Parallel Processing with Autonomous Databases in a Cluster System

Stéphane Gançarski[1], Hubert Naacke[1], Esther Pacitti[2], Patrick Valduriez[1]

[1] *LIP6, University Paris 6*
8, rue du Cap. Scott 75015 PARIS
FirstName.LastName@lip6.fr

[2] *Institut de Recherche en Informatique de Nantes*
Esther.Pacitti@irin.univ-nantes.fr

ABSTRACT*. We consider the use of a cluster system for Application Service Provider (ASP). In the ASP context, hosted applications and databases can be update-intensive and must remain autonomous. In this paper, we propose a new solution for parallel processing with autonomous databases, using a replicated database organization. The main idea is to allow the system administrator to control the tradeoff between database consistency and application performance. Application requirements are captured through execution rules stored in a shared directory. They are used (at run time) to allocate cluster nodes to user requests in a way that optimizes load balancing while satisfying application consistency requirements. We also propose a new preventive replication method and a transaction load balancing architecture which can trade-off consistency for performance using execution rules. Finally, we discuss the on-going implementation at LIP6 using a Linux cluster running Oracle 8i.*

KEYWORDS: *database, cluster architecture, transaction processing, load balancing, replication, consistency*

1. Introduction

Clusters of PC servers now provide a cheap alternative to tightly-coupled multiprocessors such as Symmetric Multiprocessor (SMP) or Non Uniform Memory Architecture (NUMA). They make new businesses like Application Service Provider (ASP) economically viable. In the ASP model, customers' applications and databases (including data and DBMS) are hosted at the provider site and need be available, typically through the Internet, as efficiently as if they were local to the customer site. Thus, the challenge for a provider is to fully exploit the cluster's parallelism and load balancing capabilities to obtain a good cost/performance ratio. The typical solution to obtain good load balancing in cluster architectures 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. Thisalso 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 Web sites such as 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. First, applications can be update-intensive. Second, applications and databases must remain autonomous so they can be subject to definition changes to accommodate customer requirements. Replicating databases at several nodes, so they can be accessed by different users through the same or different applications in parallel, can create consistency problems [14], [9]. For instance, two users at different nodes could generate conflicting updates to the same data, thereby producing an inconsistent database. This is because consistency control is done at each node through its local DBMS. There are two main solutions readily available to enforce global consistency. One is to use a transaction processing monitor to control the access to replicated data. However, this requires significant rewriting of the applications and may hurt transaction throughput. A more efficient solution is to use a parallel DBMS such as Oracle Rapid Application Cluster or DB2 Parallel Edition. Parallel DBMS typically provide a shared disk abstraction to the applications [20] so that parallelism can be automatically inferred. But this requires heavy migration to the parallel DBMS and hurts database autonomy.

Ideally, applications and databases should remain unchanged when moved to the provider site's cluster. In this paper, we propose a new solution for load balancing of autonomous applications and databases which addresses this requirement. This work is done in the context of the Leg@Net project[1] sponsored by the RNTL between LIP6, Prologue Software and ASPLine, whose objective is to demonstrate the viability of the ASP model for pharmacy applications in France. Our solution exploits a replicated database organization. The main idea is to allow the system

[1] see www.industrie.gouv.fr/rntl/AAP2001/Fiches_Resume/LEG@NET.htm

administrator to control the database consistency/performance tradeoff when placing applications and databases onto cluster nodes. Databases and applications can be replicated at multiple nodes to obtain good load balancing. Application requirements are captured (at compile time) through execution rules stored in a shared directory used (at run time) to allocate cluster nodes to user requests. Depending on the users' requirements, we can control database consistency at the cluster level. For instance, if an application is read-only or the required consistency is weak, then it is easy to execute multiple requests in parallel at different nodes. If, instead, an application is update-intensive and requires strong consistency (*e.g.* integrity constraints satisfaction), then an extreme solution is to run it at a single node and trade performance for consistency. Or, if we want both consistency and replication (*e.g.* for high availability), another extreme solution is synchronous replication with 2 phase commit (2PC) [9] for refreshing replicas. However, 2PC is both costly in terms of messages and blocking (failure of the coordinator cannot be terminated independently by the participants).

