Distributed and Parallel Databases, 18, 223–251, 2005

DOI: 10.1007/s10619-005-4257-4

Preventive Replication in a Database Cluster*

ESTHER PACITTI pacitti@lina.univ-nantes.fr
CÉDRIC COULON coulon@lina.univ-nantes.fr
PATRICK VALDURIEZ patrick.valduriez@inria.fr
INRIA and LINA, University of Nantes, France

M. TAMER ÖZSU tozsu@uwaterloo.ca
University of Waterloo, Canada

Abstract. In a database cluster, preventive replication can provide strong consistency without the limitations of synchronous replication. In this paper, we present a full solution for preventive replication that supports multi-master and partial configurations, where databases are partially replicated at different nodes. To increase transaction throughput, we propose an optimization that eliminates delay at the expense of a few transaction aborts and we introduce concurrent replica refreshment. We describe large-scale experimentation of our algorithm based on our RepDB∗ prototype (http://www.sciences.univ-nantes.fr./lina/ATLAS/RepDB) over a cluster of 64 nodes running the PostgreSQL DBMS. Our experimental results using the TPC-C Benchmark show that the proposed approach yields excellent scale-up and speed-up.

Keywords: database cluster, partial replication, preventive replication, strong consistency, TPC-C benchmarking

1. Introduction

High-performance and high-availability of database management have been traditionally achieved with parallel database systems [23], implemented on tightly-coupled multiprocessors. Parallel data processing is then obtained by partitioning and replicating the data across the multiprocessor nodes in order to divide processing. Although quite effective, this solution requires the database system to have full control over the data and is expensive in terms of software and hardware.

Clusters of PC servers now provide a cost-effective alternative to tightly-coupled multiprocessors. They have been used successfully by, for example, Web search engines using high-volume server farms (e.g., Google). However, search engines are typically read-intensive, which makes it easier to exploit parallelism. Cluster systems can make new businesses such as Application Service Providers (ASP) economically viable. In the ASP model, customers' applications and databases (including data and DBMS) are hosted at

*Work partially funded by the MDP2P project of the ACI "Masses de Donniées" of the French Ministry of Research.

the provider site and need be available, typically through the Internet, as efficiently as if they were local to the customer site. Notice that due to autonomy, it is possible that the DBMS at each node are heterogeneous. To improve performance, applications and data can be replicated at different nodes so that users can be served by any of the nodes depending on the current load [1]. This arrangement also provides high-availability since, in the event of a node failure, other nodes can still do the work. However, managing data replication in the ASP context is far more difficult than in Web search engines since applications can be update-intensive and both applications and databases must remain autonomous. The solution of using a parallel DBMS is not appropriate as it is expensive, requires heavy migration to the parallel DBMS and hurts database autonomy.

In this paper, we consider a database cluster with similar nodes, each having one or more processors, main memory (RAM) and disk. Similar to multiprocessors, various cluster system architectures are possible: shared-disk, shared-cache and shared-nothing [23]. Shared-disk and shared-cache require a special interconnect that provide a shared space to all nodes with provision for cache coherence using either hardware or software. Shared-nothing (or distributed memory) is the only architecture that supports our autonomy requirements without the additional cost of a special interconnect. Furthermore, shared-nothing can scale up to very large configurations. Thus, we strive to exploit a shared-nothing architecture.

The major problem of data replication is to manage the consistency of the replicas in the presence of updates [6]. The basic solution in distributed systems that enforces strong replica consistency[1] is synchronous (or eager) replication (typically using the Read-One-Write All—ROWA protocol [11]). Whenever a transaction updates a replica, all other replicas are updated inside the same distributed transaction. Therefore, the mutual consistency of the replicas is enforced. However, synchronous replication is not appropriate for a database cluster for two main reasons. First, all the nodes would have to homogeneously implement the ROWA protocol inside their local transaction manager, thus violating DBMS autonomy. Second, the atomic commitment of the distributed transaction should rely on the two-phase commit (2PC) protocol [11] which is known to be blocking (i.e. does not deal well with nodes' failures) and has poor scale up.

A better solution that scales up is lazy replication [14], where a transaction can commit after updating a replica, called *primary copy*, at some node, called *master node*. After the transaction commits, the other replicas, called *secondary copies*, are updated in separate refresh transactions at *slave nodes*. Lazy replication allows for different replication configurations [12]. A useful configuration is *lazy master* where there is only one primary copy. Although it relaxes the property of mutual consistency, strong consistency is assured. However, it hurts availability since the failure of the master node prevents the replica to be updated. A more general configuration is (*lazy*) *multi-master* where the same primary copy, called a multi-owner copy, may be stored at and updated by different master nodes, called multi-owner nodes. The advantage of multi-master is high-availability and high-performance since replicas can be updated in parallel at different nodes. However, conflicting updates of the same primary copy at different nodes can introduce replica incoherence.

Preventive replication [13] is an asynchronous solution that enforces strong consistency. Instead of using atomic broadcast, as in synchronous group-based replication [9], preventive replication uses First-In First-Out (FIFO) reliable multicast which is a weaker constraint.

It works as follows. Each incoming transaction is submitted, via a load balancer, to the best node of the cluster. Each transaction T is associated with a chronological timestamp value C, and is multicast to all other nodes where there is a replica. At each node, a *delay time d* is introduced before starting the execution of T. This delay corresponds to the upper bound of the time needed to multicast a message. When the delay expires, all transactions that may have committed before C are guaranteed to be received and executed before T, following the timestamp chronological order (i.e. total order). Hence, this approach prevents conflicts and enforces consistency. Its implementation over a cluster of 8 nodes showed good performance [13].

However, the original proposal has two main limitations. First, it assumes that databases are fully replicated across all cluster nodes and thus propagates each transaction to each cluster node. This makes it unsuitable for supporting large databases and heavy workloads on large cluster configurations. Second, it has performance limitations since transactions are performed one after the other, and must endure waiting delays before starting. Thus, refreshment is a potential bottleneck, in particular, in the case of bursty workloads where the arrival rates of transactions are high at times. This paper addresses these important limitations. It is based on the solution initially proposed in [4] with significant extensions regarding replication configurations, concurrency management, proofs of algorithms and performance evaluation.

In this paper, we provide support for partial replication, where databases are partially replicated at different nodes. Unlike full replication, partial replication can increase access locality and reduce the number of messages for propagating updates to replicas. To increase transaction throughput, we propose a refreshment algorithm that potentially eliminates the delay time, and we introduce concurrent replica refreshment. We describe the implementation of our algorithm in our RepDB∗ prototype [19] over a cluster of 64 nodes running the PostgreSQL DBMS. Our experimental results using the TPC-C Benchmark show that it yields excellent scale-up and speed-up.

The rest of the paper is organized as follows. Section 2 introduces the global architecture for processing user requests against applications into the cluster system. Section 3 defines the basic concepts for fully and partial replication. Section 4 describes preventive refreshment for partially replication, including the algorithm and architecture. Section 5 proposes some important optimizations to the refreshment algorithm that improves transaction throughput. Section 6 describes our validation and experimental results. Section 7 discusses related work. Section 8 concludes.

2. Database cluster architecture

In this section, we introduce the architecture for processing user requests against applications into the cluster system and discuss our general solutions for placing applications, submitting transactions and managing replicas. Therefore, the replication layer is identified together with all other general components.

In this paper, we exploit a shared-nothing architecture. This is the only architecture that allows sufficient node autonomy without the additional cost of special interconnects. In our shared-nothing architecture, each cluster node is composed of five layers (see

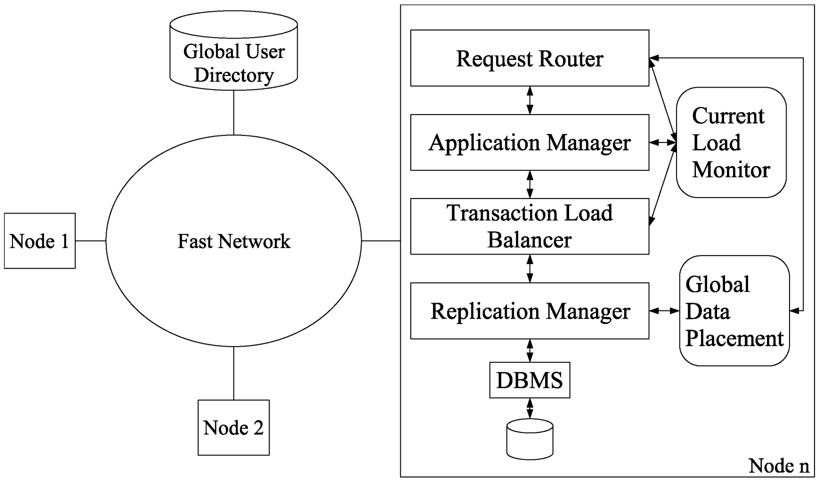


Figure 1. Peer-to-peer cluster architecture.

figure 1): Request Router, Application Manager, Transaction Load Balancer and Replication Manager. A user request may be a query or update transaction on a specific application. The general processing of a user request is as follows.

When a user request arrives at the cluster, traditionally through an access node, it is sent randomly to a cluster node *i*. There is no significant data processing at the access node, avoiding bottlenecks. Within that cluster node, the user is authenticated and authorized through the Request Router, available at each node, using a multi-threaded *global user directory* service. Notice that user requests are managed completely asynchronously. Next, if a request is accepted, then the Request Router chooses a node *j*, to submit the request. The choice of node *j* involves selecting all nodes in which the required application is available, and, among these nodes, the node with the lightest load. Therefore, eventually *i* may be equal to *j*. The *Request Router* then routes the user request to an application node using a traditional load balancing algorithm.

