Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases*

Esther Pacitti
NCE-UFRJ Rio de Janeiro
Brazil
esther.pacitti@inria.fr

Pascale Minet
INRIA Rocquencourt
France
pascale.minet@inria.fr

Eric Simon
INRIA Rocquencourt
France
eric.simon@inria.fr

Abstract

In a lazy master replicated database, a transaction can commit after updating one replica copy at some master node. After the transaction commits, the updates are propagated towards the other replicas, which are updated in separate refresh transactions. A central problem is the design of algorithms that maintain replica's consistency while minimizing the performance degradation due to the synchronization of refresh transactions. We propose a simple and general refreshment algorithm that solves this problem and we prove its correctness. We then present two main optimizations. One is based on specific properties of replicas' topology. The other uses an immediate update propagation strategy. Our performance evaluation demonstrates the effectiveness of this optimization.

1 Introduction

Lazy replication (also called asynchronous replication) is a widespread form of data replication in (relational) distributed database systems [21]. With lazy replication, a transaction can commit after updating one replica copy[1]. After the transaction commits, the updates are propagated towards the other replicas, and these replicas are updated in separate refresh transactions. In this paper, we focus on a specific lazy replication scheme, called *lazy master* replication [10] (also called Single-Master-Primary-Copy replication in [4]). There, one replica copy is designated as the *primary copy*, stored at a *master* node, and update transactions are only allowed on that replica. Updates on a primary copy are distributed to the other replicas, called *secondary copies*. A major virtue of lazy master replication is its ease of deployment [4, 10]. In addition, lazy master replication has gained considerable interest because it is the most widely used mechanism to refresh data warehouses and data marts [5, 21].

However, lazy master replication may raise a consistency problem between replicas. Indeed, an observer of a set of replica copies at some node at time t may see a state I of these copies that can never be seen at any time, before or after t, by another observer of the same copies at some other node. We shall say that I is an *inconsistent* state. As a first example, suppose that two data marts S_1 and S_2 both have secondary copies of two primary copies stored at two different data source nodes[2]. If the propagation of updates coming from different transactions at the master nodes is not properly controlled, then refresh transactions can be performed in a different order at S_1 and S_2, thereby introducing some inconsistencies between replicas. These inconsistencies in turn can lead to inconsistent views that are later almost impossible to reconciliate.

Let us expand the previous example into a second example. Suppose that a materialized view V of S_1, considered as a primary copy, is replicated in data mart S_2. Now, additional synchronization is needed so that the updates issued by the two data source nodes *and* the updates of V issued by S_1 execute in the same order for all replicas in S_1 and S_2.

Thus, a central problem is the design of algorithms that maintain replica's consistency in lazy master replicated databases, while minimizing the performance degradation due to the synchronization of refresh transactions. Considerable attention has been given to the maintenance of replicas' consistency.

Work partially supported by Esprit project DWQ.

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

[1] From now on, we suppose that replicas are relations.

[2] This frequent situation typically arises when no corporate data warehouse has been set up between data sources and data marts. Quite often, each data mart, no matter how focused, ends up with views of the business that overlap and conflict with views held by other data marts (e.g., sales and inventory data marts). Hence, the same relations can be replicated in both data marts.

First, many papers addressed this problem in the context of lazy group replicated systems, which require the reconciliation of updates coming from multiple primary copies [25, 15, 10, 28]. Some papers have proposed to use weaker consistency criterias that depend on the application semantics. For instance, in the OSCAR system [9], each node processes the updates received from master nodes according to a specific weak-consistency method that is associated with each secondary copy. However, their proposition does not yield the same notion of consistency as ours. In [3, 26, 1], authors propose some weak consistency criterias based on time and space, e.g., a replica should be refreshed after a time interval or after 10 updates on a primary copy. There, the concern is not anymore on fast refreshment and hence these solutions are not adequate to our problem. [2, 19, 24] achieve one-copy serializability of synchronous update transactions in a fully replicated database, which guarantees that all conflicting transactions execute in the same order at all sites. They share in common the use of broadcast primitives (e.g., atomic broadcast in [19, 2], reliable broadcast in [24]), as a basis for their replication protocols. However, their notion of consistency is different from ours. Furthermore [19] studies relaxed notions of consistency such as cursor stability and snapshot isolation and proposes efficient protocols to handle node failures and recovery. In [6], the authors give conditions over the placement of secondary and primary copies into sites under which a lazy master replicated database can be guaranteed to be globally serializable (which corresponds to our notion of consistency). However, they do not propose any refreshment algorithm for the cases that do not match their conditions, such as our two previous examples. Finally, synchronization algorithms have been proposed and implemented in commercial systems, such as Digital's Reliable Transaction Router [4], where the refreshment of all secondary copies of a primary copy is done in a distributed transaction. However, to the best of our knowledge, these algorithms do not assure replica consistency in cases like our second above example.

This paper makes three important contributions with respect to the central problem mentionned before. First, we analyze different types of configurations of a lazy master replicated system. A configuration represents the topology of distribution of primary and secondary copies accross the system nodes. It is a directed graph where a directed arc connects a node N to a node N' if N holds a primary copy of some secondary copy in N'. We formally define the notion of correct refreshment algorithm that assures database consistency. Then, for each type of configuration, we define sufficient conditions that must be satisfied by a refreshment algorithm in order to be correct.

As a second contribution, we propose a simple and general refreshment algorithm, which is proved to be correct for a large class of acyclic configurations (including for instance, the two previous examples). We show how to implement this algorithm using system components that can be added to a regular database system. Our algorithm makes use of a reliable multicast with a known upper bound, that preserves a global FIFO order. Our algorithm also uses a deferred update propagation strategy, as offered by all commercial replicated database systems. The general principle of the algorithm is to make every refresh transaction wait a certain "deliver time" before being executed.

As a third contribution, we propose two main optimizations to this algorithm. First, using our correctness results on configurations types, we provide a static characterization of nodes that do not need to wait. Second, we give an optimized version of the algorithm that uses an immediate update propagation strategy, as defined in [23]. We give a performance evaluation based on simulation that demonstrates the value of this optimization by showing that it significantly improves the freshness of secondary copies.

The rest of this paper is structured as follows. Section 2 introduces our lazy master replication framework, and the typology of configurations. Section 3 defines the correctness criteria for each type of configuration. Section 4 describes our refreshment algorithm, how to incorporate it in the system architecture of nodes, and proves its correctness. Section 5 presents our two main optimizations. Section 6 introduces our simulation environment and presents our performance evaluation. Section 7 discusses some related work. Finally, Section 8 concludes.

2 Lazy Master Replicated Databases

We define a (relational) lazy replicated database system as a set of n interconnected database systems, henceforth called *nodes*. Each node N_i hosts a relational database whose schema consists of a set of pairwise distinct relational schemas, whose instances are called relations. A replication scheme defines a partitioning of all relations of all nodes into partitions, called *replication sets*. A replication set is a set of relations having the same schema, henceforth called *replica copies*[3]. We define a special class of replicated systems, called *lazy master*, which is our framework.