There are cases where copy consistency can be relaxed. With optimistic replication [12], transactions are locally committed and different replicas may get different values. Replica divergence remains until reconciliation. Meanwhile, the divergence must be controlled for at least two reasons. First, since synchronization consists in producing a single history from several diverging ones, the higher the divergence is, the more difficult the reconciliation. The second reason is that read-only applicationsdo not always require to read perfectly consistent data and may tolerate some inconsistency. In this case, inconsistency reflects a divergence between the value actually read and the value that should have been read in ACID mode. Non-isolated queries are also useful in non replicated environments (*e.g.* ANSI isolation levels and their critique [2]). Specification of inconsistency for queries has been widely studied in the literature, and may be divided in two dimensions, temporal and spatial [18]. An example of temporal dimension is found in quasi-copies [1], where a cached (image) copy may be read-accessed according to temporal conditions, such as an allowable delay between the last update of the copy and the last update of the master copy. The spatial dimension consists of allowing a given "quantity of changes" between the values read-accessed and the effective values stored at the same time. This quantity of changes, referred to as import-limit in epsilon transactions [23], may be for instance the number of data items changed, the number of updates performed or the absolute value of the update. In the continuous consistency model [24], both temporal dimension (staleness) and spatial dimension (numerical error and order error) are controlled. Each node propagates its writes by either pull or push access to other nodes, so that each node maintains a predefined level of consistency for each dimension. Then each query can be sent to a node having a satisfying level of consistency (w.r.t. the query) in order to optimize load balancing.

In this paper, we strive to capitalize on the work on relaxing database consistency for higher performance which we apply in the context of cluster systems. We make the following contributions:

- replicated database architecture for cluster systems that does not hurt application and database autonomy, using non intrusive database techniques, *i.e.* techniques that work independently of any DBMS;

- new preventive replication method that provides strong consistency without the overhead of synchronous replication, by exploiting the cluster's high speed network;

- transaction load balancing architecture which can trade-off consistency for performance using optimistic replication and execution rules;

- conflict manager architecture which exploits the database logs and execution rules to perform replica reconciliation among heterogeneous databases.

This paper is organized as follows. Section 2 introduces our cluster system architecture with database replication. Section 3 presents our replication model with both preventive and optimistic replication. Section 4 describes the way we can capture and exploit execution rules about applications. Section 5 describes our execution model which uses these rules to perform load balancing and manage global consistency. Section 6 briefly describes our on-going implementation. Section 7 compares our approach with related work. Section 8 concludes.

2. Cluster architecture

In this section, we introduce the architecture for processing user requests coming, for instance, from the Internet, into our cluster system and discuss our solution for placing applications, DBMS and databases in the system.

The general processing of a user request is as follows. First, the request is authenticated and authorized using a directory which captures information about users and applications. The directory is also used to route requests to nodes. If successful, the user gets a connection to the application (possibly after instantiation) at some node which can then connect to a DBMS at some, possibly different, node and issue queries for retrieving and updating database data.

We consider a cluster system 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 [10]. 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. Using shared disk or shared cache requires a specific DBMS implementation like Oracle Rapid Application Cluster or DB2 Parallel Edition. Shared-nothing is the only architecture that supports our autonomy requirements

without the additional cost of a special interconnect. Thus, we strive to exploit a shared-nothing architecture.