Notice, however, that the database accessed by the user request may be placed at another node *k* since applications and databases are both replicated and not every node hosts a database system. In this case, the choice regarding node *k* will depend on the cluster configuration and the database load at each node.

A node load is computed by a *current load monitor* available at each node. For each node, the load monitor periodically computes application and transaction loads using traditional load balancing strategies. For each type of load, it establishes a load grade and multicasts the grades to all the other nodes. A high grade corresponds to a high load. Therefore, the Request Router chooses the best node for a specific request using the node grades (light node is better as discussed below).

The *Application Manager* is the layer that manages application instantiation and execution using an application server provider. Within an application, each time a transaction is to be executed, the *Transaction Load Balancer* layer is invoked which triggers transaction execution at the best node, using the load grades available at each node. The “best” node is defined as the one with lighter transaction load. The Transaction Load Balancer ensures that each transaction execution obeys the ACID (atomicity, consistency, isolation, durability) properties [14], and then signals to the Application Manager to commit or abort the transaction.

The *Replication Manager layer* manages access to replicated data and assures strong consistency in such a way that transactions that update replicated data are executed in the same serial order at each node. We employ data replication because it provides database access parallelism for applications. Our preventive replication approach avoids conflicts at the expense of a forced waiting time for transactions, which is negligible due to the fast cluster network system.

3. Replication model

In this section, we define all the terms and concepts of lazy replication for fully and partially replicated databases necessary to understand our solutions. Then, we present the consistency criteria for the three types of configurations: Lazy-Master, Multi-master and Partially replicated.

3.1. Configurations

We assume that a replica is an entire relational table. Given a table R, we may have three kinds of copies: primary, secondary and multi-master. A *primary copy*, denoted by R, is stored at a *master* node where it can be updated while a *secondary copy*, denoted by r_i, is stored at one or more *slave* nodes i in read-only mode. A *multi-master copy*, denoted by R_i, is a primary copy that may be stored at several multi-master nodes i. Figure 2 shows various replication configurations, using two tables R and S.

Figure 2(a) shows a bowtie (lazy master) configuration where there are only primary copies and secondary copies. This configuration is useful to speed-up the response times of read-only queries through the slave nodes, which do not manage the update transaction load. However, availability is limited since, in the case of a master node failure, its primary copies can no longer be updated.

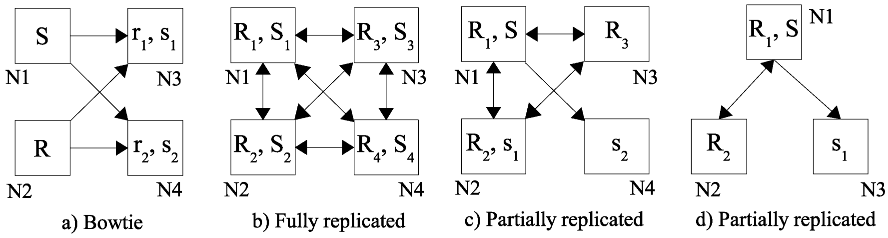


Figure 2. Replication configurations.

Figure 2(b) shows a fully replicated configuration. In this configuration, all nodes manage the update transaction load because whenever R or S is updated at one node, all other copies need be updated asynchronously at the other nodes. Thus, only the read-only query loads are different at each node. Since all the nodes perform all the transactions, load balancing is easy because all the nodes have the same load (when the specification of the nodes is homogeneous) and availability is high because any node can replace any other node in case of failure.

Figures 2(c) and (d) illustrate partially replicated configurations where all kinds of copies may be stored at any node. For instance, in figure 2(c), node N_1 carries the multi-master copy R_1 and the primary copy S, node N_2 carries the multi-master copy R_2 and the secondary copy s_1, node N_3 carries the multi-master copy R_3, and node N_4 carries the secondary copy s_2. Compared with full replication, only some of the nodes are affected by the updates on a multi-master copy (only those that hold common multi-master copies). Therefore, transactions do not have to be multicast to all the nodes. Thus, the nodes and the network are less loaded and the overhead for refreshing replicas is significantly reduced.

With partial replication a transaction T may be composed of a sequence of read and write operations followed by a commit (as produced by the SQL statement in figure 3) that updates multi-master copies. This is more general than in [13] where only write operations are considered. We define a *refresh transaction* as the sequence of write operations of a transaction, as written in the Log History. In addition, a *refreshment algorithm* is the algorithm that manages, asynchronously, the updates on a set of multi-master and secondary copies once one of the multi-master (or primary) copies is updated by T for a given configuration.

Given a transaction T received in the database cluster, there is an *origin node* chosen by the load balancer that triggers refreshment, and a set of *target nodes* that carries replicas involved with T. For simplicity, the origin node is also considered a target node. For instance, in figure 2(b) whenever node N_1 receives a transaction that updates R_1, then N_1 is the origin node and N_1, N_2, N_3 and N_4 are the target nodes. In figure 2(c), whenever N_3 receives a transaction that updates R_3, then the origin node is N_3 and the target nodes are N_1, N_2 and N_3.

To refresh multi-master copies in the case of full replication, it is sufficient to multicast the incoming transactions to all target nodes. But in the case of partial replication, even if a transaction is multicast towards all nodes, it may happen that the nodes are not be able to execute it because they do not hold all the replicas necessary to execute T locally. For instance, figure 2(c) allows an incoming transaction at node N_1, such as the one in figure 3 to read s_1 in order to update R_1. This transaction can be entirely executed at N_1 (to update R_1) and N_2 (to update R_2). However it cannot be executed at node N_3 (to update R_3)

```
UPDATE R1 SET att1=value
        WHERE att2 IN
        (SELECT att3 FROM S)
COMMIT;
```

Figure 3. Incoming transaction at node N1.

because N_3 does not hold a copy of S. Thus, refreshing multi-master copies in the case of partial replication needs to take into account replica placement.

3.2. Consistency criteria

Informally a correct refreshment algorithm guarantees that any two nodes holding a common set of replicas, $R_1, R_2, \dots, R_n$, must always produce the same sequence of updates on $R_1, R_2, \dots, R_n$. For each configuration and its sub-configurations, we provide a criterion that must be satisfied by the refreshment algorithm in order to be correct. Group communication systems provide multicast services that differ in the final order in which messages are delivered at each node. We use these known orders [14] as a guide to express our correctness criteria. An example of each configuration is presented in Section 3.1.

Lazy-Master configuration (figure 2(a)). In Lazy-Master configurations, inconsistency may arise if slave nodes can commit their refresh transactions in an order different than their corresponding master nodes. The following correctness criterion prevents this situation.

Definition 3.1 (Total order). Two refresh transactions RT_1 and RT_2 are said to be in *total order* if any slave node that commits RT_1 and RT_2, commits them in the same order.

Proposition 3.1. *For any cluster configuration C that meets a lazy-master configuration requirement, the refresh algorithm that C uses is correct if and only if the algorithm enforces total order.*

Multi-Master configuration (Figure 2(d)). In Multi-Master configurations, inconsistencies may arise whenever the serial execution orders of two transactions at two nodes are not equal. Therefore, transactions must be executed in the same serial order at any node. Thus, Global FIFO Ordering is not sufficient to guarantee the correctness of the refreshment algorithm. Hence the following correctness criterion is necessary:

Definition 3.2 (Total order). Two transactions T_1 and T_2 are said to be executed in *Total Order* if all multi-owner nodes that commit both T_1 and T_2 commit them in the same order.

Proposition 3.2. *For any cluster configuration C that meets a multi-master configuration requirement, the refresh algorithm that C uses is correct if and only if the algorithm enforces total order.*

Partially-Replicated configurations (Figures 2(c) and (d)). In a Partially-Replicated configuration, the inconsistency issues are similar to those found in each component sub-configuration, namely multi-master and lazy-master. That is, two transactions T_1 and T_2 must be executed in the same order at the multi-owner nodes, and, in addition, their corresponding refresh transactions RT_1 and RT_2 must commit in the same order in which the origin node commit T_1 and T_2. Therefore, the following correctness criterion prevents inconsistencies:

Proposition 3.3. *If a cluster configuration C meets partially replicated configuration requirement, then the refresh algorithm that C uses is correct if and only if for each sub-configuration SC correctness is enforced (see Propositions 3.1 and 3.2).*

Proposition 3.4. *For any cluster configuration C that meets the partially replicated requirements, the refresh algorithm that C uses is correct if and only if the algorithm enforces total order.*

4. Preventive refreshment

In this section, we first present the basic refreshment algorithm originally designed for full replication. Then we present the extension of the algorithm to manage partial replication. Afterwards we show the correctness of the algorithm for both fully and partially replicated configurations. Finally, we describe the Replication Manager architecture that implements these algorithms.

4.1. Full replication

We assume that the network interface provides global FIFO reliable multicast: messages multicast by one node are received at the multicast group nodes in the order they have been sent [7]. We denote by *Max*, the upper bound of the time needed to multicast a message from a node i to any other node j. It is essential to have a value of *Max* that is not over estimated. The computation of *Max* resorts to scheduling theory [22] and takes into account several parameters such as the global reliable network itself, the characteristics of the messages to multicast and the failures to be tolerated. We also assume that each node has a local clock. For fairness, clocks are assumed to have a drift and to be ε-synchronized. This means that the difference between any two correct clocks is not higher that ε (known as the precision).

