Query Answering Systems*, pages 189–206, October 2002.

[9] P. A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. on Database Sys.*, 9(4):596–615, Dec. 1984.

[10] Maurice Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Trans. on Database Sys.*, 15(1):96–124, March 1990.

[11] E. Rahm. Empirical performance evaluation of concurrency and coherency control protocols for database sharing systems. *ACM Trans. on Database Sys.*, 18(2):333–377, June 1993.

[12] Parvathi Chundi, Daniel J. Rosenkrantz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *Proceedings of the 12th International Conference on Data Engineering*, pages 469–476. IEEE Computer Society, February 1996.

[13] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *International Conference on Distributed Computing Systems*, pages 156–163, 1998.