There are various ways to organize the applications, DBMS and databases in our shared-nothing cluster system. We assume applications typically written in a programming language like C, C++ or Java making DBMS calls to stored procedures using a standard interface like ODBC or JDBC. Stored procedures are in SQL, PSM (SQL3's Persistent Stored Modules) or any proprietary language like Oracle's PL/SQL or Microsoft's TSQL. In [4], we presented and discussed three main organizations to obtain parallelism. The first one is *client-server DBMS* connection whereby a client application at one node connects to a remote DBMS at another node (where the same application can also run). The second organization is *peer-to-peer DBMS* connection whereby a client application at one node connects to a local DBMS which transparently accesses the same DBMS at another node using a distributed database capability. The third organization is *replicated database* whereby a database and DBMS is replicated across several nodes. These three organizations are interesting alternatives which can be combined to better control the consistency/performance trade-off of various applications and optimize load balancing. For instance, an application at one node could do client-server connection to one or more replicated databases, the choice of the replicated database being made depending on the load.

In this paper, we focus on the replicated database organization which is the most general as it provides for both application and database access parallelism. We use multi-master replication [14] whereby each (master) node can perform updates to the replica it holds. However, conflicting updates to the database from two different nodes can yield to consistency problems (*e.g.* the same data get different values in different replicas). The classical solution to this problem is optimistic and based on conflict detection and resolution. However, there is also a preventive solution which we propose and avoids conflicts at the expense of a forced waiting time for transactions. Thus, we support both replication schemes to provide a continuum from strong consistency with preventive replication to weaker consistency with optimistic replication.

Based on these choices, we propose the cluster system architecture in Figure 1 which does not hurt application and database autonomy. Applications, databases and DBMS are replicated at different nodes without any change by the cluster administrator. Besides the directory, we add 4 new modules which can be implemented at any node. The *application load balancer* simply routes user requests to application nodes using a traditional load balancing algorithm. The *transaction load balancer* intercepts DBMS procedure calls (in ODBC or JDBC) from the applications, generates a transaction execution plan (TEP), based on application and user consistency requirements obtained from the directory. For instance, it decides on the use of preventive or optimistic replication for a transaction. Finally, it triggers transaction execution (to execute stored procedures) at the best nodes, using run-time information on nodes' load. The *preventive replication manager* orders

transactionsat each node in a way that prevents conflicts and generates refresh transactions to update replicas. The *conflict manager* periodically detects conflicts introduced on replicas by transactions run in optimistic mode using the DBMS logs and solves them using information in the directory. At each node, the local DBMS ensures local serialization of transactions it executes, including refresh transaction and solving transactions. Global consistency is ensured at the level required by the application, either by the preventive replication manager or the conflicts manager.

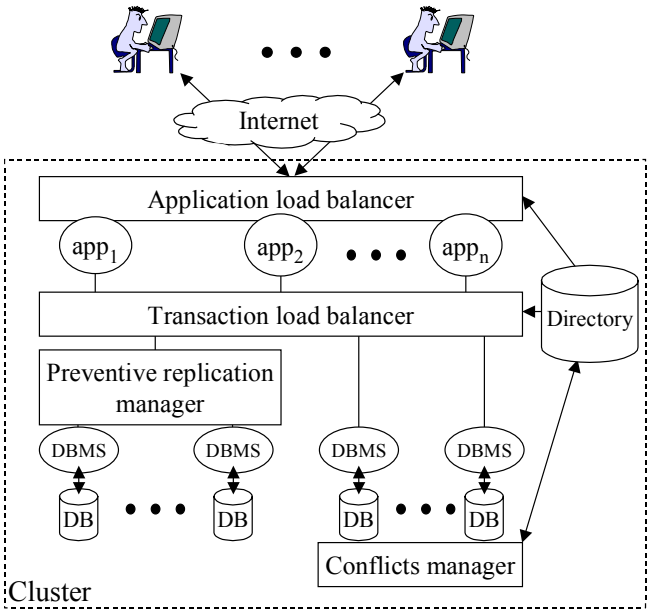

Figure 1: *Cluster system architecture*

3. Replication Model

In our context, replication of data at different cluster nodes is a major way to increase parallelism. However, updates to replicated data need be propagated efficiently to all other copies. A general solution widely used in database systems is lazy replication. In this section, we discuss the value of lazy replication in cluster systems, and propose a new multi-master lazy replication scheme with conflict prevention and its architecture. Our scheme can also reduce to the classical multi-master replication scheme with conflict resolution.