To define the refreshment algorithm, we need the formal correctness criterion presented in Section 3.2 to define strong copy consistency. Inconsistencies may arise whenever the serial orders of two transactions at two nodes are not equal. Therefore, they must be executed in the same serial order at any two nodes. Thus, global FIFO ordering is not sufficient to guarantee the correctness of the refreshment algorithm.

Each transaction is associated with a chronological timestamp value C. The principle of the preventive refreshment algorithm is to submit a sequence of transactions in the same chronological order at each node. Before submitting a transaction at node i, we must check whether there is any older transaction en route to node i. To accomplish this, the submission time of a new transaction at node i is delayed by $Max+\varepsilon$. Thus the earliest time a transaction is submitted is $C + Max + \varepsilon$ (henceforth *delivery time*).

Whenever a transaction T_i is to be triggered at some node i, node i multicasts T_i to all nodes $1, 2, \ldots, n$, including itself. Once T_i is received at some other node j (i may be equal to j), it is placed in the pending queue in FIFO order with respect to the triggering node i. Therefore, at each multi-master node i, there is a set of queues, $q_1, q_2, \ldots, q_n$, called pending queues, each of which corresponds to a multi-master node and is used by the

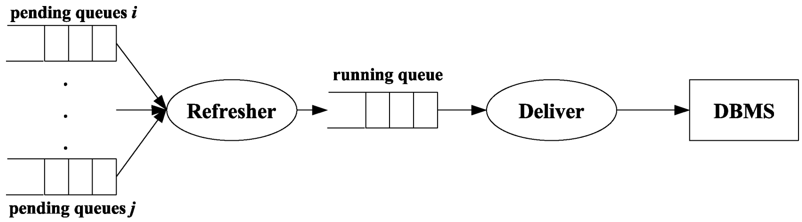


Figure 4. Refreshment architecture.

refreshment algorithm to perform chronological ordering with respect to the delivery times. Figure 4 shows part of the components necessary to run our algorithm. The *Refresher* reads transactions from the top of *pending queues* and performs chronological ordering with respect to the delivery times. Once a transaction is ordered, then the refresher writes it to the *running que*ue in FIFO order, one after the other. Finally *Deliver* keeps checking top of the running queue to start transaction execution, one after the other, in the local *DBMS*.

Let us illustrate the algorithm by an example. Suppose we have two nodes i and j, masters of the copy R. So at node i, there are two pending queues: $q(i)$ and $q(j)$ corresponding to multi-master nodes i and j. T_1 and T_2 are two transactions which update R, respectively on node i and on node j. Let us suppose that *Max* is equal to 10 and ε is equal to 1. So, on node i, we have the following sequence of execution:

– At time 10: T_2 arrives at node i with a timestamp $C_2 = 5$

- $q(i) = [T_2\ (5)]$, $q(j) = []$
- T_2 is chosen by the Refresher to be the next transaction to perform at *delivery_time* 16 $(5 + 10 + 1)$, and the time is set to expire at time 16.

– At time 12: T_1 arrives from node j with a timestamp $C_1 = 3$

- $q(i) = [T_2\ (5)]$, $q(j) = [T_1\ (3)]$
- T_1 is chosen by the Refresher to be the next transaction to perform at *delivery_time* 14 $(3 + 10 + 1)$, and the time is re-set to expire at time 14.

– At time 14: the timeout expires and the Refresher writes T_1 into the running queue.

- $q(i) = [T_2\ (5)]$, $q(j) = []$
- T_2 is selected to be the next transaction to perform at *delivery_time* 16 $(5 + 10 + 1)$

– At time 16: the timeout expires. The Refresher writes T_2 into the running queue.

- $q(i) = []$, $q(j) = []$

```
Multi-Master Refresher

Input:
 pending queues q1,…qn
Output:
 running queue
Variables:
 curr_T: currently selected transaction to be executed;
 first_T: transaction with the lowest timestamp in the
          pending queue
 timer: local reverse timer whose state is either active or
        inactive

Begin
 timer.state = inactive;
 curr_T = first_T = 0;
 Repeat
  On arrival of a new message
  or when (timer.state = active and timer.value = 0) do

  Step1:
   first_T <- message with min C among top messages
              of the pending queues;

  Step2:
   If first_T <> curr_T then
    curr_M <- first_T;
    calculate delivery_time(curr_T);
    timer.value <- delivery_time(curr_T) - local_time
    timer.state <- active;
   end if

  Step 3:
   If timer.state = active and timer.value = 0 then
    append curr_T to the running queue;
    dequeue curr_T from its pending queue;
    timer.state <- inactive;
   end if
 for ever
end
```

Figure 5. Multi-master refresher algorithm.

Although the transactions are received in wrong order with respect to their timestamps (T_2 then T_1) they are written into the running queue in chronological order according to their timestamps (T_1 then T_2). Thus, the total order is enforced even if messages are not sent in total order.

In figure 5, we can see the three steps of the algorithm used in the Refresher module. In step 1, at the reception of a new message in a pending queue, we choose the most recent message from the pending queues. In step 2, we calculate the *delivery_time* according to the timestamp of the message and the $Max + \varepsilon$, and then we set a local reverse timer that will expire at the *delivery_time*. Finally, in step 3, when the timer is over, the message is submitted to the running queue for execution.

4.2. Partial replication

With partial replication, some of the target nodes may not be able to perform a transaction *T* because they do not hold all the copies necessary to perform the read set of *T* (recall the discussion on figure 3). However the write sequence of *T*, which corresponds to its refresh transaction, denoted by *RT*, must be ordered using *T*'s timestamp value in order to ensure consistency. So *T* is scheduled as usual but not submitted for execution. Instead, the involved target nodes wait for the reception of the corresponding *RT*. Then, at origin node *i*, when the commitment of *T* is detected (by sniffing the DBMS' log—see Section 4.3), the corresponding *RT* is produced and node *i* multicasts *RT* towards the target nodes. Upon reception of *RT* at a target node *j*, the content of *T* (still waiting) is replaced with the content of incoming *RT* and *T* can be executed.

Let us now illustrate the algorithm with an example of execution. In figure 6, we assume a simple configuration with 4 nodes (N_1, N_2, N_3 and N_4) and 2 copies (*R* and *S*). N_1 carries a multi-owner copy of *R* and a primary copy of *S*, N_2 a multi-owner copy of *R*, N_3 a secondary copy of *S*, and N_4 carries a multi-owner copy of *R* and a secondary copy of *S*.

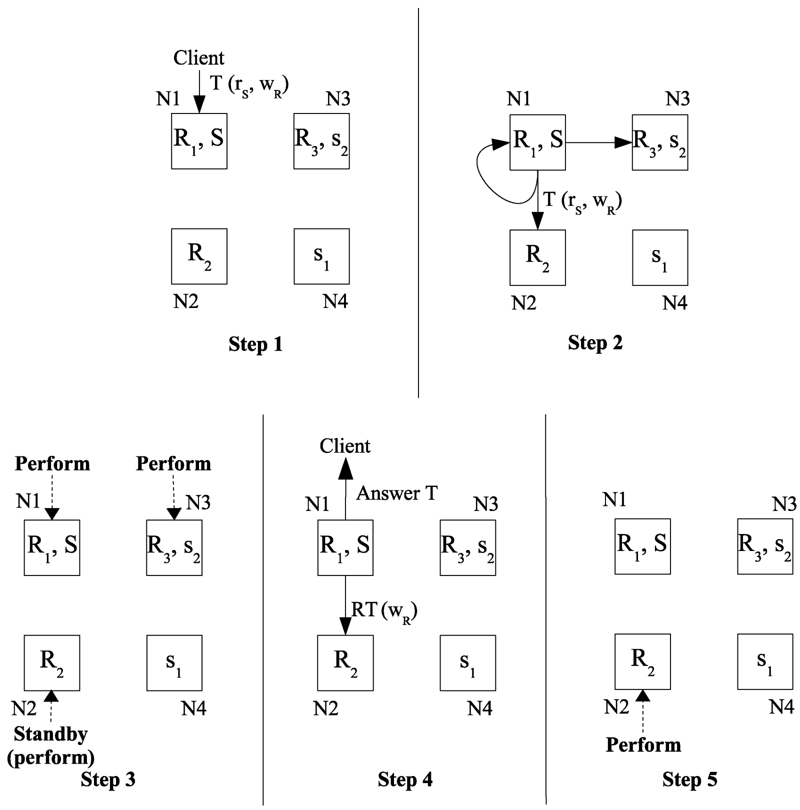


Figure 6. Example of preventive refreshment with partial configurations.

The refreshment proceeds in 5 steps. In step 1, N_1 (the origin node) receives T from a client which reads S and updates R_1. For instance, T can be the resulting read and write sequence produced by the transaction of figure 3. Then, in step 2, N_1 multicasts T to the involved target nodes, i.e. N_1, N_2 and N_4. N_3 is not concerned with T because it only holds a secondary copy s. In step 3, T can be performed using the refreshment algorithm at N_1 and N_4. At N_2, T is also managed by the Refresher and then put in the running queue. However, T cannot yet be executed at this target node because N_2 does not hold S. Thus, the Deliver needs to wait for its corresponding *RT* in order to apply the update on R (see step 4). In step 4, after the commitment of T at the origin node, the *RT* is produced and multicasts it to all involved target nodes. In step 5, N_2 receives *RT* and the Receiver replaces the content of T by the content of *RT*. The Deliver can then submit *RT*.