2.1 Ownership

Following [10], the *ownership* defines the node capabilities for updating replica copies. In a replication set, there is a single updatable replica copy, called *primary* copy (denoted by a capital letter), and all the other relations are called *secondary* copies (denoted by lower-case letters). We assume that a node never holds the primary copy and a secondary copy of the

[3] A replication set can be reduced to a singleton if there exists a single copy of a relation in the replicated system.

same replication set. We distinguish between three kinds of nodes in a lazy master replicated system.

Definition 2.1 *(Types of nodes).*

1. *A node M is said to be a* master *node iff : $\forall m \in M$ m is a primary copy.*
2. *A node S is said to be a* slave *node iff : $\forall s \in S$ s is a secondary copy of a primary copy of some master node.*
3. *A node MS is said to be a* master/slave *node iff : $\exists ms$ and $ms' \in MS$, such that ms is a primary copy and ms' is a secondary copy.*

Finally, we define the following slave and master dependencies between nodes. A node M is said to be a *master node of* a node S iff there exists a secondary copy r in S of a primary copy R in M. We also say that S is a *slave node of* M.

2.2 Configurations

Slave dependencies define a DAG, called configuration.

Definition 2.2 *(Configuration).*
A configuration *of a replicated system is defined by a directed graph, whose nodes are the nodes of the replicated system, and there is a directed arc from a node N to a node N' iff N' is a slave node of N. Node N is said to be a predecessor of N'.*

In the following, we distinguish different types of configurations. Intuitively, to each configuration will correspond a correctness criteria to guarantee database consistency. In the figures illustrating the configurations, we use integers to represent nodes in order to avoid confusion with the names of the relations that are displayed as annotation of nodes.

Definition 2.3 *(1master-per-slave configuration).*
An acyclic configuration in which each node has at most one predecessor is said to be a 1master-per-slave *configuration.*

This configuration, illustrated in Figure 1(a), corresponds to a "data dissemination" scheme whereby a set of primary copies of a master or master/slave node is disseminated towards a set of nodes. It characterizes for instance the case of several data marts built over a centralized corporate data warehouse.

Definition 2.4 *(1slave-per-master configuration).*
An acyclic configuration in which each node has at most one successor is said to be a 1slave-per-master *configuration.*

This configuration, illustrated in Figure 1(b), corresponds to what is often called a "data consolidation" scheme, whereby primary copies coming from different nodes are replicated into a single node. It characterizes for instance a configuration wherein a data warehouse node (or even, an operational data store node) holds a set of materialized views defined over a set of relations stored by source nodes. In this context, replicating the source relations in the data warehouse node has two main benefits. First, one can take advantage of the replication mechanism to propagate changes from the source towards the data warehouse. Second, it assures the self-maintainability of all materialized views in the data warehouse, thereby avoiding the problems mentioned in [29].

Definition 2.5 *(bowtie configuration).*
An acyclic configuration in which there exist two distinct replicas X_1 and X_2 and four distinct nodes M_1, M_2, S_1 and S_2 such that (i) M_1 holds the primary copy of X_1 and M_2 the primary copy of X_2, and (ii) both S_1 and S_2 hold secondary copies of both X_1 and X_2.

Such configuration, illustrated in Figure 1(c), generalizes the two previous configurations by enabling arbitrary slave dependencies between nodes. This configuration characterizes, for instance, the case of several data marts built over several data sources. The benefits of a replication mechanism are the same as for a data consolidation configuration.

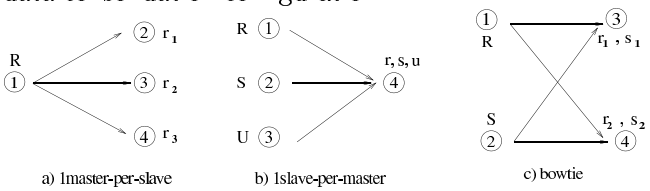

Figure 1: Examples of Configurations

Definition 2.6 *(triangular configuration).*
An acyclic configuration in which there exist three distinct nodes M, MS and S such that (i) MS is a successor of M, and (ii) S is a successor of both M and MS, is said to be a triangular *configuration. Nodes M, MS and S are said to form a* triangle.

This configuration, illustrated in Figure 2(a), slightly generalizes the two first configurations by enabling a master/slave node to play an added intermediate role between a master node and a slave node. This configuration was also considered in [6].

Definition 2.7 *(materialized view).*
A primary copy of a master/slave node MS which is defined as the result of the query over a set of secondary copies of MS is called a materialized view.

Definition 2.8 *(view triangular configuration).*
A derived configuration in which all the primary copies hold by any node MS of any triangle are materialized views of local secondary copies, is said to be a view triangular *configuration.*

This configuration, illustrated in Figure 2(b), characterizes, for instance, the case of two independent data marts defined over the same data warehouse in which one of the data mart replicates some materialized view of the other data mart. Although they overlap, the bowtie and the view triangular configurations are incomparable (none is included into the other).

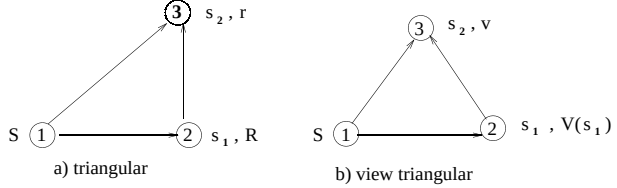

Figure 2: Examples of Configurations

2.3 Transaction Model

The *transaction model* defines the properties of the transactions that access the replica copies at each node. Moreover, we assume that once a transaction is submitted for execution to a local transaction manager at a node, all conflicts are handled by the local concurrency control protocol, in such a way that serializability of local transactions is ensured.

We focus on three types of transactions that read or write replica copies: *update transactions*, *refresh transactions* and *queries*. All these transactions access only local data.

An update transaction is a local user transaction (i.e., executing on a single node) that updates a set of primary copies. Updates performed by an update transaction T are made visible to other transactions only after T's commitment. We denote T_{R_1,R_k} an update transaction T that updates primary copies R_1, R_k. We assume that no user transaction can update a materialized view.

A refresh transaction associated with an update transaction T and a node N, is composed by the serial sequence of write operations performed by T on the replica copies hold by N. We denote RT_{r_1,r_k} a refresh transaction that updates secondary copies r_1, r_k. Finally, *a query transaction*, noted Q, consists of a sequence of read operations on replica copies.

2.4 Propagation

The propagation parameter defines "when" the updates to a primary copy must be multicast towards the nodes storing its secondary copies. The multicast is assumed to be reliable and to preserve the global FIFO order [18]: the updates are received by the involved nodes in the order they have been multicast by the node having the primary copy.

