

# Preventive Multi-Master Replication in a Cluster of Autonomous Databases<sup>1</sup>

Esther Pacitti<sup>1</sup>, M. Tamer Özsu<sup>2</sup>, Cédric Coulon<sup>1</sup>

<sup>1</sup>Atlas Group, INRIA and IRIN, Nantes – France  
Esther.Pacitti@irin.univ-nantes.fr, Cedric.Coulon@irin.univ-nantes.fr

<sup>2</sup>University of Waterloo – Canada  
tozsu@uwaterloo.ca

**Abstract.** We consider the use of a cluster of PC servers for Application Service Providers where applications and databases must remain autonomous. We use data replication to improve data availability and query load balancing (and thus performance). However, replicating databases at several nodes can create consistency problems, which need to be managed through special protocols. In this paper, we present a lazy preventive data replication solution that assures strong consistency without the constraints of *eager* replication. We first present a peer-to-peer cluster architecture in which we identify the replication manager. Cluster nodes can support autonomous, heterogeneous databases that are considered as black boxes. Then we present the multi-master refresher algorithm and show all system components necessary for implementation. Next we describe our prototype on a cluster of 8 nodes and experimental results that show that our algorithm *scales-up* and introduces a negligible loss of *data freshness* (almost equal to mutual consistency).

## 1 Introduction

Clusters of PC servers provide a cost-effective alternative to tightly-coupled multiprocessors. They 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 the provider site and need to be available, typically through the Internet, as efficiently as if they were local to the customer site. Thus, the challenge is to fully exploit the cluster's query parallelism and load balancing capabilities to improve performance. The typical solution is to replicate applications and data at different nodes so that users can be served by any of the nodes depending on the current load. This arrangement also provides high-availability since, in the event of a node failure, other nodes can still do the work. This solution has been successfully used by, for example, Web search engines using high-volume server farms (e.g., Google). However, Web sites are typically read-intensive which makes it easier to exploit parallelism. In the ASP context, the problem is far more difficult [1] since applications can be update-intensive. Replicating databases at

---

<sup>1</sup> Work partially funded by the Leg@net RNTL project.

several nodes, so they can be accessed by different users through the same or different applications in parallel, can create consistency problems [4]. Nevertheless, replication is the best solution to efficiently manage high transaction loads by exploiting parallelism.

In this paper, we present a data replication solution that assures strong consistency for parallel query processing in a cluster of autonomous databases. There are two basic approaches to manage data replication: *eager* and *lazy*. With *Eager replication* (e.g. Read-One-Write All – ROWA [9]) whenever a transaction updates one replica, all other replicas are updated inside the same transaction. Therefore mutual consistency of replicas and *strong consistency*<sup>2</sup> are enforced; however it violates system autonomy.

With *lazy replication* [4], a transaction can commit after updating one replica copy at some node. After the transaction commits, the updates are propagated to the other replicas, and these replicas are updated in separate transactions. Hence, the property of mutual consistency is relaxed and strong consistency is eventually assured. A major virtue of lazy replication is its easy deployment. In addition, lazy replication has gained considerable pragmatic interest because it is the most widely used mechanism to refresh data in several emerging distributed application environments [7]. In this paper, we focus on a particular lazy replication scheme, called *multi-master*, adapted for PC clusters.

In lazy replication, a *primary copy* is stored at a *master* node and *secondary copies* are stored in *slave* nodes. A primary copy that may be stored at and updated by different master nodes is called a *multi-owner copy*. These are stored in *multi-owner* nodes and a *multi-master* configuration<sup>3</sup> (or *lazy group*) consists of a set of multi-owner nodes on a common set of multi-owner copies.

This paper makes several contributions. First, we introduce architecture for processing user requests to applications into the cluster system and discuss our general solutions for submitting transactions and managing replicas. Second, we propose a multi-master refresher algorithm that *prevents* conflicts, by exploiting the cluster's high speed network, thus providing strong consistency, without the constraints of eager replication. We also show the architectural components necessary to implement the algorithm. Third, we describe the implementation of our algorithm over a cluster of 8 nodes and present experimental results that show that it scales-up and introduces a negligible loss of data freshness.

