Fine-grained Refresh Strategies for Managing Replication in Database Clusters

Stéphane Gançarski Cécile Le Pape Hubert Naacke

Laboratoire d'Informatique de Paris 6, Paris, France
email : Firstname.Lastname@lip6.fr

Abstract

Relaxing replica freshness has been exploited in database clusters to optimize load balancing. In this paper, we propose to support both routing-dependant and routing-independent refresh strategies in a database cluster with multi-master lazy replication. First, we propose a model for capturing refresh strategies. Second, we describe the support of this model in a middleware architecture for freshness-aware routing in database clusters. Third, we describe an algorithm for computing *refresh graphs*, which are the core of all the refresh strategies.
Keywords: replication, database cluster, load balancing, refresh strategy.

1 Introduction

Database clusters provide a cost-effective alternative to parallel database systems, *i.e.* database systems on tightly-coupled multiprocessors. A database cluster [10, 9, 26, 27] is a cluster of PC servers, each running an off-the-shelf ("black-box") DBMS and holding a (partial) replica of the database. Since the DBMS source code is not necessarily available and cannot be changed to be "cluster-aware", parallel database system capabilities such as load balancing must be implemented via middleware.

Managing replication in database clusters has recently received much attention. As in distributed databases, replication can be eager (also called synchronous) or lazy (also called asynchronous). With eager replication, a transaction updates all replicas, thereby enforcing the mutual consistency of the replicas. By exploiting efficient group communication services provided by a cluster, eager replication can be made non blocking (unlike with distributed transactions) and scale up to large cluster sizes [14, 15, 25, 13]. With lazy replication, a transaction updates only one replica and the other replicas are updated (refreshed) later on by separate refresh transactions [22, 23].

With lazy replication, two different problems may occur. First, replicas may diverge if the same data is updated simultaneously in two different nodes. This is the well known problem of *replica control* which must enforce eventual consistency : if updates stop, replicas must eventually converge to the same state. Second, a (read-only) query executed on a replica which has not been synchronized yet may read inconsistent and/or stale data. We call this the *query control* problem. Different strategies have been proposed to solve this problem : wait until data become consistent and/or fresh, or accept to read "almost consistent/fresh" data [11, 17, 27, 1, 20, 30, 3]. The client specify its consistency/freshness requirements while the system guarantees them with an adequate update propagation strategy. Little work has been done to consider these two problems together. We think that performances can greatly benefit from controlling replica and queries simultaneously, thanks to a uniform load balancing. In our approach, replicas are controlled in a preventive way : transactions which perform updates are propagated with respect to a *global transaction ordering graph* (*TOG*). Conflicts are prevented because updates are executed on all nodes in compatible orders. A transaction is executed on a node only when all transactions preceding it have been already executed on the node. Queries are not distributed thus they always read consistent states, though maybe stale. The problem of query control reduces to controlling the replica freshness, which reflects the distance between the state of the replica and the most recent state of the corresponding data. In this context, we treat both replica and query control uniformly as a refresh problem, which can be stated as follows : given an dababase cluster state and a *request* (transaction or query), evaluate for each node which transactions should be propagated before routing the request to the node such that (1) no unnecessary transaction is propagated, (2) the local execution order is compatible with the global TOG, (3) the results satisfy the freshness requirements of the request, and (4) the choice of the node minimizes the request response time.

We make the distinction between *routing-dependent*

and *routing-independent* refresh strategies. The routing-dependant strategy (or On-demand) works as follows: if the load balancer selects an underloaded node that is not fresh enough for an incoming request, it first sends refresh transactions to that node before sending the query. It is not sufficient since the freshness level of some nodes may get lower and lower, thus increasing the cost of refreshment. Thus, we add routing-independant refresh strategies, that are triggered based on events other than routing, *e.g.* when a node is too stale, idle or little busy, or after some time from the last refreshment. There are several possible refresh strategies, according to the application workload. For instance, if the workload is update-intensive and if queries are rare and require a perfect freshness, then it is better to refresh nodes frequently, *e.g.* as soon as possible, in order to take advantage of periods when nodes are query-free. On the contrary, when the workload is query intensive but queries do not require high freshness, it is better to refresh only when necessary, in order to not overload nodes with unnecessary refreshment.