Following [23], we focus on two types of propagation: *deferred* and *immediate*. When using a *deferred* propagation strategy, the sequence of operations of each refresh transaction associated with an update transaction T is multicast to the appropriate nodes within a single message M, after the commitment of T. When using an *immediate* propagation, each operation of a refresh transaction associated with an update transaction T is immediately multicast inside a message m, without waiting for the commitment of T.

2.5 Refreshment

The *refreshment algorithm* defines: (i) the *triggering parameter* i.e., when a refresh transaction is started, and (ii) the *ordering parameter* i.e., the commit order of refresh transactions.

We consider three triggering modes: *deferred*, *immediate* and *wait*. The combination of a propagation parameter and a triggering mode determines a specific update propagation strategy. With a *deferred-immediate* strategy, a refresh transaction RT is submitted for execution as soon as the corresponding message M is received by the node. With an *immediate-immediate* strategy, a refresh transaction RT is started as soon as the first message m corresponding to the first operation of RT is received. Finally, with an *immediate-wait* strategy, a refresh transaction RT is submitted for execution only after the last message m corresponding to the commitment of the update transaction associated with RT is received.

3 Correctness Criteria

In this section, we first formally define the notion of a correct refreshment algorithm, which characterizes a refreshment algorithm that does not allow inconsistent states in a lazy master replicated system. Then for each type of configuration introduced in Section 2, we provide criteria that must be satisfied by a refreshment algorithm in order to be correct.

We now introduce useful preliminary definitions similar to those used in [17] in order to define the notion of a consistent replicated database state. We do not consider node failures, which are out of the scope of this paper. As a first requirement, we impose that any committed update on a primary copy must be eventually reflected by all its secondary copies.

Definition 3.1 *(Validity). A refreshment algorithm used in a lazy master replicated system is said valid iff any node that has a copy of a primary copy updated by a committed transaction T is guaranteed to commit the refresh transaction RT associated with T.*

Definition 3.2 *(Observable State). Let N be any node of a lazy master replicated system, the observable state of node N at local time t is the instance of the local data that reflects all and only those update and refresh transactions committed before t at node N.*

In the next definitions, we assume a global clock so that we can refer to global times in defining the notion of consistent global database state. The global clock is used for concept definition only. We shall also use the notation $I_t[N](Q)$ to denote the result of a query transaction Q run at node N at time t.

Definition 3.3 *(Quiescent State). A lazy master replicated database system is in a quiescent state at a global time t if all local update transactions submitted before t have either aborted or committed, and all the refresh transactions associated with the committed update transactions have committed.*

Definition 3.4 *(Consistent Observable State). Let N be any node of a lazy master replicated system D. Let t be any global time at which a quiescent state of D is reached. An observable state of node N at time $t_N \leq t$ is said to be consistent iff for any node N' holding a non-empty set X of replica copies hold by N and for any query transaction Q over X, there exists some time $t_{N'} \leq t$ such that $I_{t_N}[N](Q) = I_{t_{N'}}[N'](Q)$.*

Definition 3.5 *(Correct Refreshment Algorithm for a node N). A refreshment algorithm used in a lazy master replicated system D, is said to be correct for a node N of D iff it is valid and for any quiescent state reached at time t, any observable state of N at time $t_N \leq t$ is consistent.*

Definition 3.6 *(Correct Refreshment Algorithm). A refreshment algorithm used in a lazy master replicated system D, is said to be correct iff it is correct for any node N of D.*

In the following, we define correctness criteria for acyclic configurations that are sufficient conditions on the refreshment algorithm to guarantee that it is correct. Due to space limitation, the proofs of all propositions are omitted and can be found in the long version of this paper [22].

3.1 Global FIFO Ordering

For 1master-per-slave configurations, inconsistencies may arise if slaves can commit their refresh transactions in an order different from their corresponding update transactions. Although in 1slave-per-master configurations, every primary copy has a single associated secondary copy, the same case of inconsistency could occur between the primary and secondary copies. The following correctness criterion prevents this situation.

Definition 3.7 *(Global FIFO order). Let T_1 and T_2 be two update transactions committed by the same master or master/slave node M. If M commits T_1 before T_2, then at every node having a copy of a primary copy updated by T_1, the refresh transaction associated with T_2 can only commit after the refresh transaction associated with T_1.*

Proposition 3.1 *If a lazy master replicated system D has an acyclic configuration which is neither a bowtie nor a triangular configuration, and D uses a valid refreshment algorithm meeting the global FIFO order criterion, then this refreshment algorithm is correct.*

A similar result was shown in [6] using serializability theory.

3.2 Total Ordering

Global FIFO ordering is not sufficient to guarantee the correctness of refreshment for bowtie configurations. Consider the example in Figure 1(c). Two master nodes, node 1 and node 2, store relations $R(A)$ and $S(B)$, respectively. The updates performed on R by some transaction T_R: insert $R(A : a)$, are multicast towards nodes 3 and 4. In the same way, the updates performed on S by some transaction T_S: insert $S(B : b)$, are multicast towards nodes 3 and 4. With the correctness criterion of proposition 3.1, there is no ordering among the commits of refresh transactions RT_r and RT_s associated with T_R and T_S. Therefore, it might happen that RT_r commits before RT_s at node 3 and in a reverse order at node 4. In which case, a simple query transaction Q that computes $(R - S)$ could return an empty result at node 4, which is impossible at node 3. The following criterion requires that RT_r and RT_s commit in the same order at nodes 3 and 4.

Definition 3.8 *(Total order). Let T_1 and T_2 be two committed update transactions. If two nodes commit both the associated refresh transactions RT_1 and RT_2, they both commit RT_1 and RT_2 in the same order.*

Proposition 3.2 *If a lazy master replicated system D that has a bowtie configuration but not a triangular configuration, uses a valid refreshment algorithm meeting the global FIFO order and the total order criteria, then this refreshment algorithm is correct.*

3.3 Master/Slave Induced Ordering

We first extend the model presented in Section 2 to deal with materialized views as follows. From now on, we shall consider that in a master/slave node MS having a materialized view, say $V(s_1)$, any refresh transaction of s_1 is understood to encapsulate the update of some virtual copy $\hat{V}$. The actual replica copies V and v are then handled as if they were secondary copies of $\hat{V}$. Hence, we consider that the update of the virtual copy $\hat{V}$ is associated with:

- at node MS, a refresh transaction of V, noted RT_V,
- at any node S having a secondary copy of v, a refresh transaction of v, noted RT_v.

With this new modeling in mind, consider the example of Figure 2(b). Let $V(A)$ be the materialized view defined from the secondary copy $s1$. Suppose that at the initial time t_o of the system, the instance of $V(A)$ is: $\{V(A : 8)\}$ and the instance of $S(B)$ is: $\{S(B : 9)\}$. Suppose that we have two update transactions T_S and $T_{\hat{V}}$, running at nodes 1 and 2 respectively: T_S: [delete $S(B : 9)$; insert $S(B : 6)$], and $T_{\hat{V}}$: [if exists $S(B : x)$ and $x \leq 7$ then delete $V(A : 8)$; insert $V(A : 5)$]. Finally, suppose that we have the query transaction Q over V and S, Q: [if exists $V(A : x)$ and $S(B : y)$ and $y < x$ then $bool = true$ else $bool = false$], where *bool* is a variable local to Q.