The rest of the paper is structured as follows. Section 2 presents the ASP cluster architecture in which we identify the role of the replication manager. Section 3 presents the refreshment management issues including the principles of the refresher algorithm and the architectural components necessary for its implementation. Section 4 describes our prototype over a cluster of 8 nodes and show our performance model and experiments results. Section 5 compares our work with the most relevant related work. Finally, section 6 concludes the paper.

---

<sup>2</sup> For any two nodes, the same sequence of transactions is executed in the same order.

<sup>3</sup> In addition to the multi-master that we consider in this paper, there are other lazy cluster configurations such as lazy master and hybrid. Our research addresses these as well, but space limitations force us to focus only on multi-master.

## 2 Multi-Master Cluster Architecture

In this section, we discuss the system issues and solutions when using the multi-master refreshment management to refresh multi-owner copies in a cluster of replicated databases. 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.

Shared-nothing is the only architecture that supports sufficient node autonomy without the additional cost of special interconnects. Thus, we exploit a shared-nothing architecture. Each cluster node is composed of five layers (see Figure 1): Request Router, Application Manager, Transaction Load Balancer and Replication Manager. We discuss these in the following. A 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. There is no significant data processing at the access node to avoid 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  $j$  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.

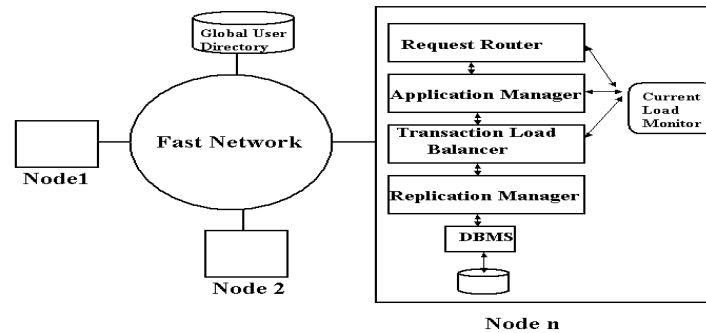


Fig. 1. Cluster Architecture

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 specified 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.

The *Application Manager* is the layer that manages application instantiation and execution using an application service 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 lightest transaction load.

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. Transaction execution is managed by the local transaction of each DBMS. Therefore the ACID (atomicity, consistency, isolation, durability) properties [9] are ensured. The Replication Manager signals to the Transaction load balancer whenever a transaction commits or abort. Afterwards the transaction load balancer signals the Application Manager the same event. We employ data replication because it provides database access parallelism for applications. Our preventive replication approach, implemented by the multi-master refreshment algorithm, avoids conflicts at the expense of a forced waiting time for transactions, which is negligible due to the fast cluster network system.

### 3 Multi-Master Refreshment

In this section, we present, in more detail, multi-master refreshment with its algorithm and system architecture. In a first step we consider fully replicated multi-master configurations<sup>4</sup>. We assume that the network interface provides a global FIFO reliable multicast: messages multicast by one node are received at the multicast group nodes in the order they have been sent. We denote by *Max*, the upper bound of the time needed to *multicast* a message from a node *i* to any other node *j* [3]. We also assume that each node has a local clock. For fairness reasons, clocks are assumed to have a drift and to be  $\epsilon$ -synchronized [3]. This means that the difference between any two correct clocks is not higher than the  $\epsilon$  (known as the *precision*). We now describe the multi-master refresher algorithm.

To define the algorithm, we need a formal correctness criterion to define strong consistency. In *Multi-Master* configurations, inconsistencies may arise whenever the serial orders of two multi-owner transactions at two nodes are not equal. Therefore, multi-owner transactions 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. Hence the following correctness criterion is necessary:

---

<sup>4</sup> N multi-master nodes. Each node stores the same set of multi-owner copies.

**Definition 3.1** (*Total Order*). Two multi-owner transactions  $MOT_1$  and  $MOT_2$  are said to be executed in Total Order if all multi-owner nodes that commit both  $MOT_1$  and  $MOT_2$  commit them in the same order.