Many refresh strategies have been proposed in the context of distributed databases, data warehouse and and database clusters. A popular strategy is to propagate updates from the source to the copies as soon as possible (ASAP), as in [4, 5, 7]. Another simple strategy is to refresh replicas periodically [6, 19] as in data warehouses [8]. Another strategy is to maintain the freshness level of replicas, by propagating updates only when a replica is too stale [29]. There are also mixed strategies. In [21], data sources push updates to cache nodes when their freshness is too low. However, cache nodes can also force refreshment if needed. In [17], an asynchronous Web cache maintains materialized views with an ASAP strategy while regular views are regenerated on demand. Refreshment in [27] is interleaved with query scheduling which makes difficult to analyze the impact of the refresh strategy itself. In all these approaches, refresh strategies are not chosen to be optimal with respect to the workload. In particular, refreshment cost is not taken into account in the routing strategy. There has been very few studies of refresh strategies and they are incomplete (ex. [12]). For instance, they do not take into account the starting time of update propagation [28, 16] or only consider variations of ASAP [24].

This paper has three main contributions, which clearly distinguish it from our previous work [18]. First, we propose a model which allows describing and implementing refresh strategies, independent of other load balancing issues. Second, we describe the support of this model in our prototype. It is based on the concept of *refresh graph* whose execution brings a node to a required level of freshness while guaranteeing global serializability. For transactions, it brings the node to a perfectly fresh state, in order to be compatible with the global order. For queries, it brings the node to the required level of freshness specified by the application. Routing independant strategies are also described through refresh graphs, one for each node involved in the strategy. Third, we give an algorithm that computes minimal refresh graphs with respect to a given freshness requirement. In comparison, [18] is based on a mono-master replication, based on refresh sequences, while this paper presents a multi-master replication scheme based on refresh graphs. Furthermore, our previous work had only a routing-dependant refresh strategy while here we added routing-independant refresh strategies.

The paper is organized as follows. Section 2 describes our database cluster architecture, with emphasis on load balancing and refreshment. Section 3 defines our model to describe refresh strategies. Section 4 describes the algorithm for computing refresh graphs. Section 6 concludes.

2 Database Cluster Architecture

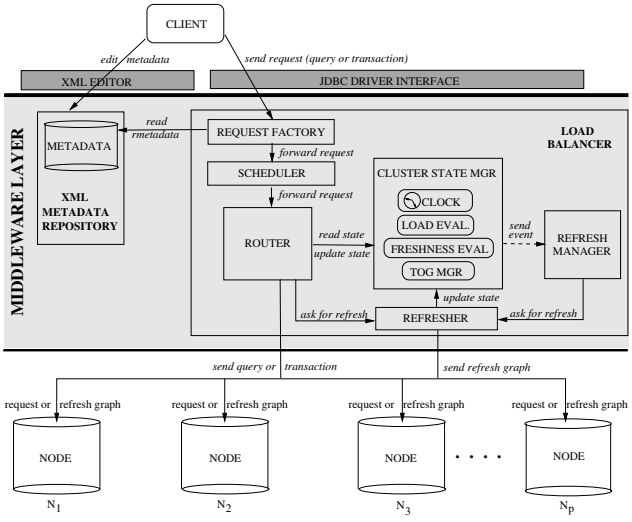

Figure 1: Multi-master replicated database architecture

Figure 1 gives an overview of our multi-master database cluster middleware. We assume that the database composed of relations $R^1, R^2, \ldots, R^k$ is fully replicated on nodes $N_1, N_2, \ldots, N_p$. We note R^i_j the replica of relation R^i on node N_j. Our middleware receives *requests* (transactions or queries) from the applications through a standard JDBC interface. All additional information necessary for routing anf refreshing is stored in a metadata repository and managed separately of the requests. The metadata repository includes for instance the default level of freshness required by a query. It also includes information about which part of the database is read and which part is updated by the requests, thus enabling the detection of

potential conflicts between updates and queries. Our architecture preserves the autonomy of both applications and databases which can remain unchanged, as required in ASP [9] for instance.