Now, a possible execution is the following. First, T_S commits at node 1 and its update is multicast towards nodes 2 and 3. Then, RT_{s_1} commits at node 2. At this point of time, say t_1, the instance of s_1 is $\{s_1(B : 6)\}$.

Then the update transaction $T_{\hat{V}}$ commits, afterwards the refresh transaction RT_V commits. The instance of V is $\{V(A:5)\}$. Then at node 3, RT_v commits (the instances of v and s_2 are $\{v(A:5)\}$ and $\{s_2(B:9)\}$), and finally, RT_{s_2} commits (the instances of v and s_2 are $\{v(A:5)\}$ and $\{s_2(B:6)\}$). A quiescent state is reached at this point of time, say t_2.

However, there exists an inconsistent observable state. Suppose that Q executes at time t_1 on node 2. Then, Q will return a value *true* for *bool*. However, for any time between t_0 and t_2, the execution of Q on node 3 will return a value *false* for *bool*, which contradicts our definition of consistency.

The following criterion imposes that the commit order of refresh transactions must reflect the commit order at the master/slave node.

Definition 3.9 *(Master/slave induced order). If MS is a node holding a secondary copy s_1 and a materialized view V, then any node N_i, $i > 1$, having secondary copies s_i and v_i must commit its refresh transactions RT_{s_i} and RT_{v_i} in the same order as RT_V and RT_{s_1} commit at MS.*

Proposition 3.3 *If a lazy master replicated system D that has a view triangular configuration but not a bowtie configuration, uses a valid refreshment algorithm meeting the global FIFO order and the master/slave induced order criteria then this refreshment algorithm is correct.*

As said before, a configuration can be both a bowtie and a view triangular configuration. In this case, the criteria for both configurations must be enforced.

Proposition 3.4 *If a lazy master replicated system D having both a view triangular configuration and a bowtie configuration, uses a valid refreshment algorithm meeting the global FIFO order, the master/slave induced order and the total order criteria, then this refreshment algorithm is correct.*

4 Refreshment Algorithm

We start this section by presenting the system architecture assumed by our algorithms. Then, we present our refreshment algorithm that uses a deferred update propagation strategy and prove its correctness. Finally we discuss the rationale for our algorithm.

4.1 System Architecture of Nodes

To maintain the autonomy of each node, we assume that four components are added to a regular database system, that includes a transaction manager and a query processor, in order to support a lazy master replication scheme. Figure 3 illustrates these components for a node having both primary and secondary copies. The first component, called *Replication Module*, is itself composed of three sub-components: a Log Monitor, a Propagator and a Receiver. The second component, called *Refresher*, implements a refreshment strategy. The third component, called *Deliverer*, manages the submission of refresh transactions to the local transaction manager. Finally, the last component, called *Network Interface*, is used to propagate and receive update messages (for simplicity, it is not portrayed on Figure 3). We now detail the functionality of these components.

We assume that the *Network Interface* provides a global FIFO reliable multicast [18] with a known upper bound [11]: messages multicast by a same node are received in the order they have been multicast. We also assume that each node has a local clock. For fairness reasons, clocks are assumed to have a bounded drift and to be ε synchronized. This means that the difference between any two correct clocks is not higher than the precision ε.

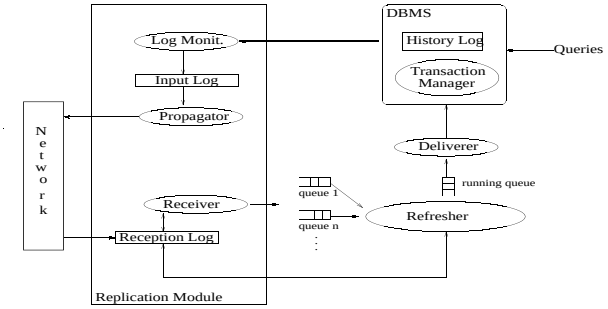

Figure 3: Architecture of a Node

The *Log Monitor* uses *log sniffing* [25, 20] to extract the changes to a primary copy by continuously reading the content of a local History Log (noted H). We safely assume (see Chap. 9 of [16]) that a log record contains all the information we need such as the timestamp of a committed update transaction, and other relevant attributes that will be presented in the next section. Each committed update transaction T has a timestamp (henceforth denoted C), which corresponds to the real time value at T's commitment time. When the log monitor finds a write operation on a primary copy, it reads the corresponding log record from H and writes it into a stable storage, called *Input Log*, that is used by the Propagator. We do not deal with conflicts between the write operations on the History Log and the read operations performed by the Log Monitor.

The *Receiver* implements update message reception. Messages coming from different masters or master/slaves are received and stored into a *Reception Log*. The receiver then reads messages from this log and stores them in FIFO *pending queues*. We denote Max, the upper bound of the time needed to multicast a message from a node and insert it into a pending queue at a receiving node. A node N has as many pending queues $q_1, ...q_n$ as masters or master/slaves nodes from which N has a secondary copy. The contents of these queues form the input to the Refresher.

The *Propagator* implements the propagation of update messages constructed from the Log Monitor. Such messages are first written into the *Input Log*. The propagator then continuously reads the *Input Log* and propagates messages through the network interface.

The *Refresher* implements the refreshment algorithm. First, it reads the contents of the pending queues, and based on its refreshment parameters, submits refresh transactions by inserting them into a *running queue*. The running queue contains all ordered refresh transactions not yet entirely executed.

Finally, the *Deliverer* submits refresh transactions to the local transaction manager. It reads the content of the running queue in a FIFO order and submits each write operation as part of a refresh transaction to the local transaction manager. The local transaction manager ensures serializability of local transactions. Moreover, it executes the operations requested by the refresh transactions according to the submission order given by the *Deliverer*.

4.2 Refreshment Algorithm

As described in Section 2, the refreshment algorithm has a triggering and an ordering parameters. In this section, we present the refreshment algorithm in the case of a *deferred-immediate* update propagation strategy (i.e., using an immediate triggering), and focus on the ordering parameter.