3.1. *Lazy replication*

With lazy replication, a transaction can commit after updating a replica at some node. After the transaction commits, the updates are propagated towards the other replicas, which are then updated in separate transactions. Unlike synchronous replication (with 2 phase commit), updating transactions need not wait that mutual copy consistency be enforced. Thus lazy replication does not block and scales up much better compared with the synchronous approach. This performance advantage has made lazy replication widely accepted in practice, *e.g.* in data warehousing and collaborative applications on the Web [12].

Following [13] we characterize a lazy replication scheme using: ownership, configuration, transaction model propagation, refreshment. The *ownership* parameter defines the permissions for updating replicas. If a replica *R* is updateable, it is called a *primary copy*, otherwise it is called a *secondary copy*, noted *r*. A node *M* is said to be a master node if it only stores primary copies. A node *S* is said to be a slave node if it only stores secondary copies. In addition, if a replica copy *R* is updateable by several master nodes then it is said to be a *multi-owner copy.* A node *MO* is said to be a *multi-owner master* node if it stores only multi-owner copies. For cluster computing we only consider master, slave and multi-owner master nodes. A master node *M* or a multi-owner node *MO* is said to be a master of a slave node *S* iff there exists a secondary copy of r in *S* of a primary copy *R* in *M* or *MO*. We also say that *S* is a *slave* of *M* or *MO.*

The *transaction model* defines the properties of the transactions that access replicas at each node. Moreover, we assume that, once a transaction is submitted for execution to a local transaction manager at a node, all conflicts are handled by the local concurrency control protocol. In our framework, we fix the properties of the transactions. We focus on four types of transactions that read or write replicas: update transactions, multi-owner transactions, refresh transactions and queries. An *update transaction T* updates a set of primary copies. A refresh transaction, *RT,* is associated with an update transaction *T*, and is made of the sequence of write operations performed by *T* used to refresh secondary copies. We use the term *multi-owner transaction,* noted *MOT*, to refer to a transaction that updates a *multi-owner copy*. Finally, a *query Q,* consists of a sequence of read operations on primary or secondary copies.

The *propagation* parameter defines when the updates to a primary copy or multi-owner copy *R* must be *multicast* towards the slaves of *R* or all owners of *R*. The multicast protocol is assumed to be reliable and preserve the global FIFO order [16]. We focus on deferred update propagation: the sequence of operations of each refresh transaction associated with an update transaction *T* is multicast to the appropriate nodes within a single message *M,* after the commitment of *T.*

The refreshment parameter defines when should a *MOT* or *RT* be triggered and the commit order of these transactions. We consider the *deferred* triggering mode.

With a *deferred-immediate* strategy, a refresh transaction *RT* or multi-owner transaction *MOT* is submitted for execution as soon as the corresponding message *M* is received by the node.

3.2. *Managing replica consistency*

Depending on which node is allowed user updates to a replica, several replication configurations can be obtained. The lazy master (or asymmetric) configuration allows only one node, called master node, to perform user updates on the replica; the other nodes can only perform reads. Figure 2(a) shows an example of a lazy master bowtie configuration in which there are two nodes storing primary copies R and S and their secondary copies r_1, s_1 and r_2, s_2 at the slave nodes. The multi-master (or symmetric) configuration allows all nodes storing a replica to be masters. Figure 2(b) shows an example of a multi-master configuration in which all master nodes stores a primary copy of S and R. There are also hybrid configurations.