To support general applications such as ASP, the access to the database is through stored procedures : clients requests are procedure calls. The request factory enriches requests with metadata obtained from the repository and dynamic information provided by the clients (*e.g.* parameters for stored procedures). Then it sends the requests to the scheduler.

As we focus on refresh policies, we use here a simple FIFO scheduling : requests are sent to the router in the same order they arrive to the scheduler.

Dynamic information such as transaction commit time on nodes, data freshness on nodes, estimated nodes load, is maintained by the cluster state manager. The information related to each transaction is maintained until it has been executed on every node, after which it is removed. It also maintains a *transaction ordering graph (TOG)*. Intuitively, a transaction T precedes a transaction T' in the TOG if T' arrives in the system when T is currently running and T potentially conflicts with T'. The load evaluation module evaluates the nodes' load by the summing the remaining execution times of running requests on nodes. The execution time of a request is estimated through a weighted mean algorithm based on the previous executions of the request.

The router implements an enhanced version of SELF (Shortest Execution Length First) [2], which includes the estimated cost of refreshing a node on-demand. Upon receiving a request, it computes a cost function for every node and selects for execution the node which minimizes the cost function, thus minimizing the request response time. The cost of executing a request on a node is composed of the node's load plus the cost of preparing the node for executing the request. Preparing a node consists in executing a *refresh graph* on the node prior to request execution. The refresh graph is a minimal subgraph (in the sense of inclusion) of the TOG which, when applied to the node, makes it fresh enough for the request (perfectly fresh if the request is a transaction). Transactions in the refresh graph are executed on the node according to the refresh graph (partial) order.

Executing the refresh graph for a request is called *routing-dependent (on-demand) refreshment*. On the other side, the refresh manager handles *routing-independent refreshment*. According to the refresh strategy, it receives events coming from different part of the cluster state manager: load evaluation module, freshness evaluation module or external events such as time. It then triggers the selected routing-independent refresh policy which eventually asks the refresher module to perform refresh graphs. Building a refresh graph depends on the nature of the refresh strategy. The alogrithm that performs this task is presented in Section 4 . Whenever the refresher sends refresh graphs to a node, it updates the cluster state for further freshness evaluations by the corresponding module.

3 Modeling Refresh Strategies

In this section, we propose a model for defining various refresh strategies. It can be used as a basis to help a DBA specifying when, to which nodes, and how much to refresh. The refresh model is based on a freshness model which allows measuring the staleness of a slave node with respect to the master node.

3.1 Conflicts detection

We detect conflicts based only on procedure codes, *i.e.* the procedure code is known in advance. Thus we detect *potential conflicts*, at the relation level, because relations potentially read or written by a request can be easily infered from the procecure code. Each request *req* is associated to the set of relation its potentially reads (resp. writes), called *req.read* (resp. *req.write*). A query Q potentially conflicts with a transaction T if T potentially writes a data that Q potentially reads. A transaction T_i potentially conflicts with another transaction T_j if T_i potentially writes a data that T_j potentially reads or writes. Potential conflict detection is more formally described in [18].

3.2 Freshness Model

In [18] we introduced several freshness measures. For simplicity in this paper, we consider only measure *Age*. $Age(R_j^i)$ denotes the maximum time since at least one transaction updating R^i has committed on a node and has not yet been propagated on node N_j, *i.e.*

$$Age(R_j^i) = \begin{cases} Max(now() - T.ct),\ T \in U(R_j^i) \\ 0 \text{ if } U(R_j^i) = \emptyset \end{cases}$$

where $U(R_j^i)$ is the set of transactions updating R^i and not yet propagated to node N_j and $T.ct$ is the commit time of T on the first node it has validated. Measure *Age* allows modelling queries such as "Give the value of X as it was no later than Y minutes ago". It is also useful for queries accessing history relations. Other freshness measures defined in [18] can be used, according to applications needs and can even be combined.