Deferred-Immediate Refresher
input: pending queues $q_1 ... q_n$
output: running queue
variables:
  $curr\_M$, new_M: messages from pending queues;
  timer: local reverse timer whose state is either active or inactive;
begin
  timer.state = inactive;
  $curr\_M = new\_M = \emptyset$;
  repeat
    on message arrival or change of timer's state to inactive do
      Step 1:
      $new\_M \leftarrow$ message with min C among top messages of q_1, q_n;
      Step 2:
      if $new\_M \neq curr\_M$ then
        $curr\_M \leftarrow new\_M$;
        calculate $deliver\_time(curr\_M)$;
        timer.value $\leftarrow deliver\_time(curr\_M) - local\_time$
        timer.state $\leftarrow$ active;
      endif
    on timer.value = 0 do
      Step 3:
      write $curr\_M$ into running queue;
      dequeue $curr\_M$ from its pending queue;
      timer.state $\leftarrow$ inactive;
  for ever
end

Figure 4: Deferred-Immediate Refreshment Algorithm

The principle of the refreshment algorithm is the following. A refresh transaction RT is committed at a slave or master/slave node (1) once all its write operations have been done, (2) according to the order given by the timestamp C of its associated update transaction, and (3) at the earliest, at real time $C + Max + \varepsilon$, which is called the deliver time, noted $deliver\_time$. Therefore, as clocks are assumed to be ε synchronized, the effects of updates on secondary copies follow the same chronological order in which their corresponding primary copies were updated.

We now detail the algorithm given in Figure 4. Each element of a pending queue is a message that contains: a sequence of write operations corresponding to a refresh transaction RT, and a timestamp C of the update transaction associated with RT. Since messages successively multicast by a same node are received in that order by the destination nodes, in any pending queue, messages are stored according to their multicast order (or commitment order of their associated update transactions).

Initially, all pending queues are empty, and $curr\_M$ and new_M are empty too. Upon arrival of a new message M into some pending queue signaled by an event, the Refresher assigns variable new_M with the message that has the smallest C among all messages in the top of all pending queues. If two messages have equal timestamps, one is selected according to the master or master/slave identification priorities. This corresponds to Step 1 of the algorithm. Then, the Refresher compares new_M with the currently hold message $curr\_M$. If the timestamp of new_M is smaller than the timestamp of $curr\_M$, then $curr\_M$ gets the value of new_M. Its deliver time is then calculated, and a local reverse timer is set with value $deliver\_time - local\_time$. This concludes Step 2 of the algorithm. Finally, whenever the timer expires its time, signaled by an event, the Refresher writes $curr\_M$ into the running queue and dequeues it from its pending queue. Each message of the running queue will yield a different refresh transaction. If an update message takes Max time to reach a pending queue, it can be processed immediately by the Refresher.

4.3 Refreshment Algorithm Correctness

We first show that the refreshment algorithm is valid for any acceptable configuration. A configuration is said *acceptable* iff (i) it is acyclic, and (ii) if it is a triangular configuration, then it is a view triangular configuration.

Lemma 4.1 *The Deferred-immediate refreshment algorithm is valid for any acceptable configuration.*

Lemma 4.2 *(Chronological order). The Deferred-immediate refreshment algorithm ensures for any acceptable configuration that, if T_1 and T_2 are any two update transactions committed respectively at global times t_1 and t_2 then :*

- *if $t_2 - t_1 > \varepsilon$, the timestamps C_2 for T_2 and C_1 for T_1 meet $C_2 > C_1$.*
- *any node that commits both associated refresh transactions RT_1 and RT_2, commits them in the order given by C_1 and C_2.*

Lemma 4.3 *The Deferred-immediate refreshment algorithm satisfies the global FIFO order criterium for any acceptable configuration.*

Lemma 4.4 *The Deferred-immediate refreshment algorithm satisfies the total order criteria for any acceptable configuration.*

Lemma 4.5 *The Deferred-immediate refreshment algorithm satisfies the master/slave induced order criteria for any acceptable configuration.*

From the previous lemmas and propositions, we have:

Theorem 4.1 *The Deferred-immediate refreshment algorithm is correct for any acceptable configuration.*

4.4 Discussion

A key aspect of our algorithm is to rely on the upper bound Max on the transmission time of a message by the global FIFO reliable multicast. Therefore, it is essential to have a value of Max that is not overestimated. The computation of Max resorts to scheduling theory (e.g., see [27]). It usually takes into account four kinds of parameters. First, there is the global reliable multicast algorithm itself (see for instance [18]). Second, are the characteristics of the messages to multicast (e.g. arrival laws, size). For instance, in [12], an estimation of Max is given for sporadic message arrivals. Third, are the failures to be tolerated by the multicast algorithm, and last are the services used by the multicast algorithm (e.g. medium access protocol). It is also possible to compute an upper bound Max_i for each type i of message to multicast. In that case, the refreshment algorithm at node N waits until $max_{i \in J}\ Max_i$ where J is the set of message types that can be received by node N.

Thus, an accurate estimation of Max depends on an accurate knowledge of the above parameters. However, accurate values of the application dependent parameters can be obtained in performance sensitive replicated database applications. For instance, in the case of data warehouse applications that have strong requirements on freshness, certain characteristics of messages can be derived from the characteristics of the operational data sources (usually, transaction processing systems). Furthermore, in a given application, the variations in the transactional workload of the data sources can often be predicted.

In summary, the approach taken by our refreshment algorithm to enforce a total order over an algorithm that implements a global FIFO reliable multicast trades the use of a worst case multicast time at the benefit of reducing the number of messages exchanged on the network. This is a well known tradeoff. This solution brings simplicity and ease of implementation.

5 Optimizations of the Refreshment

In this section, we present two main optimizations for the refreshment algorithm presented in Section 4. First, we show that for some configurations, the deliver time of a refresh transaction needs not to include the upper bound (Max) of the network and the clock precision (ε), thereby considerably reducing the waiting time of a refresh transaction at a slave or master/slave node. Second, we show that without sacrificing correctness, the principle of our refreshment algorithm can be combined with immediate update propagation strategies, as they were presented in [23]. Performance measurements, reported in Section 6, will demonstrate the value of this optimization.

5.1 Eliminating the Deliver Time

There are cases where the waiting time associated with the deliver time of a refresh transaction can be eliminated. For instance, consider a multinational investment bank that has traders in several cities, including New York, London, and Tokyo. These traders update a local database of positions (securities held and quantity), which is replicated using a lazy master scheme (each site is a master for securities of that site) into a central site that warehouses the common database for all traders. The common database is necessary in order for risk management software to put limits on what can be traded and to support an internal market. A trade will be the purchase of a basket of securities belonging to several sites. In this context, a delay in the arrival of a trade notification may expose the bank to excessive risk. Thus, the time needed to propagate updates from a local site to the common database must be very small (e.g., below a few seconds).

This scheme is a 1slave-per-master configuration, which only requires a global FIFO order to ensure the correctness of its refreshment algorithm (see proposition 3.1). Since, we assume a reliable FIFO multicast network, there is no need for a refresh transaction to wait at a slave node before being executed. More generally, given an arbitrary acceptable configuration, the following proposition characterizes those slave nodes that can process refresh transactions without waiting for their deliver time.

Proposition 5.1 *Let N a node of a lazy master replicated system D. If for any node N' of D, X being the set of common replicas between N and N', we have:*

- *$cardinal(X) \leq 1$, or*
- *$\forall X_1, X_2 \in X$, the primary copies of X_1 and X_2 are hold by the same node,*

then any valid refreshment algorithm meeting the global FIFO order criteria is correct for node N.