**Proposition 3.1** 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.

We now present the refresher algorithm. Similar to [8], each multi-owner transaction is associated with a chronological time stamp value. The principle of the multi-master refresher algorithm is to submit a sequence of multi-owner transactions in chronological order at each node. As a consequence, total order is enforced since multi-owner transactions are executed in same serial order at each node. To assure chronological ordering, before submitting a multi-owner transaction at node  $i$ , the algorithm has to check whether there is any older committed multi-owner transaction enroute to node  $i$ . To accomplish this, the submission time of a new multi-owner transaction  $MOT_i$  is delayed by  $Max + \epsilon$  (recall that  $Max$  is the upper bound of the time needed to multicast a message from a node to any other node). After this delay period, all older transactions are guaranteed to be received at node  $i$ . Thus chronological and total orderings are assured.

However, different from [8], for the multi-master configuration, whenever a multi-owner transaction  $MOT_i$  is to be triggered at some node  $i$ , the same node multicasts  $MOT_i$  to all nodes  $1, 2, \dots, n$ , of the multi-owner configuration, including itself. Once  $MOT_i$  is received at some other node  $j$  (notice that  $i$  may be equal to  $j$ ), it is placed in the *pending queue* for the multi-owner triggering node  $i$ , called multi-owner pending queue (noted  $moq_i$ ). Therefore, at each multi-owner node  $i$ , there is a set of multi-owner queues,  $moq_1, moq_2, \dots, moq_n$ . Each pending queue corresponds to a node of the multi-owner configuration and is used by the refresher to perform chronological ordering.

To implement the multi-master refresher algorithm in a multi-owner node we assume that six components are added to a regular DBMS in order to support lazy replication. Our goal is to maintain node autonomy since the implementation solution does not require the knowledge of system internals. We also assume that it is known whether users write on multi-owner copies. Figure 2 shows all architectural components necessary to implement our algorithm. The *Replica Interface* manages the incoming multi-owner transaction submission. The *Receiver* and *Propagator* implement reception and propagation of messages, respectively. The *Refresher* implements the multi-master refreshment algorithm. Finally, the *Deliverer* manages the submission of multi-owner transactions to the local transaction manager in FIFO order using a *running queue*. In the following we focus on the *Replica Interface* and *Refresher* components.

Multi-owner transactions are submitted through the *Replica Interface*. The application program calls the *Replica Interface* passing as parameter the multi-owner transaction  $MOT$ . The *Replica Interface* then establishes a timestamp value  $C$  for  $MOT$ , that corresponds to  $MOT$  start submission time. Afterwards, the sequence of operations of  $MOT$  is written in the *Owner Log* followed by  $C$ . Whenever the multi-

owner transaction commits, the *Deliverer* notifies the Replica Interface. Meanwhile, the Replica Interface waits for *MOT* commitment.

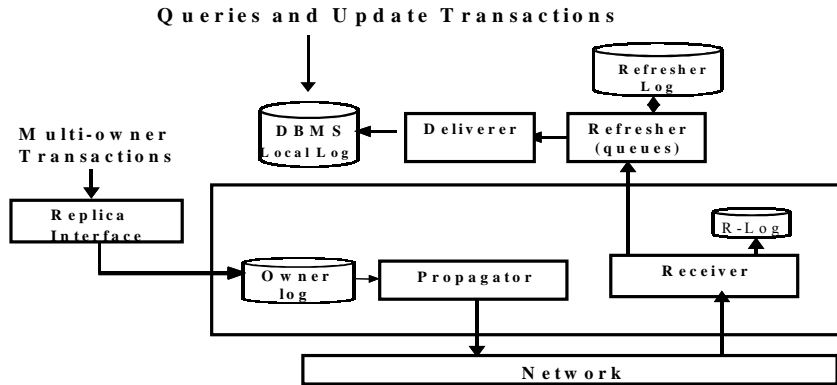


Fig. 2. Multi-owner node Architecture

The Refresher continuously reads the contents of the multi-owner pending queues. Each pending queue  $i$  stores a sequence of multi-owner transactions in multicast order. Therefore the contents of these pending queues form the input for the refresher algorithm. Whenever a *MOT* is selected for submission the refresher inserts it into the *running queue*, which contains all ordered transactions that are not yet entirely executed. The Deliverer then manages the submission of multi-owner transactions to the local transaction manager in the order in which they were inserted into the running queue.

#### 4 Validation

To validate our solution to multi-master refreshment, we implemented the refresher algorithm and architecture in a cluster system. We use this implementation to study the refresher's behavior in terms of *scalability* [10] and *data freshness*. In this section, we briefly describe our implementation and present our performance model and experimentation results.