Different configurations yield different performance/consistency trade-offs. For instance a lazy master configuration such as bowtie is well suited for read intensive workloads because reading secondary copies does not conflict with any update transaction. In addition, since the updates are rare, the results of a query on a secondary copy *r* at time *t* would be, in most cases, the same as reading the corresponding primary copy R at time *t*. Thus, the choice of a configuration should be based on the knowledge of the transaction workload. For update intensive workloads, the multi-master configuration seems best as the load of update transactions can be distributed among several nodes.

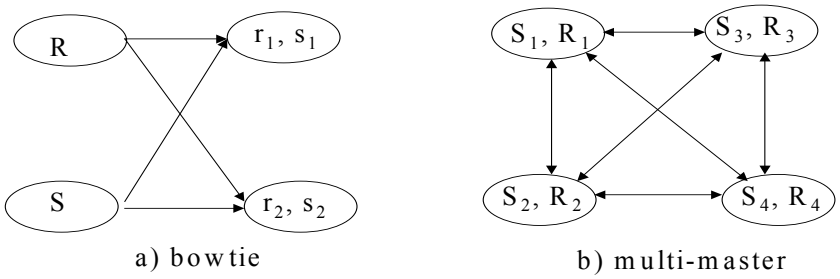

Figure 2: *Replication configurations*

For all configurations, the problem is to manage data consistency. That is, any node that holds a replica should always see the same sequence of updates to this replica. Consistency management for lazy master has been addressed in [13]. The

problem is more difficult with multi-master where independent transactions can update the same replica at different master nodes. A conflict arises whenever two or more transactions update the same object. The main solution used by replication products [19] is to tolerate and resolve conflicts. After the commitment of a transaction, a conflict detection mechanism checks for conflicts which are resolved by undoing and redoing transactions using a log history. During the time interval between the commitment of a transaction and conflict resolution, users may read and write inconsistent data. This solution is optimistic and works best with few conflicts. However, it may introduce inconsistencies.

We propose an alternative, new solution which prevents conflicts and thus avoids inconsistency. A detailed presentation of the preventive replication scheme and its algorithms is in [14]. With this preventive solution, each transaction T is associated with a chronological timestamp value, and a delay d is introduced before each transaction submission. This delay corresponds to the maximum amount of time to propagate a message between any two nodes. During this delay, all transactions received are ordered following the timestamp value. After the delay has expired, all transactions younger than T are guaranteed to be received. Therefore, transactions at each node are executed following the same timestamp order and consistency is assured.

This preventive approach imposes waiting a specific delay d, before the execution of multi-owner and refresh transactions. Our cluster computing context is characterized by short distance, high performance inter-process communication where error rates are typically low. Thus, d can be negligible to attain strong consistency. On the other hand, the optimistic approach avoids the waiting time d but must deal with inconsistency management. However, there are many applications that tolerate reading inconsistent data. Therefore, we decided to support both replication schemes to provide a continuum from strong consistency with preventive replication to weaker consistency with optimistic replication.

3.3. *Preventive Replication Manager Architecture*

This section presents the system architecture of a master, multi-master or slave node with conflict prevention. This architecture can be easily adapted to the simpler optimistic approach, with the addition of a conflict manager (see Section 5). To maintain the autonomy of each node, we assume that six components are added to a regular database system in order to support lazy replication (see Figure 3). The *Replica Interface* manages the incoming multi-owner transaction submission. The *Receiver* and *Propagator* implement reception and propagation of messages, respectively. The *Refresher* implements a refreshment algorithm. Finally, the *Deliverer* manages the submission of multi-owner transactions and refresh transactions to the local transaction manager.

The *Log Monitor* uses log sniffing to extract the changes to primary copies by continuously reading the content of a local History Log (noted *H*). The sequence of updates of an update transaction T and its timestamp *C* are read from *H* and writtento the *Input Log,* that is used by the Propagator.

Next, 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.* Afterwards, the sequence of operations of *MOT* is written into the *Owner Log* followed by *C*. Whenever the multi-owner transaction commits, the *Deliverer* notifies the event Replica Interface. After *MOT* commitment, the replica interface ends its processing and the application program continues its next execution step.