Partial replication may be blocking in case of failures. After the reception of T, some target nodes would be waiting for *RT*. Thus, if the origin node fails, the target nodes are blocked. However, this drawback can be easily solved by replacing the origin node by an equivalent node, a node that holds all the replicas necessary to execute T. Once the target nodes detect the failure of the origin node, it can request an equivalent node j to multicast *RT* given T's identifier. At node j, *RT* was already produced in the same way that at the origin node: transaction T is executed and, upon detection of T's commitment, an *RT* is produced and stored in a *RT* log (see Section 4.4), necessary to handle failure of the origin node. In the worst case where no other node holds all the replicas necessary to execute T, T is globally aborted. Reconsider the example in figure 6: if N_1 fails at Step 3, N_2 can not receive the *RT* corresponding to the waiting T. So, once N_2 detects that N_1 is out of service, it can identify that N_4 has all copies necessary for T (remember that the global data placement is known) and request the transfer of *RT* to N_4. So, we assume that *RT*'s logs are kept at each node (see Section 4.4). In addition, if N_4 is also out of service, then no node can perform T. Thus, N_2 would abort transaction T. Consistency is enforced because none of the active nodes has performed the transaction. In this case, at recovery time, the failed nodes would undo T.

4.3. Correctness of the refresher algorithm

In this section we show that the refresher algorithm is correct. The proofs for the lazy master based configurations appear in [12] and we do not re-discuss them here. The proofs for partial configurations we consider come directly from those of lazy-master and multi-master configurations, as we will show.

Lemma 4.1. *The refreshment algorithm is correct for multi-master configurations.*

Proof: Let us consider any node N of a multi-master configuration holding multi-owner copies. Let T be any transaction committed by node N. The propagator located at node N will propagate the operations performed by T by means of a message using reliable multicast. Hence any node involved in the execution of the transaction receives the update message. Since (i) the message containing the timestamp of any transaction T is the last one related to that transaction, and (ii) the reliable multicast preserves the global FIFO order, when a node *N'* receives the message containing the timestamp of *T* (i.e., at delivery time $C + Max + \varepsilon$), it has previously received all operations related to *T* and involving that node. Hence the transaction can be committed when all its operations are done and earliest at delivery time $C + Max + \varepsilon$. □

Lemma 4.2. *The refreshment algorithm is correct for partial configurations.*

Proof: Let us consider any node *N* of a partial configuration holding at least one multi-owner copy. Let *T* be any transaction submitted to node *N*, so *N* is the origin node of *T*. When the update message is received by any node involved in the execution of the transaction, by Lemma 4.1, transaction *T* can be committed when all its operations are done and earliest at delivery time $C + Max + \varepsilon$. But in the case where the node does not hold all the copies necessary to the transaction, *T* waits. Since an origin node must hold all the copies necessary to the transaction submitted by a client, the node *N* can perform *T*. Then, node *N* produces and multicasts *RT* which contains the write set associated to *T* to all waiting target nodes. So, the waiting target nodes can perform *T* by replacing the content of the transaction by its write set. Hence the transaction is still committed earliest at delivery time $C + Max + \varepsilon$. □

Lemma 4.3 (Transaction chronological order). *The refreshment algorithm ensures that, if* T_1 *and* T_2 *are any two transactions that start execution at global times* t_1 *and* t_2*, respectively, then: if* $t_2 - t_1 > \varepsilon$*, the timestamps* C_2 *for* T_2 *and* C_1 *for* T_1 *satisfy* $C_2 > C_1$*; any node that commits both* T_1' *and* T_2'*, commits them in the order given by* C_1 *and* C_2*.*

Proof: Let us assume that $t_2 – t_1 > \varepsilon$ Even if the clock of the node committing T_1 is ε ahead with regard to the clock of the node committing T_2, we have $C_2 > C_1$. We now assume that we have $C_2 > C_1$ and we consider a node *N* that commits first T_1' and then T_2'. According to the algorithm, T_2' is not committed before local time $C_2 + Max + \varepsilon$. At that time, if *N* commits T_2' before T_1', it means that *N* has not received the message related to T_1. Since clocks are ε synchronised, that message would have experienced a multicast delay higher than *Max*. □

Lemma 4.4 (Total order). *The refreshment algorithm satisfies the total order criterion for any configurations.*

Proof: If the refreshment algorithm is correct (Lemmas 4.1 and 4.2) and the transactions are performed in chronological order on each node (Lemma 4.3), then the total order is enforced. □

Lemma 4.5 (Deadlock). *The refreshment algorithm ensures that no deadlock appears.*

Proof: Let us consider a transaction T_1 which has for origin node N_1 and waits for its write set at node N_2 and a transaction T_2 which has for origin node N_2 and waits for its write set at N_1. A deadlock appears if and only if T_1 is performed before T_2 on N_2 and if T_2 is performed before T_1 on N_1. Hence, the total order is not enforced. This contradicts Lemma 4.3 since transactions are always performed in their chronological order at all the nodes. □

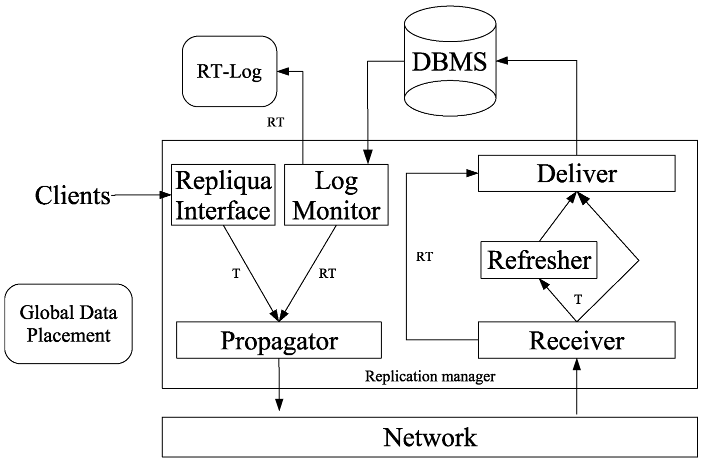


Figure 7. Replication Manager architecture.

4.4. Replication Manager architecture

In this section, we present the Replication Manager architecture to implement the Preventive Partial Replication algorithm (see figure 7). We add several components to a regular DBMS while preserving node autonomy, i.e. without requiring the knowledge of system internals. The Replica Interface receives transactions coming from the clients. The Propagator and the Receiver manage the sending and reception (respectively) of transactions and refresh transactions inside messages within the network.

Whenever the Receiver receives a transaction, it places it in the appropriate pending queue, used by the Refresher, and in the running queue used by the Deliver to start its execution. Next, the Refresher executes the refreshment algorithm to ensure strong consistency. The Deliver submits transactions, read from the running queue, to the DBMS and commits them only when the Refresher ensures that the transactions have been performed in chronological order.

With partial replication, when a transaction *T* is composed of a sequence of reads and writes, the Refresher at the target nodes must assure correct ordering. However, in case where the node does not hold all the necessary copies, *T* 's execution must be delayed until its corresponding refresh transaction *RT* is received. This is because *RT* is produced only after the commitment of the corresponding *T* at the origin node. At the target node, the content of *T* (sequence of read and write operations) is replaced by the content of the *RT* (sequence of write operations) in the Deliver. Thus, at the target node, when the Receiver receives *RT*, it interacts directly with Deliver.

The Log Monitor constantly checks the content of the DBMS log to detect whether replicas have been updated. For each transaction *T* that updated a replica, it produces a corresponding refresh transaction. At the origin node, whenever the corresponding transaction is composed of reads and writes and some of the target nodes do not hold all the necessary replicas, the Log Monitor submits the refresh transaction to the propagator, which multicasts it to those nodes. Then, upon receipt of the refresh transaction, the target nodes can perform the corresponding waiting transaction. To provide fault-tolerance in case of failure of the origin node (see Section 4.2), Log Monitor stores *RT*, in addition to the origin node, in all the nodes that are able to perform the transaction (nodes which hold all necessary replicas to perform a transaction *T*). Thus, in case of failure of the origin node, one of these nodes can replace the origin node and multicast the *RT* to the target nodes that can not perform the corresponding *T*.

5. Improving response time

In this section, we present optimizations for both Full and Partial Replication that improve transaction throughput. First, we modify the algorithm to eliminate partially the delay times ($Max + \varepsilon$) before submitting transactions. Then, we introduce concurrency control features in the algorithm to improve transaction throughput. Finally, we show the correctness of these optimizations.

5.1. Eliminating delay time

In a cluster network (which is typically fast and reliable), in most cases messages are naturally chronologically ordered [16]. Only a few messages can be received in an order that is different than the sending order. Based on this property, we can improve our algorithm by submitting a transaction to execution as soon as it is received, thus avoiding the delay before submitting transactions. Yet, we still need to guarantee strong consistency. In order to do so, we schedule the commit order of the transactions in such a way that a transaction can be committed only after $Max + \varepsilon$. Recall that to enforce strong consistency, all the transactions must be performed according to their timestamp order. So, a transaction is out-of-order when its timestamp is lower than the timestamps of the transactions already received. Thus, when a transaction *T* is received out-of-order, all younger transactions must be aborted and re-submitted according to their correct timestamp order with respect to *T*. Therefore, all transactions are committed in their timestamp order.