We did our implementation on a cluster of 5 nodes, where one node is used exclusively as the access point to the cluster. Each node is configured with a 2GHz Pentium 4 processor, 512 MB of memory and 40GB of disk. The nodes are linked by a 1 Gb network. We use Linux Mandrake 8.0/Java and Spread toolkit that provides a reliable FIFO message bus and high performance message service among the cluster nodes that is resilient to faults. We use Oracle8i as the underlying database system at each node. In our implementation we consider four modules: *Client*, *Replicator*, *Network* and *Database Server*. The Client module simulates the clients of the ASP. It submits multi-owner transactions randomly to any cluster node, via RMI-JDBC, which implement the Replica Interface. Each cluster node hosts a Database Server and at least one instance of the Replicator module. The Replicator module implements

all system components necessary for a multi-owner node: Replica Interface, Propagator, Receiver, Refresher and Deliverer. Notice that each time a transaction is to be executed it is first sent to the Replica Interface which checks whether the incoming transaction writes on replicated data. Whenever a transaction does not write on replicated data it is sent directly to the local transaction manager. Even though we do not consider node failures in our performance evaluation, we implement all the necessary logs for recovery to understand the complete behaviour of the algorithm. The Network module interconnects all cluster nodes through the Spread toolkit.

Our performance model takes into account multi-owner transaction size, arrival rates and the number of multi-owner nodes. We vary multi-owner transactions' arrival rates (noted  $\lambda$ ). We define two types of multi-owner transactions (denoted by  $|MOT|$ ), defined by the number of write operations. Small transactions have size 5, while long transactions have size 50. To understand the behaviour of our algorithm in the presence of short and long transactions, we define four scenarios. Each scenario is defined in terms of a parameter called *long transaction ratio* (denoted by  $ltr$ ). We set  $ltr$  as follows:  $ltr = 0$  (all update transactions are short),  $ltr = 30$  (30% of update transactions are long),  $ltr = 60$  (60% of update transactions are long) and  $ltr = 100$  (all update transactions are long). Updates are done on the same attribute of a different tuple.  $Nb-Inst$  defines the number of multi-owner instances  $i$  (Replicator instances) on the same node. That is, since our cluster has 4 nodes, we let the same cluster node execute more than one Replicator instance to be able to simulate up to 8 multi-owner nodes. Hence,  $Nb-Multi-Owner$  defines the number of multi-owner nodes. To simulate 8 multi-owners, each cluster node carries two Replicator instances. Each node may execute at most 2 multi-owner node instances because the delay introduced by memory management may compromise our results. For fairness, we always consider an equal number of multi-owner instances at each node. Finally our single table schema has two attributes and carry 100000 tuples and no indexes. The parameters of the performance model are described in Table 1.

<b>Param.</b>	<b>Definition</b>	<b>Values</b>
$ MOT $	Transaction size	5; 50
$Ltr$	Long transaction ratio	0; 30; 60; 100%
$Nb-Multi-Owner$	Number of Replicators	2; 3; 4; 8
$Nb-Inst$	Number of Replicators on 1 node	1, 2
$\lambda$	Mean time between transactions (workload)	bursty: 200ms; low: 20s
$M$	Time of the test	60000ms; 600000ms
$Max + \epsilon$	Submission time of a multi-owner transaction is delayed	100ms

Table 1. Performance parameters

We describe two experiments to study scalability and freshness. The first experiment studies the algorithm's scalability. That is, for a same set of incoming multi-owner transactions, scalability is achieved whenever in augmenting the number of nodes the response times remain the same. We consider *bursty* and *low* workloads ( $\lambda = 200ms$  and  $\lambda = 20s$ ), varying the number of multi-master nodes (2, 4 and 8). In addition, we vary the ratio of long transactions ( $ltr = 0/30/60/100$ ). Finally, we perform this experiment during a time interval of 60000ms.