The *freshness level of a request Req* is a conjunction of conditions of the form $Age(R^i) < th_i$, for each $R^i \in Req.write \cup Req.read$, where th_i is the maximum age (threshold) of R^i tolerated by *Req*. The default value of th_i is 0 for both queries or transactions (they must access perfectly fresh relations). If *Req* is a query, th_i can be overwritten by the user in order to increase the tolerated freshness. In all cases, a node N_j is fresh enough to satisfy *Req* if the freshness level of *Req* is satisfied on N_j. The freshness level of *Req* is stored in

the vector $Req.FL[1..k]$, such that $Req.FL[i] = th_i$ if R^i is accessed by Req, and ∞ otherwise.

3.3 Refresh Model

Refresh Strategy ::=	({Event}, Dest. , Quantity)
Event ::=	*Routing*(N_j,*Req*)
	| *Underloaded*(N_j,*limit_load*)
	| *Stale*(N_j,R^i, *limit_age*)
	| *Trans_commit*(N_j,*T*)
	| *Period*(*t*)
Dest. ::=	{ node }
Quantity ::=	$Age[1..k]$

Figure 2: Refresh model

We propose to capture refresh strategies with the model in Figure 2. A refresh strategy is described by the *triggering events* which raise its activation, the nodes where the refresh transactions are propagated and the quantity of refreshment to do. A refresh strategy may handle one or more triggering events, among:

- *Routing*(N_j, Req) : a request Req is routed to node N_j.
- *Underloaded*(N_j,*limit_load*): the load of node N_j decreases below the *limit_load* value.
- *Stale*(N_j,R^i, *limit_age*) : the age of R^i_j increases above *limit_age* value. In other words, the freshness atom $Age(R^i) < limit\_age$ is no more satisfied on node N_j.
- *Trans_commit*(N_j,*T*) : transaction T has commited on node N_j.
- *Period*(*t*) : triggers every t seconds.

As soon as an event handled by the refresh manager is raised, the refresher computes a refresh graph to propagate. The refresh graph can be sent to one or several nodes. For instance, *Routing*(N_j, Req) usually activates a refreshment only on node N_j while *Period*(*t*) usually activates a refreshment on all the nodes.

Finally, the refresh quantity of a strategy indicates "how many" refresh transactions are part of the refresh graph for each node to refresh. The $Age[1..k]$ vector expresses for each R^i, the age that must not be overpassed after applying the refresh graph. The refresh graph is thus a minimal subgraph (in the sense of inclusion) to make the node fresh enough with respect to $Age[1..k]$. Note that the default value for $Age[i]$ is ∞.

We apply our refresh model to the following strategies. Other strategies are possible, we give here some examples inspired from the state-of-art strategies.

3.3.1 On-Demand.

The On-Demand strategy is triggered by a *Routing*(N_j, Req) event. It sends a minimal refresh graph to node N_j to make it fresh enough for Req, *i.e.* $Age[1..k] = Req.FL$.

3.3.2 ASAP

The ASAP (As Soon As Possible) strategy is triggered by a *trans_commit*(N_j,*T*) event. It sends a refresh graph to all the nodes where T has not been sent yet. As ASAP strategy maintains nodes perfectly fresh, the refresh is specified with $Age[i] = 0, \forall i \ s.t. \ R^i \in T.write \cup T.read$.

3.3.3 Periodic(t,R^i, limit_age)

The Periodic strategy is triggered by a *period*(t) event. It sends refresh graphs to all nodes to keep the staleness of R^i under the *limit_age* value. Thus, the refresh graph for N_j is defined by $Age[i] = limit\_age$. If $limit\_age = 0$, then the strategy brings R^i to a perfect freshness on every node.

3.3.4 ASAUL(limit_load, limit_age)

The ASAUL (As Soon As underloaded) strategy is triggered by a *Underloaded*(N_j,*limit_load*) event. It sends a refresh graph to N_j to bring the staleness of all the relations replica on N_j under a *limit_age* value. Thus, the refresh graph for N_j is defined by $Age[i] = limit\_age, \forall i = 1 \dots k$

3.3.5 ASATS(limit_age)

The ASATS (As Soon As Too Stale) strategy is triggered by a *Stale*(N_j,R^i, *limit_age*) event. It sends a refresh graph to N_j to make the local copy R^i_j perfectly fresh, *i.e.* $Age[i] = 0$.