From an implementation point of view, the same refreshment algorithm runs at each node. The behavior of the refreshment algorithm regarding the need to

wait or not, is simply conditioned by a local variable. Thus, when the configuration changes, only the value of the variable of each node can possibly change.

5.2 Immediate Propagation

We assume that the Propagator and the Receiver both implement an immediate propagation strategy as specified in [23], and we focus here on the Refresher. Due to space limitations, we only present the *immediate-immediate* refreshment algorithm. We have chosen the *immediate-immediate* version because it is the one that provides the best performance compared with *deferred-immediate*, as indicated in [23].

```
Immediate_immediate Refresher
input: pending queues q1...qn
output: running queue
variables:
  curr_m, new_m: messages from pending queues;
  timer: local reverse timer whose state is either active or inactive;
begin
  timer.state = inactive;
  curr_m = new_m = ∅;
  repeat
    on message arrival or change of timer's state to inactive do
      if m ≠ commit then
        write m into the running queue;
        dequeue m from its pending queue;
      else
        new_m ← commit message with min C
          among top messages of q1...qn;
        if new_m ≠ curr_m then
          curr_m ← new_m;
          calculate deliver_time(curr_m);
          timer.value ← deliver_time(curr_m) − local_time
          timer.state ← active;
        endif
      endif
    on timer.value = 0 do
      write curr_m into running queue;
      dequeue curr_m from its pending queue;
      timer.state ← inactive;
  for ever
end
```

Figure 5: Immediate-Immediate Refreshment Algorithm

5.2.1 Immediate-Immediate Refreshment

We detail the algorithm of Figure 5. Unlike deferred-immediate refreshment, each element of a pending queue is a message m that carries an operation o of some refresh transaction, and a timestamp C. Initially, all pending queues are empty. Upon arrival of a new message m in some pending queue, signaled by an event, the Refresher reads the message and if m does not correspond to a *commit*, inserts it into the running queue. Thus, any operation carried by m other than commit can be immediately submitted for execution to the local transaction manager. If m contains a *commit* operation then new_m is assigned with the commit message that has the smallest C among all messages in the top of all pending queues. Then, new_m is compared with $curr\_m$. If new_m has a smallest timestamp than $curr\_m$, then $curr\_m$ is assigned with new_m. Afterwards, the Refresher calculates the *deliver_time* for $curr\_m$, and timer is set as in the *deferred-immediate* case. Finally, when the timer expires, the Refresher writes $curr\_m$ into the running queue, dequeues it from its pending queue, sets the timer to inactive and re-executes Step 1.

5.2.2 Algorithm Correctness

Like the deferred-immediate refreshment algorithm, the immediate-immediate algorithm enforces refresh transactions to commit in the order of their associated update transactions. Thus, the proofs of correctness for any acceptable configuration are the same for both refreshment algorithms.

6 Performance Evaluation

In this section, we summarize the main performance gains obtained by an *immediate-immediate* refreshment algorithm against a *deferred-immediate* one. More extensive performance results are reported in [23]. We use a simulation environment that reflects as much as possible a real replication context. We focus on a bowtie configuration which requires the use of a $Max + \varepsilon$ deliver time, as explained in Section 5.2. However, once we have fixed the time spent to reliably multicast a message, we can safely run our experiments with a single slave and several masters.

Our simulation environment is composed of *Master*, *Network*, *Slave* modules and a database server. The Master module implements all relevant capabilities of a master node such as log monitoring and message propagation. The Network module implements the most significant factors that may impact our update propagation strategies such as the delay to reliably multicast a message. The Slave module implements the most relevant components of the slave node architecture such as Receiver, Refresher and Deliverer. In addition, for performance evaluation purposes, we add the Query component in the slave module, which implements the execution of queries that read replicated data. Finally, a database server is used to implement refresh transactions and query execution.

Our environment is implemented on a Sun Solaris workstation using Java/JDBC. We use sockets for inter-process communication and Oracle 7.3 to implement refresh transaction and query processing. For simulation purposes, each write operation corresponds to an UPDATE command submitted to the server.

6.1 Performance Model

The metrics used to compare the two refreshment algorithms is given by the freshness of secondary copies. More formally, given a replica X, which is either a secondary or a primary copy, we define $n(X,t)$ as the number of committed update transactions on X at global time t. We assume that update transactions can have different sizes but their occurence is uniformly

distributed over time. Using this assumption, we define the degree freshness of a secondary copy r at global time t as: $f(r,t) = n(r,t)/n(R,t)$. Therefore, a degree of freshness close to 0 means bad data freshness while close to 1 means excellent.

The mean degree of freshness of r at a global time T is defined as: $mean_f = 1/T \int_0^T f(r,t)dt$.

We now present the main parameters for our experimentations summarized in Table 6.1. We assume that the mean time interval between update transactions, noted λ_t, as reflected by the history log of each master, is bursty. Updates are done on the same attribute of a different tuple. We focus on *dense* update transactions, i.e., transactions with a small time interval between each two writes. We define two types of update transactions. Small update transactions have size 5 (i.e., 5 write operations), while long transactions have size 50. We define four scenarios in which the proportion of long transactions, noted *ltr*, is set respectively to 0, 30, 60, and 100. Thus, in a scenario where $ltr = 30$, 30 % of the executed update transactions are long. Finally, we define an abort transaction ratio, noted *abr*, of 0, 5%, 10%, 20%, that corresponds to the percentage of transactions that abort in an experiment. Furthermore, we assume that a transaction abort always occurs after half of its execution.

Table 1: Performance Parameters

Param.	*Definition*	*Values*
λ_t	mean time between Trans.	bursty:200ms
λ_q	mean time between Queries	low:15s
nbmaster	Number of Master nodes	1 to 8
$\|Q\|$	Query Size	5
$\|RT\|$	Refresh Transaction Size	5; 50
ltr	Long Transaction Ratio	0; 30; 60; 100%
abr	Abort Ratio	0; 5; 10; 20%
t_{short}	Multicast Time per record	20ms; 100ms

Network delay is calculated by $\delta + t$, where δ is the waiting time in the input queue of the Network module, and t is the reliable multicast time of a message until its insertion in the pending queue of the Refresher. Concerning the value of t used in our experiments, we have a short message multicast time, noted t_{short}, which represents the time needed to reliably multicast a single log record. In addition, we consider that the time spent to reliably multicast a sequence of log records is linearly proportional to the number of log records it carries. The network overhead delay, δ, is implicity modeled by the system overhead to read from and write to sockets. The *Total propagation time* (noted t_p) is the time spent to reliably multicast all log records associated with a given transaction[4]. Finally, when $ltr > 0$, the value of Max is calculated using the maximun time spent to reliably multicast a long transaction ($50 * t_{short}$). On the other hand, when $ltr = 0$, the value of Max is calculated using the maximun time spent to reliably multicast a short transaction ($5 * t_{short}$). The refresh transaction execution time is influenced by the existence of possible conflicting queries that read secondary copies at the slave node. Therefore, we need to model queries. We assume that the mean time interval between queries is low, and the number of data items read is small (fixed to 5). We fix a 50% conflict rate for each secondary copy, which means that each refresh transaction updates 50% of the tuples of each secondary copy that are read by a query.