The *Receiver* implements message reception. Messages are received and stored in a *Reception Log.* The receiver then reads messages from this log and stores each message in an appropriate *FIFO pending queue*. The content of the queues form the input to the Refresher. The *Propagator* reads continuously the contents of the *Owner* and *Input Log* and for each sequence of updates followed by *C* read, it constructs a message *M.* Messages are multicast through the network interface.

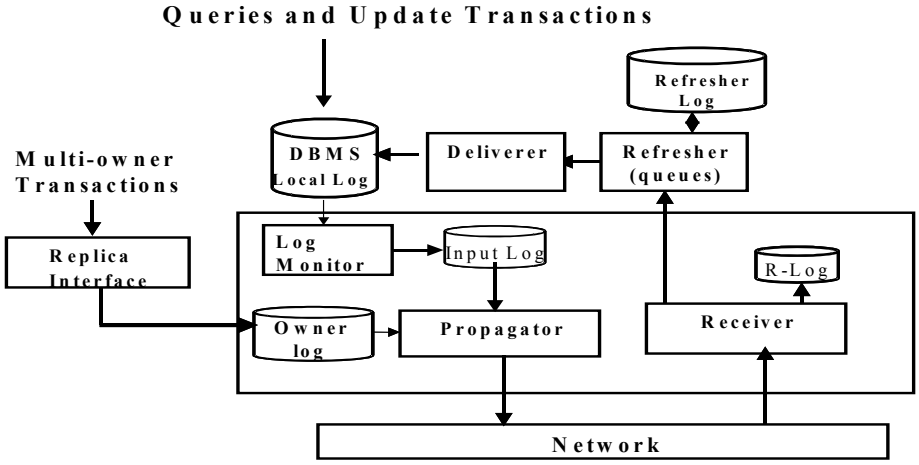

Figure 3: *Master, Multi-owner or Slave node Architecture*

The *Refresher* implements the refreshment algorithm. It reads the contents of a set of pending queues. Based on its refreshment parameters, it submits refresh transactions and multi-owner update transactions by inserting them into the running queue. The running queue contains all ordered transactions not yet entirely executed. Finally, the *Deliverer* submits refresh and multi-owner transactions to the local transaction manager. It reads the contents of the running queue in a FIFO order

and submits each write operation as part of a transaction to the local transaction manager. Whenever a multi-owner transaction is committed, it notifies the event to the *Replica Interface.*

4. Trading consistency for load balancing

The replicated database organization may increase transaction parallelism. For simplicity, we focus on inter-transaction parallelism, whereby transactions updating the same database are dynamically allocated to different master nodes. There are two important decisions to make for an incoming transaction:choosing the node to run it, which depends on the current load of the cluster, and the replication mode (preventive or optimistic) which depends on the degree of consistency desired. In this section, we show how we can capture and use execution rules about applications in order to obtain transaction parallelism, by exploiting the optimistic replication mode.

4.1 *Motivating example*

To illustrate how we can tolerate inconsistencies, we consider a very simple example adapted from the TPC-C benchmark[2]. Similar to the Pharmacy applications in our Leg@net project, TPC-C deals with customers that order products whose stock must be controlled by a threshold value. We focus on table Stock(item, quantity, threshold). Procedure DecreaseStock decreases the stock quantity of item *id* by *q*.

```
procedure DecreaseStock(id, q) :
UPDATE Stock
SET quantity = quantity – q
WHERE item = id;
```

Let us consider a Stock tuple <item=1,quantity=30,threshold=10> replicated at nodes N1 and N2, transaction T1 at N1 that calls DecreaseStock(1, 15) and transaction T2 at N2 that calls DecreaseStock(1, 10). If T1 and T2 are executed in parallel in optimistic mode, we get <item=1,quantity=15,threshold=10> at N1 and <item=1,quantity=20,threshold=10> at N2. Thus, the Stock replicas are inconsistent and require reconciliation. After reconciliation, the tuple value will be <item=1,quantity=5,threshold=10>. Now, assume query Q that checks for stocks to renew:

[2] see www.tpc.org/tpcc

SELECT item FROM Stock where quantity < threshold

Executing Q at either node N1 or N2 will not retrieve item 1. However, after reconciliation (see Section 6.2), the final value of item 1 will be <item=1,quantity=5,threshold=10>. If the application tolerates inconsistencies, it is aware that the results may have been incomplete and can either reissue Q after some time necessary to reach the next reconciliation step and produce a correct result, or execute Q at either node N1 or N2 and produce the results with a bounded inaccuracy. In our example, item 1 would not be selected, which may be acceptable for the user.

Assume now there is an integrity constraint C : (quantity > threshold*0.5) on table Stock. The final result after reconciliation clearly violates the constraint for item 1. However, this violation cannot be detected by either N1 after executing T1 or N2 after executing T2. There are two ways to solve this problem: either prevent T1 and T2 to be executed at different nodes in optimistic mode, or, at reconciliation time, validate one transaction (*e.g.* with highest priority) and compensate, if possible, the other one.

4.2. *Execution rules*

Application consistency requirements are expressed in terms of *execution rules*. Examples of execution rules are data-independency between transactions, integrity constraints [1], access control rules, etc. They may be stored explicitly by the system administrator or inferred from the DBMS catalogs. They are primarily used by the system administrator to place and replicate data in the cluster, similar to parallel DBMS [12], [20]. They are also used by the system to decide at which nodes and under which conditions a transaction can be executed.

Execution rules are stored in the directory (see Figure 4) . They are expressed in a declarative language. Implicit rules refer to data already maintained by the system (*e.g.* users authorizations). Hence, they include queries sent to the database catalog to retrieve the data. Incoming transactions are managed by the policy manager. It retrieves execution rules associated with a given transaction and defines a run-time policy for the transaction. The run-time policy controls the execution of the transaction at the required level of consistency. The couple (transaction, run-time policy) is called *transaction policy* (TP) and is sent to the transaction router, which in turns computes a cost function to elaborate the *transaction execution plan* (TEP) which includes the best node among the candidates to perform the transaction with the appropriate mode.

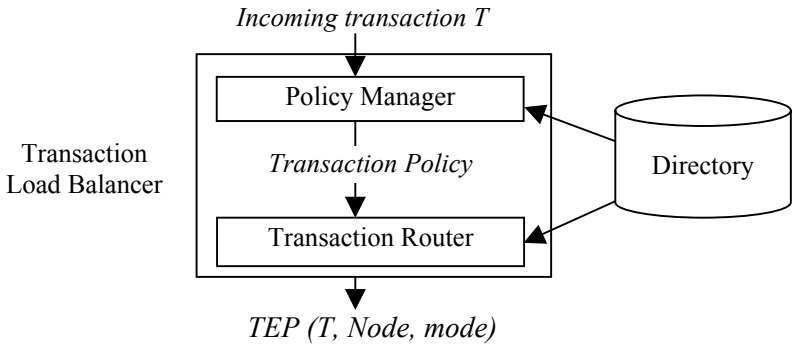

Figure 4 : *transaction load balancer architecture*

4.3. *Defining execution rules*

A transaction is defined by T = (P, param, user, add-req), where :

P is the application program of which T is an instance with *param* as parameters.