Thus, in most cases the delay time ($Max + \varepsilon$) is eliminated. Let *t* be the time to execute transaction *T*. In the previous algorithm [13], the time spent to refresh a multi-master copy, after reception of *T*, is $Max + \varepsilon + t$. Now, a transaction *T* is ordered while it is executed. So, the time to refresh a multi-master copy is max[($Max + \varepsilon$), *t*]. In most cases, *t* is higher than the delay $Max + \varepsilon$. Thus, this simple optimization can well improve throughput as we show in our performance study.

Figure 8 shows part of the components necessary to run our algorithm. The *Refresher* reads transactions from the head of *pending queues* and performs chronological ordering with respect to the delivery times. Once a transaction *T* is ordered, the refresher notifies *Deliver* that *T* is ordered and ready to be committed. Meanwhile, Deliver keeps checking the head of the running queue to start transaction execution optimistically, one after the other, inside the local *DBMS*. However, to enforce strong consistency Deliver only commits a transaction when the Refresher has signaled it.

Let us illustrate the algorithm with an example from figure 8. Suppose we have a node *i* that holds the master of the copy *R*. Node *i* receives T_1 and T_2, two transactions that update

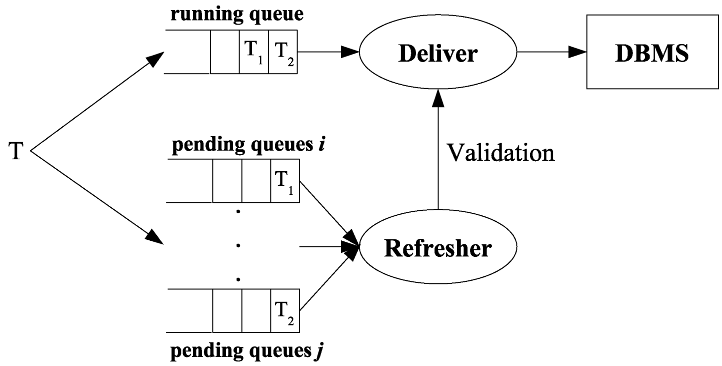


Figure 8. Refreshment architecture.

R, respectively from node i with a timestamp $C_1 = 10$ and from node j with a timestamp $C_2 = 15$. T_1 and T_2 must be performed in chronological order, T_1 then T_2. Let us see what happens when the messages are not received chronologically ordered at node i. In our example, T_2 is received before T_1 at node i and immediately written into the running queue and the corresponding pending queue. Thus, T_2 is submitted to execution by the Deliver but must wait the Refresher's decision to commit T_2. Meanwhile, T_1 is received at node i, it is similarly written into both pending and running queues. However the Refresher detects that the younger transaction T_2 has already been submitted before T_1. So, T_2 is aborted and re-started, causing it to be re-inserted into the running queue (after T_1). T_1 is chosen to be the next transaction to commit. Finally, T_2 is performed and elected to commit by Refresher. Thus, the transactions are committed in their timestamp order, even if they have been received unordered.

Preventive algorithm details. We can define three different states for a transaction T represented in figure 9. When a transaction T arrives at the Replication Manager, its state is

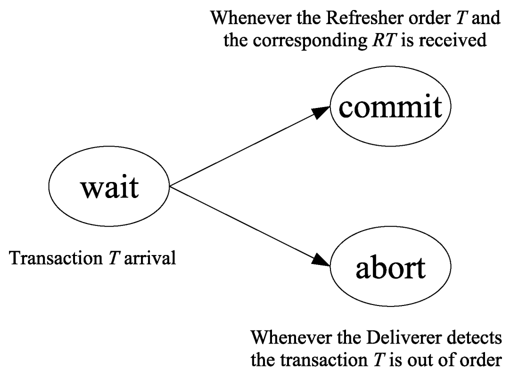


Figure 9. Transition state graph for T.

initialized to *wait*. Then, when T can be executed (a transaction can be executed when the node holds all necessary replicas or when its corresponding *RT* is received), and when the Refresher has ordered the transaction, the state of Transaction T is set to *commit*. Finally, when the Deliver receives an out-of-order transaction T (its timestamp is lower than the timestamps of the transactions already received), the state of the current running transactions is set to *abort*.

The Preventive algorithm is described in detail in figures 10 and 11. Figure 10 describes the Refresher algorithm. The Refresher selects the next totally ordered transaction. A transaction is guaranteed to be totally ordered at its *delivery_time* ($C + Max + \varepsilon$). Thus, in step 1, on the arrival of a new transaction, the refresher chooses the oldest transaction T

```
Partial Replication Refresher

Input:
 pending queues q1,…qn
Variables:
 curr_T:    currently selected transaction to be ordered;
 first_T:   transaction with the lowest timestamp
            in the pending queues;
 running_T: transactions in the running queue;
 timer:     local reverse timer whose state is
            either active or inactive;

Begin
 timer.state = inactive;
 curr_T = 0
 Repeat

  Step 1:
  On arrival of a new transaction in the pending queue
  or when timer expires do
   first_T <- message with min timestamp among top
              messages of the pending queues;
   If first_T <> curr_T then
    curr_T <- first_T;
    calculate delivery_time(curr_T);
    timer.value <- delivery_time(curr_T) - local_time
    timer.state <- active;
   endif

  Step 2:
   If timer expires then
    For all transactions running_T in the running queue
    such as curr_T = running_T do
     running_M.state = commit;
    end for;
    Dequeue curr_T from its pending queue;
    timer.state <- inactive;
   endif;
 for ever;
end;
```

Figure 10. Partial replication refresher algorithm with elimination of delay times.

```
Partial Replication Deliver (no concurrency)

Input:
 running queue, RT list
Variables:
 new_T: new incoming transactions from the running queue
 curr_T: currently running transaction;
 new_RT: new incoming refresh transactions from the RT list
Begin
 curr_T = 0;
 Repeat
  On arrival of a new transaction new_T
  or at the end of curr_T
  or the state of curr_T changes
  or on arrival of new refresh transaction new_RT do

  Step 1:
   If curr_T finishes performing then
    If curr_T.state = abort then
     Rollback curr_T;
     Dequeue curr_T from the running queue;
     curr_T = 0;
    elseif curr_T.state = commit then
     Commit curr_T;
     Dequeue curr_T from the running queue;
     curr_T = 0;
    endif;
   endif;

  Step 2:
   If new_T <> 0 then
    new_T.state = wait;
    If curr_T <> 0 and curr_T.C > new_T.C then
     curr_T.state = abort;
     Introduce a copy of curr_T in the running queue
     according to its timestamp with a wait state;
    end if;
   end if;

  Step 3:
   If curr_T = 0 and the running queue is not empty then
    curr_T <- the transaction at the top of
              the running queue;
    If node holds all copies necessary to curr_T then
     curr_T.standby = false;
     Perform curr_T;
    else
     curr_T. standby = true;
    end if;
   end if;

  Step 4:
   If curr_T <> 0 and curr_T.standby = true then
    Extract the new_RT corresponding to curr_T from RT_list;
    If new_RT <> 0 then
     new_RT.state = curr_T.state;
     curr_T = new_RT;
     curr_T.standby = false;
     Perform curr_T;
    end if;
   end if;
 for ever;
end;
```

Figure 11. Partial replication deliver algorithm with elimination of delay times.

from the top of the pending queues and computes *T*'s *delivery_time*. Next the Refresher initializes a timer that will expire at *T*'s *delivery_time*. So, if the incoming transaction *T* is not out-of-order according to the current selected transaction, *curr_T*, nothing happens. In the other case, the new *T*'s *delivery_time* is calculated according to *T*'s timestamp. In step 2, when the timer expires, the Refresher looks for the non aborted transactions corresponding to *curr_T*. Then, it sets the state of *curr_T* to commit.

Figure 11 describes the Deliver algorithm in the optimistic arrival approach. Deliver reads transactions from the running queue and executes them. If a transaction *T* is out-of-order, Deliver aborts the current running transaction, *curr_T*, and executes *T* followed by *curr_T*. Deliver commits a transaction when the Refresher sets its state to commit. In step 1, at the end of the execution of the current transaction *curr_T*, Deliver commits or rolls-back *curr_T* according to its state (commit or abort). Since we do not have access to the transaction manager of the DBMS, we cannot abort directly the transactions and we must wait until the end of the transaction to abort it. In step 2, Deliver sets the state of the newly received transaction (*new_T*) to wait and checks whether *new_T* is not an out-of-order transaction. If the transaction is out-of-order, the state of the current transaction (*curr_T*) is set to abort. As the Deliver has to wait the end the transaction to rollback the transaction, a copy of *curr_T* is reintroduced in the running queue and its state is set to wait while the aborted *curr_T* is running. Thus, a transaction *T* aborted due to an unordered message will be re executed from a copy of *curr_T*. In step 3, the Refresher selects the transaction at the top of the running queue and performs it if the node holds all the copies necessary to the transaction, Otherwise, the Refresher set the transaction in stand by. Finally, in step 4, on arrival of a new refresh transaction *new_RT*, the Deliver replaces the content of the waiting *T* by the content of its corresponding *RT*. So, the current transaction can execute.

5.2. Improving transaction throughput

To improve throughput, we now introduce concurrent replica refreshment. In the previous section, the Receiver writes transactions directly into the running queue (optimistically), and afterwards the Deliver reads the running queue contents in order to execute the transaction, and in the other hand, to assure consistency, the same transactions are written as usually in the pending queues to be ordered by the Refresher. Hence, the Deliver extracts the transactions from the running queue and performs them one by one in serial order. So, if the Receiver fills the running queue faster than the Deliver empties it, and if the average arrival rate is higher than the average running rate of a transaction (typically in bursty workloads), the response time increases exponentially and performance degrades.