3.3.6 Hybrid Strategies.

Refresh strategies can be combined to improve performance. For instance, the interaction between routing-dependent (On-Demand) and routing-independent strategies (all other strategies) allows using any node for executing any request, since On-Demand always refreshes the node where the request is routed before sending the request. Another example is to create different periodic strategies for different relations with different periods, allowing to associate a smaller period for "hot-spot" relations and a higher for rarely requested relations.

4 Computing refresh graphs

In this section, we present our method to compute refresh graphs. We first introduce the data structures, and present the algorithms.

4.1 Data structures and auxiliary functions

- $TOG = (\{T\}, \prec)$ is the transaction ordering graph, defined as a set of transaction $\{T\}$ and a partial order $\prec$. TOG is obviously acyclic since its order is compatible with the transaction arrival time.
- Each transaction T is associated with attributes $T.ct$ (commit time, see Section 3.2) and $T.write$, (set of relations potentially written by T, see Section 3.1).
- Each node N_j is associated with an attribute $N_j.yet$ (Youngest Executed Transactions) which is the set of the youngest (w.r.t. $\prec$) transactions already executed on N_j. Attribute $N_j.yet$ can be seen as the current freshness state of node N_j.
- Function *Leaves()* returns the leaves of acyclic graph *TOG*.
- Function *Parents(T)* returns the parents of T in the TOG.
- Vector $Age[1..k]$ is the specification of the refresh graph to compute (see Section 3.3).

4.2 Algorithm

The algorithm is shown on Figure 3. Function *Refresh_graph(Age[1..k],N_j)* computes a minimal refresh graph for refreshing node N_j in order to fulfill freshness requirements specified by vector *Age[1..k]*. The main idea is, starting from the leaves of the TOG, to recursively include in the refresh graph all the necessary transactions, detected by function *Necessary(T, $Age[1..k]$)*. The process stops when reaching transactions already executed on N_j, *i.e.* transactions belonging to $N_j.yet$. For sake of simplicity, we do not handle individual precedences among transactions, they can be deduced from the $\prec$ precedence order. The age of data on node N_j is computed based on the cluster current time when the algorithm starts.

```
Function Refresh_graph(Age[1..k],N_j)
  Tset := ∅
  t = now()
  for all T ∈ Leaves() do
    Tset := Tset ∪ Refresh_set(T,Age[1..k],N_j,t)
  end for
  return (Tset,≺)
Function Refresh_set(T,Age[1..k],N_j,t)
  Nset := ∅
  if T ∈ N_j.yet then
    return ∅
  end if
  if Necessary(T,Age[1..k],t) then
    Nset = {T}
  end if
  for all T' in Parents(T) do
    Nset := Nset ∪ Refresh_set(T,Age[1..k],N_j,t)
  end for
  return Nset
Function Necessary(T, Age[1..k],t)
  for all R_i ∈ T.write do
    if Age[i] ≤ (t − T.ct) then
      return true
    end if
  end for
  return false
```

Figure 3: The algorithm for computing refresh graphs

5 Conclusion

In this paper, we proposed a refresh model that allows capturing state-of-the-art refresh strategies in a database cluster with multi-master lazy replication. We distinguish between the routing-dependent (or on-demand) strategy, which is triggered by the router, and routing-independent strategies, which are triggered by other events, based on time-outs or on nodes state. The on-demand strategy serves for both query routing, according to query freshness requirement, and transaction routing, in order to guarantee global serializability based on a global transaction order graph. The output of any refresh strategy is a refresh graph to be executed, for each target node. The refresh model allows for specifying the refresh graph to execute in a simple way, and we give the algorithm which produces a refresh graph based on its specification. The refresh manager is independent of other load balancing functions such as routing and scheduling. We are currently testing the prototype to optimize the implementation of the Refresh_graph algorithm. We plan to run it with different workload types in order to determine the best strategy to select with respect to the workload.

References

[1] R. Alonso, D. Barbará, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Trans. on Database Systems*, 15(3):359–384, 1990.

[2] C. Amza, A. Cox, and W. Zwaenepoel. Conflict-aware scheduling for dynamic content applications. In *Proceedings of the Fifth USENIX Symposium on Internet Technologies and Systems, March 2003*, 2003.