To measure the mean degree of freshness, we use the following variables. Each time an update transaction commits at a master, variable *version_master* for that master, is incremented. Similarly, each time a refresh transaction commits at the slave, variable *version_slave*, is incremented. Whenever a query conflicts with a refresh transaction we measure the degree of freshness.

6.2 Experiments

We present three experiments. The results are average values obtained from the execution of 40 update transactions. The first experiment shows the mean degree of freshness obtained for the *bursty* workload. The second experiment studies the impact on freshness when update transactions abort. In the third experiment, we study the effect of an increase in network delay.

We now summarize our major results. As depicted in Figure 6, when $ltr = 0$ (short transactions), the mean degree of freshness is already impacted because on average, $\lambda_t < t_p$[5]. With 2, 4, and 8 masters, the results of *immediate-immediate* are much better than those of *deferred-immediate*, as *ltr* increases. For instance, with 4 masters with $ltr = 30$, the mean degree of freshness is 0.62 for *immediate-immediate* and 0.32 for *deferred-immediate*. In fact, *immediate-immediate* always yields the best mean degree of freshness even with network contention due to the parallelism of log monitoring, propagation, and refreshment.

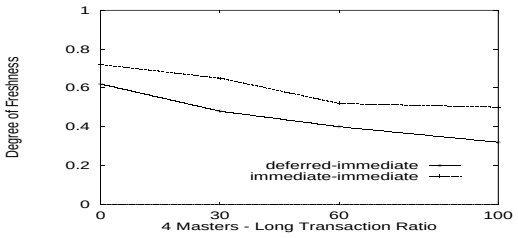

Figure 6: Bursty Workload

With *immediate-immediate*, the mean query response time may be seriously impacted because each time a query conflicts with a refresh transaction, it may be blocked during a long period of time since the propagation time may be added to the refresh transaction execution time. When $\lambda_q >> \lambda_t$, the probability

[4]If n represents the size of the transaction with immediate propagation, we have $t_p = n \times (\delta + t_{short})$, while with deferred propagation, we have $t_p = (\delta + n \times t_{short})$.

[5]Therefore, during T_i's update propagation, T_{i+1}...T_{i+n} may be committed.

of conflicts is quite high. The higher the network delay value, the higher are the query response times in conflict situations. However as also pointed out in [19], we verified that the use of a multiversion protocol on the slave node may significantly reduce query response times, without a significant decrease in the mean degree of freshness.

The *abort* of an update transaction with *immediate-immediate* does not impact the mean degree of freshness since the delay introduced to undo a refresh transaction is insignificant compared to the propagation time. As shown in Figure 7, for $ltr = 0$ and various values of *abr* (5,10,20), the decrease of freshness introduced by update transactions that abort with *immediate-immediate* is insignificant[6]. This behavior is the same for other values of *ltr* (30, 60, 100).

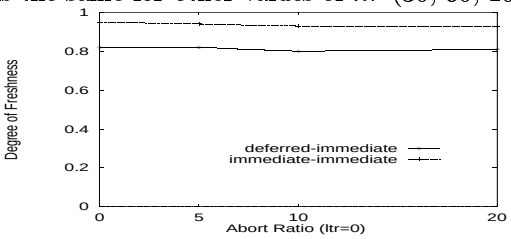

Figure 7: *Abort* Effects

Finally, the improvements brought by *immediate-immediate* are more significant when the network delay to propagate a single operation augments. Figure 8 and Figure 9 compares the freshness results obtained when $\delta = 100ms$ and $\delta = 20ms$. For instance, when $\delta = 20$ and $ltr = 100$ *immediate-immediate* improves 1.1 times better than *deferred-immediate* and when $\delta = 100$, *immediate-immediate* improves 5 times better. This clearly shows the advantages of having tasks being executed in parallel.

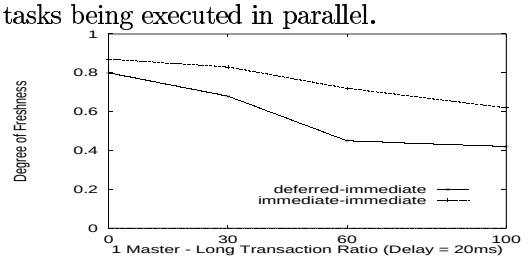

Figure 8: Network Delay =20ms

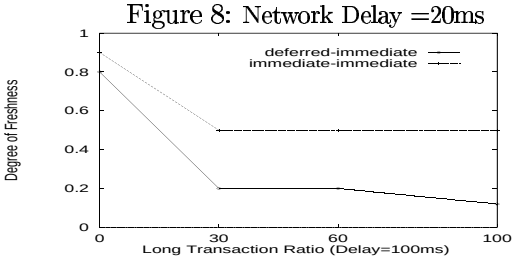

Figure 9: Network Delay =100ms

7 Related Work

Apart from the work cited in Section 1, the closest work to ours is in [6]. The authors show that for any strongly acyclic configuration a refreshment algorithm which enforces a global FIFO ordering, guarantees a global serializability property, which is similar to our notion of correction. Their result is analogous to our Proposition 3.1. They also propose an algorithm, which assigns, when it is possible, a site to each primary copy so that the resulting configuration is strongly acyclic. However, no algorithm is provided to refresh secondary copies in the cases of non strongly acyclic configurations.

Much work has been devoted to the maintenance of integrity constraints in federated or distributed databases, including the case of replicated databases [7, 13, 8, 17]. These papers propose algorithms and protocols to prevent the violation of certain kind of integrity constraints by local transactions. However, their techniques are not concerned with the consistent refreshment of replicas.

In [25], the authors describe a lazy group replication scheme in which the update propagation protocol applies updates to replicated data in their arrival order, possibly restoring inconsistencies when arrivals violate the timestamp ordering of transactions. The major difference with our work is that they achieve consistency by undoing and re-executing updates which are out-of-order, whereas we do not allow inconsistent database states.

The timestamp message delivery protocol in [15] implements eventual delivery for a lazy group replication scheme [10]. It uses periodic exchange of messages between pairs of servers that propagate messages to distinct groups of master nodes. At each master node incoming messages are stored in a history log (as initially proposed in [20]) and later delivered to the application in a defined order. Eventual delivery is not appropriate in our framework since we are interested in improving data freshness.

The goal of epidemic algorithms [28] is to ensure that all replicas of a single data item converge to the same value in a lazy group replication scheme. Updates are executed locally at any node. Later, nodes communicate to exchange up-to-date information. In our approach, updates are propagated from each primary copy towards its secondary copies.