To improve response time in bursty workloads we propose to trigger transactions concurrently. In our solution, concurrency management is done outside the database to preserve autonomy (different from [9]). Using the existing isolation property of database systems [11], at each node, we can guarantee that each transaction sees a consistent database at all times. To maintain strong consistency at all nodes, we enforce that transactions are committed in the same order in which they are submitted. In addition, we guarantee that transactions are submitted in the order in which they have been written to the running queue. Thus, total order is always enforced.

However, without access to the DBMS concurrency controller (for autonomy reasons), we cannot guarantee that two conflicting concurrent transactions obtain a lock in the same order at two different nodes. Therefore, we do not trigger conflicting transactions concurrently. To detect that two transactions are conflicting, we determine a subset of the database items accessed by the transaction according to the transaction. If the subset of a transaction does not intersect with a subset of another transaction, then the transactions are not conflicting. For example, in the TPC-C benchmark, the transactions' parameters allow us to define a subset of tuples that could be read or updated by the transaction. Notice that if the subset of the transaction cannot be determined, then we consider the transaction to be conflicting with all other transactions. This solution is efficient if most transactions are known, which is true in OLTP environments.

We can now define two new conditions to be verified by the Deliver before triggering and before committing a transaction:

(i) Start a transaction iff the transaction is not conflicting with transactions already started (but not committed) and iff no older transaction waits for the commitment of a conflicting transaction to start.
(ii) Commit a transaction iff no older transactions are still running.

Figure 12 shows examples of concurrent executions of transactions. Figure 12(a) illustrates a case where the transactions are triggered sequentially, which is equivalent to the case where all the transactions are conflicting. Figures 12(b), (c) and (d) show parallel executions of transaction T_1, T_2 and T_3. In figures 12(b) and (c), transaction T_2 finishes before T_1 but waits for commit because T_1 is still running (this is represented by a dashed line in the figure). In figure 12(b), T_1, T_2 and T_3 are not conflicting, so they can run concurrently. On the other hand, in figure 12(c), T_2 is conflicting with T_3, so T_3 must wait for the end of T_2 before starting. Finally, in figure 12(d), T_1 and T_2 are conflicting, so T_2 cannot start before the commitment of T_1 and T_3 cannot start before T_2 because transactions must be executed in the order they are in the running queue.

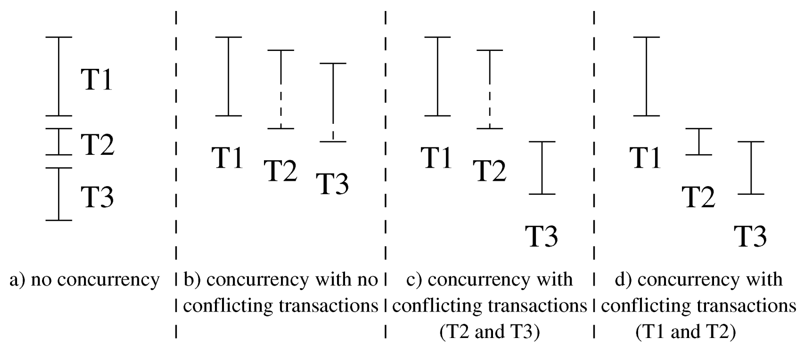


Figure 12. Example of concurrent execution of transactions.

5.3. Correctness

In this section, we prove that the Preventive Replication algorithm is also correct with the optimizations.

Lemma 5.1. *The elimination of the delay Max + ε does not introduce inconsistency.*

Proof: Let T_1 and T_2 be any two transactions with timestamps C_1 and C_2. If T_1 is older than T_2 ($C_1 < C_2$) and T_2 is received on node i before T_1, then T_2 is managed optimistically. However T_2 cannot be committed before $C_2 + Max + \varepsilon$, and as T_1 is received at node i at the latest at $C_1 + \text{Max} + \varepsilon$, then, T_1 is received before T_2 is committed ($C_1 + Max + \varepsilon < C_2 + Max + \varepsilon$). Therefore, T_2 is aborted, and both transactions are written in the running queue, executed and committed according to their timestamp values. Afterwards, T_1 is executed before T_2, and the strong consistency is enforced even in the case of unordered messages. □

Lemma 5.2. *The parallel execution of transactions does not break the enforcement of strong consistency.*

Proof: Let T_1 and T_2 be any two transactions with timestamps C_1 and C_2 that start execution at times t_1 and t_2, and commit at times c_1 and c_2, respectively. In the case where T_1 and T_2 are received unordered, the transactions are aborted and re-executed in the correct order as described in Lemma 5.2. Now, in the case where the transactions are received correctly ordered, if T_1 and T_2 are conflicting, they start and commit one after the other according to their timestamp values. Hence, if $C_1 < C_2$, then $t_1 < c_1 < t_2 < c_2$. If they are not conflicting, T_2 can start before T_1 commits. However, a transaction is never committed before all older transactions have been committed. If $C_1 < C_2$, then $t_1 < t_2$ and $c_1 < c_2$. Thus, the state of the database viewed by a transaction before its execution and its commitment is the same at all the nodes. Hence, strong consistency is enforced. □

6. Validation

In this section, we describe our implementation and our performance model. Then, we describe two experiments to study scale up and speed-up.

6.1. Implementation

We implemented our Preventive Replication Manager in our RepDB∗ prototype [2, 19] on a cluster of 64 nodes (128 processors). Each node has 2 Intel Xeon 2.4 GHz processors, 1 GB of memory and 40 GB of disk. The nodes are linked by a 1 Gb/s network. We use Linux Mandrake 8.0/Java and CNDS's Spread toolkit that provides a reliable FIFO message bus and high-performance message service among the cluster nodes. We use PostgreSQL Open Source DBMS at each node. We chose PostgreSQL because it is quite complete in terms of transaction support and easy to work with.

Our implementation has four modules: Client, Replicator, Network and Database Server. The Client module simulates the clients. It submits transactions randomly to any cluster node, via RMI-JDBC, which implements the Replica Interface. Each cluster node hosts a Database Server and one instance of the Replicator module. For this validation, we implemented most of the Replicator module in Java outside of PostgreSQL. For efficiency, we implemented the Log Monitor module inside PostgreSQL. The Replicator module implements all system components necessary for a multi-master node: Replica Interface, Propagator, Receiver, Refresher and Deliver. Each time a transaction is to be executed, it is first sent to the Replica Interface that checks whether the incoming transaction updates a replica. Whenever a transaction does not write a replica, it is sent directly to the local transaction manager. Even though we do not consider node failures in our performance evaluation, we implemented all the necessary logs for recovery to understand the complete behavior of the algorithm. The Network module interconnects all cluster nodes through the Spread toolkit.

6.2. *Performance model*

To perform our experiments, we use the TPC-C Benchmark [18] which is an OLTP workload with a mix of read-only and update intensive transactions. It has 9 tables: Warehouse, District, Customer, Item, Stock, New-order, Order, Order-line and History; and 5 transactions: Order-status, Stock-level, New-order, Payment and Delivery.[2]

The parameters of the performance model are shown in Table 1. The values of these parameters are representative of typical OLTP applications. The size of the database is proportional to the number of warehouses (a tuple in the Warehouse table represents a warehouse). The number of warehouses also determines the number of clients that submit a transaction. As specified in the TPC-C benchmark, we use 10 clients per warehouse. For a client, we fix the transaction arrival rate λ_{client} at 10 s. So with 100 clients (10 warehouses and 10 clients per warehouse), the average transactions' arrival rate λ is 100 ms. In our experiments, we vary the number of warehouses W to be either 1, 5 or 10. Then, the different average transactions' arrival rates are 1 s, 200 ms and 100 ms.

During an experiment, each client submits to a random node a transaction among the 4 TPC-C transactions used. In the end, each client must have submitted M transactions

Table 1. Performance parameters

Parameter	Definition	Values
W	Number of warehouse	1, 5, 10
Clients	Number of clients by warehouse	10
λ_{client}	Average arrival rate for each client	10 s
λ	Average arrival rate	1 s, 200 ms, 100 ms
Conf.	Replication of tables	FR, PR
M	Number of transactions submitteddurning the tests for each client	100
$Max + \varepsilon$	Delay introduced for submitting a Transaction	200 ms

and must have maintained a percentage of mixed transactions: 6% for Order-status, 6% for Stock-level, 45% for New-order and 43% for Payment.

The TPC-C defines a number of different types of transactions. New-order represents a mid-weight, read-write transaction with a high frequency of execution. Payment represents a lightweight, read-write transaction with a high frequency of execution. Order-status represents a mid-weight, read-only transaction with a low frequency of execution. Stock-level represents a heavy, read-only transaction with a low frequency of execution. Thus, we can consider New-order and Payment as multi-master transactions.

Finally, for our experiments, we use two replication configurations. In the *Fully Replicated* (*FR*) configuration all the nodes carry all the tables as multi-master copies. In the *Partially Replicated* (*PR*) configuration, one fourth of the nodes hold tables needed by the Order-status transaction as multi-master copies, another fourth holds tables needed by the New-order transaction as multi-master copies, another fourth holds tables needed by the Payment transaction as multi-master copies and the last fourth holds tables needed by the Stock-level transaction as multi-master copies.

6.3. Scale up experiments

These experiments study the algorithm's scalability. That is, for a same set of incoming transactions (New-order and Payment transactions), scalability is achieved whenever increasing the number of nodes yields the same response times. We vary the number of nodes for each configuration (*FR* and *PR*) and for different numbers of warehouses (1, 5 and 10). For each test, we measure the average response time per transaction. The duration of this experiment is the time to submit 100 transactions for each client.