This experiment results (see Figure 3.a and 3.b) clearly shows that for all *ltr* values scalability is achieved. In fact, for each *ltr* value the response times<sup>5</sup> correspond to the average transaction response time. We can clearly see in Figure 3 that the difference in response times is a function of the *long transaction ratio* (*ltr*). The higher the *ltr* value, the higher are the response times, because, due to the size of transactions and the *bursty* (Figure 3.a) workload, the running queue (recall that the running queue is used by the Deliverer to submit multi-owner transactions), becomes a bottleneck for transaction processing, since transaction submission is done sequentially. As a consequence the average response times are much higher compared to the low workload scenario (Figure 3.b). However, the algorithm still scales-up since the network delay introduced to multicast a message is very small, avoiding network contention. Furthermore, in contrast with synchronous algorithms, our algorithm has linear response time behaviour since it requires only the multicast of  $n$  messages for each incoming multi-owner transaction, where  $n$  corresponds to the number of nodes.

The second experiment analyzes the degree of freshness. Suppose  $\Pi = \{i, j, k, \dots\}$  is a set of multi-owner nodes of  $R$ <sup>6</sup>. Intuitively, the degree of freshness of  $R$  on node  $i$  over  $\Pi$  is defined as the difference between the number of committed multi-owner transactions on  $R$  at node  $i$  and the average of number of committed multi-owner transactions on others multi-owner nodes ( $\Pi - i$ ).

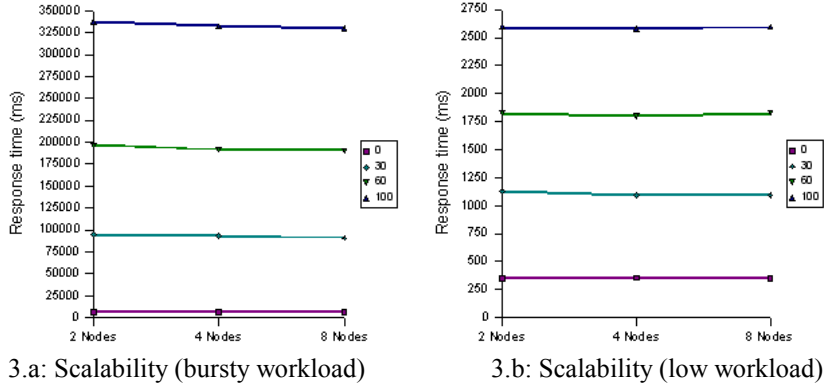


Fig. 3. Performance Results

Thus, the best degree of freshness is 1, which means that multi-owner copies are *mutually consistent*. Recall that the mutual consistency property is assured by eager algorithms. On the other hand, degrees of freshness close to zero express the fact that there are a significant number of updates committed at some multi-owner copy that were still not performed in others copies.

We consider *bursty* workloads ( $\lambda = 200\text{ms}$ ) varying the number of multi-owner nodes ( $Nb\text{-Multi-Owner} = 2/4/8$ ) and the long transactions ratio ( $ltr = 0/30/60/100$ ). Finally, we perform this experiment during a time interval of 60000ms. The freshness

<sup>5</sup> Time spent to execute a transaction submitted by a client (involves the Multi-Master Refresher Algorithm).

<sup>6</sup> Each node holds a multi-owner copy  $R$ .

degrees for the 4 scenarios are almost equal (between 0.98 and 0.99). In all cases very good degrees of freshness are attained (very close to 1). These results clearly demonstrate that the time delay introduced to submit transactions at each node is the same, because all nodes executed the same sequence of multi-owner transactions. As a result, the transaction processing speed at each node is uniform, independent of the number of nodes and the ratio of long transactions. Notice however that we do not consider query loads. Nevertheless, we can safely assume that query loads do not impact freshness degrees whenever the underlying database server does not use strict two-phase-locking concurrency protocol [9]. That is, queries do not block multi-owner transactions. This experiment shows that our algorithm, in a cluster environment, does not introduce any significant loss of data freshness. In fact, it almost provides mutual consistency.

## 5 Related Work

There are several interesting projects that deal with replicated data management in cluster architectures: PowerDB [2], GMS [11], and TACT [13]. However, none of them employ asynchronous multi-master replication as we do. A general framework for optimistic and preventive data replication for the same ASP cluster system is presented in [1]. As an evolution of [1], we propose, in this paper, a cluster architecture that enables user requests to be managed completely asynchronously in order to avoid any type of bottleneck. In addition, we detail the preventive refreshment management, showing the implementation environment and some performance results. In the following, we compare our replication solution with other existing solutions.