[3] D. Barbará and H. Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *VLDB Journal*, 3(3):325–353, 1994.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ansi isolation levels. In *ACM SIGMOD Int. Conf.*, 1995.

[5] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databates. In *ACM SIGMOD Int. Conf.*, pages 97–108, 1999.

[6] D. Carney, S. Lee, and S. Zdonik. Scalable application aware data freshening. In *IEEE Int. Conf. on Data Engineering*, 2002.

[7] P. Chundi, D. J. Rosenkrantz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *IEEE Int. Conf. on Data Engineering*, pages 469–476, 1996.

[8] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *ACM SIGMOD Int. Conf.*, pages 469–480, 1996.

[9] 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.

[10] S. Gançarski, H. Naacke, and P. Valduriez. Load balancing of autonomous applications and databases in a cluster system. In *Workshop on Distributed Data and Structures (WDAS)*, 2002.

[11] H. Guo, P.-A. Larson, R. Ramakrishnan, and J. Goldstein. Relaxed currency and consistency: How to say "good enough" in sql. In *ACM SIGMOD Int. Conf.*, 2004.

[12] Y. Huang, R. H. Sloan, and O. Wolfson. Divergence caching in client server architectures. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, September 28-30, 1994*, pages 131–139. IEEE Computer Society, 1994.

[13] R. Jiménez-Peris, M. Patino-Martinez, B. Kemme, and G. Alonso. Are quorums an alternative for database replication. *ACM Trans. on Database Systems*, 28(3):257–294, 2003.

[14] 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 Data Bases*, pages 134–143, 2000.

[15] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Trans. on Database Systems*, 25(3):333–379, 2000.

[16] S. Krishnamurthy, W. H. Sanders, and M. Cukier. An adaptive framework for tunable consistency and timeliness using replication. In *Int. Conf. on Dependable Systems and Networks*, pages 17–26, 2002.

[17] A. Labrinidis and N. Roussopoulos. Balancing performance and data freshness in web database servers. In *Int. Conf. on Very Large Data Bases*, pages 393–404, 2003.

[18] C. Le Pape, S. Gançarski, and P. Valduriez. Refresco: Improving query performance through freshness control in a database cluster. In *Int. Conf. On Cooperative Information Systems (CoopIS)*, pages 174–193, 2004.

[19] H. Liu, W.-K. Ng, and E.-P. Lim. Scheduling queries to improve the freshness of a website. *World Wide Web*, 8(1):61–90, 2005.

[20] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Int. Conf. on Very Large Data Bases*, 2000.

[21] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Int. Conf. on Very Large Data Bases*, 2000.

[22] E. Pacitti, P. Minet, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Int. Conf. on Very Large Data Bases*, 1999.

[23] E. Pacitti, P. Minet, and E. Simon. Replica consistency in lazy master replicated databases. *Distributed and Parallel Databases*, 9(3):237–267, 2000.

[24] E. Pacitti and E. Simon. Update propagation strategies to improve freshness in lazy master replicated databases. *VLDB Journal*, 8(3–4):305–318, 2000.

[25] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme, and G. Alonso. Scalable Replication in Database Clusters. In *Int. Conf. on Distributed Computing (DISC'00)*, pages 315–329, 2000.

[26] U. Röhm, K. Böhm, and H.-J. Schek. Cache-aware query routing in a cluster of databases. In *IEEE Int. Conf. on Data Engineering*, 2001.

[27] U. Röhm, K. Böhm, H.-J. Schek, and H. Schuldt. Fas - a freshness-sensitive coordination middleware for a cluster of olap components. In *Int. Conf. on Very Large Data Bases*, 2002.

[28] Y. Saito and H. M. Levy. Optimistic replication for internet data services. In *Int. Symp. on Distributed Computing*, pages 297–314, 2000.

[29] S. Shah, K. Ramamritham, and P. Shenoy. Maintaining coherency of dynamic data in cooperative repositories. In *Int. Conf. on Very Large Data Bases*, 1995.

[30] H. Yu and A. Vahdat. Efficient numerical error bounding for replicated network services. In *Int. Conf. on Very Large Data Bases*, 2000.