The experimental results (see figure 13) show that for all tests, scalability is achieved. The performance remains relatively constant according to the number of nodes. Our algorithm has linear response time behavior even when the number of node increases. Let n be the number of target nodes for each incoming transaction, our algorithm requires only the multicast of n messages for the nodes that carry all required copies

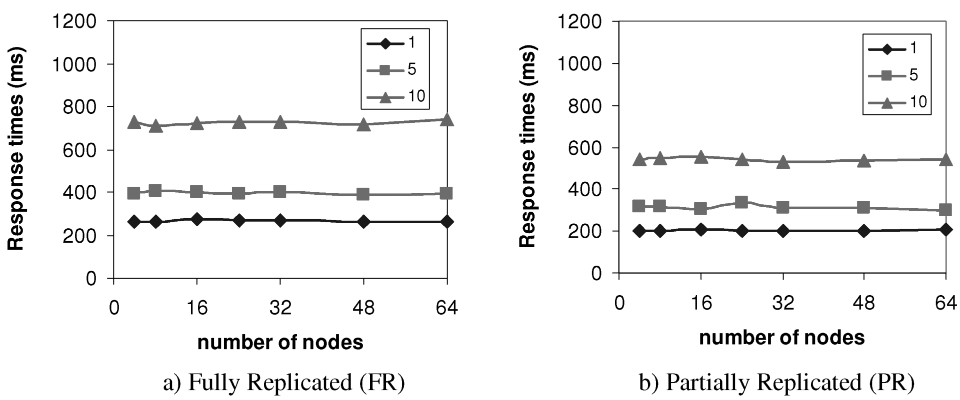


Figure 13. Scale up results.

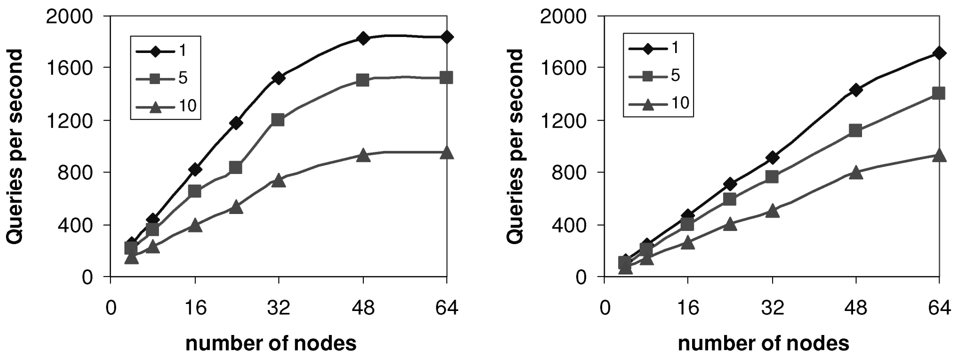
plus $2n$ messages for the nodes that do not carry all required copies. The performance decreases as the number of warehouses increases (which increases the workload). In figure 13(a), although the workload is twice higher for 10 warehouses than for 5 warehouses, the response times remain twice as worse as expected, i.e., 400 ms for 5 warehouses and about 800 ms for 10 warehouses. This demonstrates that our algorithm has good response time when the workload increases and we can expect similar behavior with higher workloads.

The results also show the impact of the configuration on transaction response time. As the number of transactions increases (with the number of nodes that receive incoming transactions), *PR* increases inter-transaction parallelism more than *FR* by allowing different nodes to process different transactions. Thus, transaction response time is slightly better with *PR* (figure 13(a) than with *FR* (figure 13(b)) by about 15%. In *PR*, nodes only hold tables needed by one type of transaction, so they do not have to perform the entire updates of the other type of transactions. Hence, they are less overloaded than in *FR*. Thus the configuration and the placement of the copies should be tuned to selected types of transactions.

6.4. Speed-up experiments

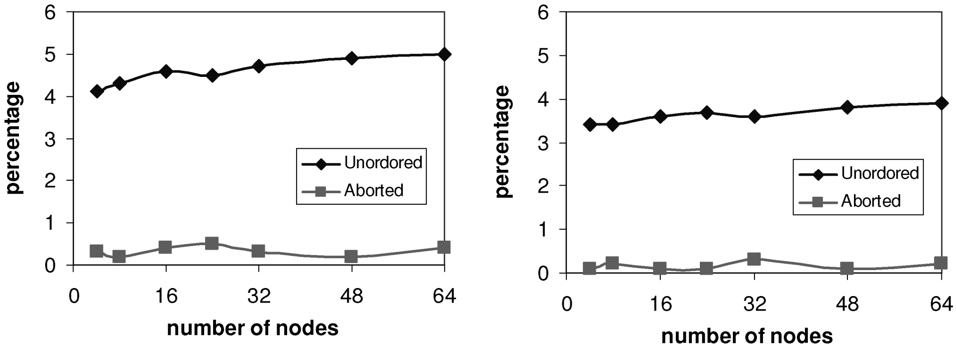
These experiments study the performance improvement (speed-up) for read queries when we increase the number of nodes. To test speed-up, we reproduced the previous experiments and we introduced clients that submit queries. We vary the number of nodes for each configuration (*FR* and *PR*) and for different number of warehouses (1, 5 and 10). The duration of this experiment is the time to submit 100 transactions for each client.

The number of clients that submit queries is 128. The clients submit lightweight queries (Order-status transaction) sequentially while the experiment is running. Each client is associated to one node and we produce an even distribution of clients at each node. Thus, the number of read clients per node is 128 divided by the number of nodes that support the



a) Fully Replicated (FR) b) Partially Replicated (PR)

Figure 14. Speed-up results.



a) Fully Replicated (FR) b) Partially Replicated (PR)

Figure 15. Percentage of unordered messages and aborted transactions for 10 warehouses.

Order-status transaction. For each test, we measured the throughput of the cluster, i.e. the number of read queries per second.

The experiment results (see figure 14) show that the increase in the number of nodes improves the cluster's throughput. For example in figure 14(a), whatever the number of warehouses, the number of queries per seconds with 32 nodes (1500 queries per seconds) is almost twice that with 16 nodes (800 queries per seconds). However, if we compare *FR* with *PR*, we can see that the throughput is better with *FR*. Although the nodes are less overloaded than in *FR*, performance is half of *FR* because only half of the nodes support the transaction. This is due to the fact that, in *PR*, not all the nodes hold all the tables needed by the read transactions. In *FR*, beyond 48 nodes, the throughput does not increase anymore because the optimal number of nodes is reached, and the queries are performed as fast as possible.

6.5. Effect of optimistic execution

Now, we study the effect of optimistically executing transactions as soon as they arrive. Our first study shows the impact of the unordered messages on the number of aborted transactions due to optimistic execution (see Section 5.1). Then, our second study shows the gain of the optimistic approach on the refreshment delay.

In our first experiment, figures 15(a) and (b) show the percentage of the unordered messages and the percentage of the aborted transactions for the scale up experiment (Section 6.3). Below 5% of the messages are unordered, and only 1% of the transactions are aborted. At most only 20% of the unordered messages introduce aborts because two unordered messages are received in a very short period of time (around 2 ms). So, the second message is received before the first message has been processed. Therefore, they are reordered before the execution of the first message.

For *PR*, the Partially Replicated configuration (figure 15(b)), the percentage of the unordered messages is lower than the percentage for *FR*, the Fully Replicated configuration

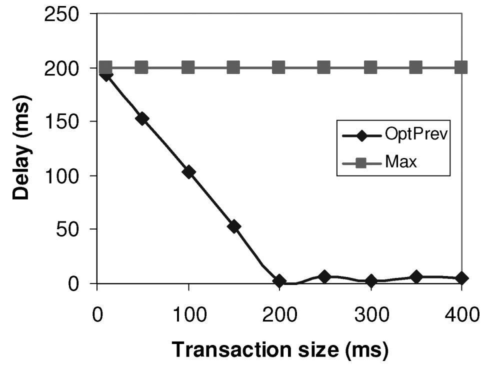


Figure 16. Delay versus transaction size.

(figure 15(a)), because less messages are involved. Thus, the number of aborted transactions is small enough to warrant the gain introduced by the elimination of the delay time.

In our second experiment, we study how the transaction size affects the elimination of the refreshment delay. In the Optimistic Approach, transactions still need to be delayed ($Max + \varepsilon$) before committing. Figure 16 shows the relative importance of the delay time with respect to transaction size. Our test involves only 8 nodes with a FR configuration because the waiting time is not affected by the increase of the number of nodes. We submit 100 transactions in a low workload and we vary the size of the transaction. Then, we measure the delay time introduced by the refreshment algorithm. Recall that the normal delay (Max) value is 200 ms without optimization.

An important observation is that the delay introduced by the refreshment quickly decreases as the transaction time increases. This is due to the fact that, since the transaction is performed as soon as possible, the scheduling of a transaction is performed in parallel with its execution. As the scheduling time is equal to $Max + \varepsilon$, the delay introduced is equal to Max $+ \varepsilon$ minus the size of the transaction. For example, with a transaction size of 50 ms, the delay is 150 ms. Thus, with transactions longer than 200 ms, the delay is almost zero because the scheduling time is included in the execution time. Hence, the gain is almost equal to Max, which is the optimal gain for the elimination of Max. Finally, the number of aborted transactions is not enough significant, so we do not put it on the figure.