Formal concepts for specifying coherency conditions in a replicated distributed database have been introduced in [14]. The authors focus on a *deferred-immediate* update propagation strategy and propose concepts for computing a measure of relaxation[7]. Their concept of *version* is closely related to our notion of freshness.

8 Conclusion

In a lazy master replicated system, a transaction can commit after updating one replica copy (primary copy) at some node. The updates are propagated towards

[6] In the worst case, it achieves 0.2.

[7] called *coherency index*.

the other replicas (secondary copies), and these replicas are refreshed in separate refresh transactions.

We proposed refreshment algorithms which address the central problem of maintaining replicas' consistency. An observer of a set of replicas at some node never observes a state which is never seen by another oberver of the same set of replicas at another node.

This paper has three major contributions. Our first contribution is a formal definition of (i) the notion of correct refreshment algorithm and (ii) correctness criteria for any acceptable configuration.

Our second contribution is an algorithm meeting these correctness criteria for any acceptable configuration. This algorithm can be easily implemented over an existing database system. It is based on a deferred update propagation, and it delays the execution of a refresh transaction until its deliver time.

Our third contribution concerns optimizations of the refreshment algorithm in order to improve the data freshness. With the first optimization, we characterized the nodes that do not need to wait. The second optimization uses *immediate-immediate* update propagation strategy. This strategy allows parallelism between the propagation of updates and the execution of the associated refresh transactions.

Finally, our performance evaluation shows that the *immediate-immediate* strategy always yields the best mean degree of freshness for a bursty workload.

References

[1] G. Alonso and A. Abbadi, *Partitioned Data Objects in Distriduted Databases*,Distributed and Parallel Databases, 3(1):5-35, 1995.

[2] D. Agrawal and G. Alonso and A. El Abbadi and I. Stanoi,*Exploiting Atomic Broadcast in Replicated Databases*, EURO-PAR Int. Conf. on Parallel Processing, August 1997.

[3] R. Alonso and D. Barbara and H. Garcia-Molina, *Data Caching Issues in an Information Retrieval System*, ACM Transactions on Database Systems, 15(3):359-384, September 1990.

[4] P.A. Bernstein and E. Newcomer, *Transaction Processing*, Morgan Kaufmann, 1997.

[5] S. Chaudhuri and U. Dayal, *An Overview of Data Warehousing and OLAP Technology*, ACM SIGMOD Record, 26(1), March 1997.

[6] P. Chundi and D. J. Rosenkrantz and S. S. Ravi, *Deferred Updates and Data Placement in Distributed Databases*, Int. Conf. on Data Engineering (ICDE), Louisiana, February 1996.

[7] S. Ceri and J. Widom, *Managing Semantic Heterogeneity with Production Rules and Persistent Queues*, Int. Conf. on VLDB, 1993.

[8] L. Do and P. Drew, *The Mangement of Interdependent Asynchronous Transactions in Heterogeneous Database Environments*, Int. Conf. on Database Systems for Advanced Applications, 1995.

[9] A. Downing and I. Greenberg and J. Peha, *OSCAR: A System for Weak-Consistency Replication*, Int. Workshop on Management of Replicated Data, pages26-30, Houston, November 1990.

[10] J. Gray and P. Helland and P. O'Neil and D. Shasha, *The Danger of Replication and a Solution*, ACM SIGMOD Int. Conf. on Management of Data, Montreal, June 1996.

[11] L. George and P. Minet, *A FIFO Worst Case Analysis for a Hard Real-Time Distributed Problem with Consistency Constraints*, Int. Conf. on Distributed Computing Systems (ICDCS97), Baltimore, May 1997.

[12] L. George and P. Minet, *A Uniform Reliable Multicast Protocol with Guaranteed Responses Times*, ACM SIGPLAN workshop on Languages Compilers and Tools for Embedded Systems, Montreal, June 1998.

[13] H. Gupta and I.S. Mumick and V.S. Subrahmanian, *Maintaining Views Incremently*, ACM SIGMOD Int. Conf. on Management of Data, Washington, DC, May 1993.

[14] R. Gallersdorfer and M. Nicola, *Improving Performance in Replicated Databases through Relaxed Coherency*, Int. Conf. on VLDB, Zurich, September 1995.

[15] R. Goldring, *Weak-Consistency Group Communication and Membership*, PhD Thesis, Univ. of Santa Cruz, 1992.

[16] J.N. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan-Kaufmann, 1993.

[17] P. Grefen and J. Widom, *Protocols for Integrity Constraint Checking in Federated Databases*, Distributed and Parallel Databases, 54,October 1997.

[18] V. Hadzilacos and S. Toueg, *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*, Technical Report TR 94-1425, Cornell Univ., Ithaca, NY 14853, 1994.

[19] B. Kemme and G. Alonso, *A Suite of Database Replication Protocols based on Group Communication Primitives*, Int. Conf. on Distributed Computing Systems (ICDCS98), Amsterdam, May 1998.

[20] B. Kahler and O. Risnes, *Extending Logging for Database Snapshot Refresh*, Int. Conf. on VLDB, Brighton, September 1987.

[21] T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*, 2nd Edition, Prentice Hall, 1999.

[22] E. Pacitti and P. Minet and E. Simon, *Fast Scheduling Algorithms for Maintaining Replica Consistency in Lazy Master Configurations*, Long Paper, http://www-rodin.inria.fr/~pacitti/public_html/v99l.ps, February 1999.

[23] E. Pacitti and E. Simon and R. de Melo, *Improving Data Freshness in Lazy Master Schemes*, Int. Conf. on Distributed Computing Systems (ICDCS98), Amsterdam, May 1998.

[24] I. Stanoi and D. Agrawal and A. El Abbadi, *Using Broadcast Primitives in Replicated Databases*, Int. Conf. on Distributed Computing Systems (ICDCS98), Amsterdam, May 1998.

[25] S. K. Sarin and C. W. Kaufman and J. E. Somers, *Using History Information to Process Delayed Database Updates*, Int. Conf. on VLDB, Kyoto, August 1986.

[26] A. Sheth and M. Rusinkiewicz, *Management of Interdependent Data: Specifying Dependency and Consistency Requirements*, Int. Workshop on Management of Replicated Data, Houston, November 1990.

[27] K. Tindell and J. Clark, *Holistic Schedulability Analysis for Distributed Hard Real-Time Systems*, Microprocessors and Microprogramming, 40, 1994.

[28] D. B. Terry and M. M. Theimer and K. Petersen and A. J. Demers and M. J. Spreitzer and C. H. Hauser, *Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System*, Symposium on Operating System Principles (SIGOPS), Colorado, December 1995.

[29] Y. Zhuge and H. Garcia-Molina and J. L. Wiener, *View Maintenance in a Warehouse Environment*, ACM SIGMOD Int. Conf. on Management of Data, 1995.