With synchronous (or eager) replication, the property of mutual consistency is assured. A solution proposed in [5] reduces the number of messages exchanged to commit transactions compared to 2PC, but the protocol is still blocking and it is not clear if it scales up. It uses, as we do, communication services [6] to guarantee that messages are delivered at each node according to some ordering criteria.

Recently, [8] proposed a refreshment algorithm that assures strong consistency for lazy master-based configurations. They do not consider the multi-master configuration as we do. Our solution uses the same principle of their refreshment algorithm. However, in our work, we show how it can be employed in a multi-master configuration.

## 6 Conclusion

In this paper, we proposed a preventive lazy multi-master replication database solution in a cluster system for ASP management. In this context, data replication is used to improve data availability and query load balancing (and thus performance). In this paper we first proposed a cluster architecture that enables users' requests to be managed completely asynchronously in order to avoid any type of bottleneck. Second, we proposed a multi-master refresher algorithm that *prevents* conflicts, by exploiting

the cluster's high speed network, thus providing strong consistency, without the constraints of eager replication. Cluster nodes can support autonomous, heterogeneous databases that are considered as black boxes. Thus our solution achieves full node autonomy. We also showed the system architecture components necessary to implement the refresher algorithm. Finally, we described our experiments over a cluster of 8 nodes. In our experimental results, we showed that the multi-master algorithm scales-up and introduces a negligible loss of data freshness (almost equal to mutual consistency).

For future work, we will propose some optimizations for the refresher algorithm in managing multi-owner transactions on the running queue. For instance, for *bursty* workloads one can check whether ordered transactions in the running queue do not conflict. If there is no conflict, these transactions can be triggered concurrently. In addition, we plan to consider other types of multi-master configurations.

## References

1. S. Gançarski, H. Naacke, E. Pacitti, P. Valduriez: Load Balancing of Autonomous Applications and Databases in a Cluster System, Parallel Processing with Autonomous Databases in a Cluster System, *Int. Conf. of Cooperative Information Systems (CoopIS)*, 2002.
2. T. Grabs, K. Bohm, H. Scheck: Scalable Distributed Query and Update Service Implementation for XML Documents Elements, *IEEE RIDE Int. Workshop on Document Management of Data Intensive Business and Scientific Applications*, 2001.
3. L. George and P. Minet: A FIFO worst analysis for a hard real time distributed problem with consistency constraints, *Int. Conf. On Distributed Computing Systems, Baltimore (ICDCS97)*, 1997
4. 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, 1996*.
5. B. Kemme, G. Alonso: Don't be lazy be consistent: Postgres-R, a new way to implement Database Replication, *Int. Conf. on Very Large Databases (VLDB)*, 2000.
6. V. Hadzilacos and S. Toueg: A Modular Approach to Fault-Tolerant Broadcasts and Related Problems, *Technical Report TR 94-1425, Dept. of Computer Science, Cornell University, Ithaca, NY 14853*, 1994.
7. E. Pacitti, P. Valduriez: Replicated Databases: concepts, architectures and techniques, *Network and Information Systems Journal, Hermès*, 1(3), 1998.
8. E. Pacitti, P. Minet, E. Simon: "Replica Consistency in Lazy Master Replicated Databases". *Distributed and Parallel Databases, Kluwer Academic*, 9(3), May 2001.
9. T. Özsu, P. Valduriez: *Principles of Distributed Database Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 2nd edition, 1999.
10. P. Valduriez: Parallel Database Systems: open problems and new issues. *Int. Journal on Distributed and Parallel Databases*, 1(2), 1993.
11. G. M. Voelker et. al: Implementing Cooperative Prefetching and Caching in a Globally-Managed Memory System, *In ACM Sigmetrics Conf. on Performance Measurement, Modeling, and Evaluation*, 1998.
12. U. Röhm, K. Böhm, H.-J. Schek. Cache-Aware Query Routing in a Cluster of Databases. *Int. Conf. on Data Engineering (ICDE)*, 2001.
13. H. Yu, A. Vahdat: Efficient Numerical Error Bounding for Replicated Network Service, *Int. Conf. on Very Large Databases (VLDB)*, 2000.