7. Related work

Data replication has been extensively studied in the context of distributed database systems [11]. In the context of database clusters, the main issue is to provide scalability (to achieve performance with large numbers of nodes) and autonomy (to exploit black-box DBMS) for various replication configurations such as master-slave, multi-master and partial replication.

Synchronous (eager) replication can provide strong consistency for most configurations including multi-master but its implementation, typically through 2PC, violates system autonomy and does not scale up. In addition, 2PC may block due to network or node failures. The synchronous solution proposed in [9] reduces the number of messages exchanged to commit transactions compared to 2PC. It uses, as we do, group communication services to guarantee that messages are delivered at each node according to some ordering criteria. However, DBMS autonomy is violated because the implementation must combine concurrency control with group communication primitives. In addition solutions based on total order broadcast is not well suited for large scale replication because as the number of nodes increases the overhead of messages exchanged may dramatically increase to assure total order. The Database State Machine [20, 21] supports partial replication for heterogeneous databases and thus does not violate autonomy. However, its synchronous protocol uses two-phase locking that is known for its poor scalability, thus making it inappropriate for database clusters.

Asynchronous (lazy) replication typically trades consistency for performance. A refreshment algorithm that assures correctness for lazy master configurations is proposed in [12]. This work does not consider multi-master and partial replication as we do. The preventive replication solution in [13] is asynchronous and achieves strong consistency for multi-master configurations. However, it introduces heavy message traffic in the network since transactions are multicast to all cluster nodes. In [3], we extended preventive replication to deal with partial replication. However, it also has performance limitations since transactions are forced to wait a delay time before executing. The solution proposed in this paper addresses these important limitations.

The algorithm proposed in [8] provides strong consistency for multi-master and partial replication while preserving DBMS autonomy. However, it requires that transactions update a fixed primary copy: each type of transaction is associated with one node so a transaction of that type can only be performed at that node. This is a problem for update intensive applications. For example, with the TPC-C benchmark, two nodes support 88% of the transactions (45% at one node for the New Order transactions and 43% at another node for the Payment transactions). Furthermore, the algorithm uses 2 messages to multicast the transaction, the first is a reliable multicast and the second is a total ordered multicast. The cost of these messages is higher than the single FIFO multicast message we use. Furthermore, using a logical total order message increases the overhead of physical messages exchanged when increasing the number of nodes. However, one advantage of this algorithm is that it avoids redundant work: the transaction is performed at the origin node and the target nodes only apply the write set of the transaction. In our algorithm, all the nodes that hold the resources necessary for the transaction perform it entirely. We could also remove this redundant work to generalize the multicast of refresh transactions for all nodes instead of only for the nodes that do not hold all the necessary replicas. However, the problem is to decide whether it is faster to perform the transaction entirely or to wait for the corresponding write set from the origin node for short transactions. Finally their experiments do not show scale-up with more than 15 nodes while we go up to 64 in our experiments.

More recent work has focused on snapshot isolation to improve the performance of read-only transactions. The RSI-PC [17] algorithm is a primary copy solution which separates update transactions from read-only transactions Update transactions are always routed to a main replica, whereas read-only transactions are handled by any of the remaining replicas, which act as read-only copies. Postgres-R(SI) [24] proposes a smart solution that does not need to declare transactions properties in advance. It uses the replication algorithm of [8] which must be implemented inside the DBMS. The experiments are limited to at most 10 nodes. SI-Rep [10] provides a solution similar to Postgres-R(SI) on top of PostgreSQL which needs the write set of a transaction before its commitment. Write sets can be obtained by either extending the DBMS, thus hurting DBMS autonomy, or using triggers.

8. Conclusion

In this paper, we introduced two algorithms for preventive replication in order to scale up to large cluster configurations. The first algorithm supports fully replicated configurations where all the data are replicated on all the nodes, while the second algorithm supports partially replicated configurations, where only a part of the data are replicated. Both algorithms enforce strong consistency. Then, we proposed a complete architecture that supports a large numbers of configurations. Moreover, we presented two optimizations that improve transaction throughput; the first optimization eliminates optimistically the delay introduced by the preventive replication algorithm while the second optimization introduces concurrency control features outside the DBMS in which non conflicting incoming transactions may execute concurrently.

We did an extensive performance validation based on the implementation of Preventive Replication in our RepDB∗ prototype over a cluster of 64 nodes running PostgreSQL. Our experimental results using the TPC-C benchmark show that our algorithm scales up very well and has linear response time behavior. We also showed the impact of the configuration on transaction response time. With partial replication, there is more inter-transaction parallelism than with full replication because of the nodes being specialized to different tables and thus transaction types. Thus, transaction response time is better with partial replication than with full replication (by about 15%). The speed-up experiment results showed that the increase of the number of nodes can well improve the query throughput. Finally, we showed that, with our optimistic approach, unordered transactions introduce very few aborts (at most 1%) and that the waiting delay for committing transactions is very small (and reaches zero as transaction time increases). To summarize, the performance gains strongly depend on the types of transactions and of the configuration. Thus an important conclusion is that the configuration and the placement of the copies should be tuned to selected types of transactions.

Notes

1. For any two nodes, the same sequence of transactions is executed in the same order.
2. For our experiments, we do not use the delivery transaction because it is executed in a deferred mode that is not relevant to test the response times on which our measures are based.

References

1. T. Anderson, Y. Breitbart, H. Korth, and A. Wool, "Replication, consistency, and practicality: Are these mutually exclusive"? in SIGMOD Conference, 1998, pp. 484–495.

2. C. Coulon, G. Gaumer, E. Pacitti, and P. Valduriez, The RepDB∗ prototype: Preventive Replication in a Database Cluster, Base de Données Avancées, Montpellier, France, 2004.
3. C. Coulon, E. Pacitti, and P. Valduriez, "Scaling up the preventive replication of autonomous databases in cluster systems," in Int. Conf. on High Performance Computing for Computational Science (VecPar 2004), Valencia, Spain, 2004, Lecture Notes in Computer Science 3402, Springer 2005, pp. 174–188.
4. C. Coulon, E. Pacitti, and P. Valduriez, "Consistency management for partial replication in a high Performance database cluster," in IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS 2005), Fukuoka, Japan, 2005.
5. S. Gançarski, H. Naacke, E. Pacitti, and P. Valduriez, "Parallel processing with autonomous databases in a cluster system," in Int. Conf. on Cooperative Information Systems (CoopIS), 2002.
6. J. Gray, P. Helland, P. O' Neil, and D. Shasha, "The danger of replication and a solution," in ACM SIGMOD Int. Conf. on Management of Data, Montreal, 1996.
7. V. Hadzilacos and S. Toueg, Fault-Tolerant Broadcasts and Related Problems. Distributed Systems, 2nd edition, S. Mullender (Ed.), Addison-Wesley, 1993.
8. R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso, "Improving the scalability of fault-tolerant database clusters: early results," in Int. Conf. on distributed Computing Systems (ICDCS), 2002.
9. B. Kemme and G. Alonso, "Don't be lazy be consistent: Postgres-R, a new way to implement database replication," in Int. Conf. on Very Large Databases (VLDB), 2000.
10. Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris, "Middleware based data replication providing snapshot isolation," in ACM SIGMOD Int. Conf. on Management of Data. Baltimore, USA, June 2005.
11. T. Özsu and P. Valduriez, Principles of Distributed Database Systems, 2nd edition, Prentice Hall, 1999.
12. E. Pacitti, P. Minet, and E. Simon, "Replica Consistency in Lazy Master Replicated Databases," Distributed and Parallel Databases, Kluwer Academic, vol. 9, no. 3, 2001.
13. E. Pacitti, T. Özsu, C. Coulon, "Preventive Multi-Master Replication in a cluster of autonomous databases," in Euro-Par Int. Conf., 2003.
14. E. Pacitti and P. Valduriez, "Replicated Databases: Concepts, architectures and techniques," Network and Information Systems Journal, Hermès, vol. 1, no. 3, 1998.
15. Paris Project: http://www.irisa.fr/paris/General/cluster.htm.
16. F. Pedonne, "A schiper: optimistic atomic broadcast," in Distributed Information Systems Conf. (DISC), 1998.
17. C. Plattner and G. Alonso, "Ganymed: scalable replication for transactional web applications," in Proc. of the 5th International Middleware Conference, Toronto, Canada, 2004.
18. F. Raab, TPC-C—The Standard Benchmark for Online transaction Processing (OLTP). The Benchmark Handbook for Database and Transaction Systems, 2nd edition, Morgan Kaufmann, 1993.
19. RepDB∗: Data Management Component for Replicating Autonomous Databases in a Cluster System (released as open source software under GPL). http://www.sciences.univ-nantes.fr/lina/ATLAS/RepDB.
20. A. Sousa, F. Pedone, R. Oliveira, and F. Moura, "partial replication in the database state machine," in IEEE Int. Symposium on Network Computing and Applications (NCA), 2001.
21. A. Sousa, J. Pereira, F. Moura, and R. Oliveira, "Optimistic total order in wide area networks," in *Proc. 21st IEEE Symposium on Reliable Distributed Systems*, 2002, pp. 190–199.
22. K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," Micro-processors and Microprogramming, vol. 40, 1994.
23. P. Valduriez, "Parallel database systems: Open problems and new issues," Int. Journal on Distributed and Parallel Databases, Kluwer Academic, vol. 1, no. 2, 1993.
24. S. Wu and B. Kemme, "Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation," in IEEE Int. Conference on Data Engineering (ICDE), 2005.