user is the user who sends the transaction, and

add-req are additional execution requirements for the transaction

Execution rules use information about the transaction (the four elements above) and the data stored in the database. In order to preserve application autonomy, execution rules cannot be defined at a finer granularity than the program. Such information may be already existing in the database catalog. Otherwise, it must be explicitly specified or inferred from other information. Information related to a program includes its type (query or update), its conflict classes [15] *i.e.* the data the program may read or write, and its relative priority. If P is a query, the required precision of the results must be specified. If P is an update, the directory must capture under which conditions (parameters) T is compensatable, the compensating transaction, whether T should be retried and under which temporal conditions. For a couple (P, D), where D is a data in the write conflict class of P, the administrator may specify *max-change(P,D)* which is defined as follows If D is an attribute, *max-change* states how much changes (relative or absolute) a transaction T(P) may do to D. If D is a table, max-change states how many tuples T(P) may update. In our example, we have max-change(Decrease-Stock(id,q), Stock.quantity) = q, max-change(Decrease-Stock(id,q), Stock) = 1.

This information is used to determine the run-time policy for transaction T and a partially ordered set of candidate nodes at which the transaction may be executed.

As those nodes may be master, slave or multi-owner, the run-time policy may be different for each type of node. The run-time policy is described as follows :

type(T) : denotes if T is a query or a (multi-owner) transaction.

priority(T) : the absolute priority level of the transaction, computed from the relative priorities of the user and the program, and the relative priority of the transaction itself (included in the *add-req* parameter).

compatible(T, T') : for each transaction T', this vector stores whether T and T' are disjoint (resp. commutative). It is computed with compatibility information about programs, conflict classes and effective parameters of the transactions. This information is used by the load balancer to address transactions to different nodes in non-isolated mode to the replication manager. In our example, it is obvious that T1 and T2 are not disjoint but are commutative.

query-mode(T) : if T is a query, the query mode models the spatial and temporal dimensions of the query quality requirements. For instance, a query may tolerate an imprecision of 5% if the answer is delivered within 10 seconds, or of 2% if delivered within one minute and may choose to abort if the response time is beyond 2 minutes. The query mode may include different spatial dimensions (absolute value, number of updates) and give a trade-off between the temporal and spatial dimensions (*e.g.* find the best precision within a given time or answer as soon as possible within a given acceptable precision). In our example, we assume that Q accepts an error of at most 5 items in the results.

update-mode(T): the update mode models if a multi-owner transaction may be performed in non-isolated mode on a master copy and compensated if the conflict resolution fails, under which temporal conditions and with which compensating transaction. The update mode also models under which conditions a transaction should be automatically retried if aborted. In our example, neither T1 nor T2 are compensatable since the Decrease-Stock procedure corresponds to a real withdrawal of goods in stock.

IC(T) : the set of integrity constraints T is likely to violate. In our example, IC(T1) = IC(T2) = {C}.

max-change(T,D): for each data D, the maximum of change the transaction may produce. In our example,we have max-change(T1, Stock.quantity) = 15, max-change(T2, Stock.quantity) = 10, max-change(T1, Stock) = max-change(T2, Stock) = 1.

In our example, the following TPs would be produced :

(*T1*, *type* = trans., *priority* = null, *compatible* = (), *update-mode* = no-compensate, *IC* = (C), *max-change* = { (Stock.quantity, 15), (Stock, 1) })

(*T2*, *type* = trans., *priority* = null, *compatible* = ((T1, commut.)), *update-mode* = no-compensate, *IC* = (C), *max-change* = { (Stock.quantity, 10), (Stock, 1) })

(*Q*, *type* = query., *priority* = null, *compatible* = ((T1, no-commut.), (T2, no-commut.)), query-mode = ((imprecision = 5 unit), (time-bound = no), (priority = time)).

5. Execution model

In this section, we present the execution model for our cluster system. The objective is to increase load balancing based on execution rules. The problem can be reduced as follows: given the cluster's state (nodes load, running transactions, etc.), the cluster's data placement, and a transaction T with a number of consistency requirements, choose one optimal node and execute T at that node. Choosing the optimal node requires to first choose the replication mode, and then choose the candidate nodes where to execute T with that replication mode. This yields a set of TEPs (one TEP per candidate node) among which the best one can be selected based on a cost function. In the rest of this section, we present the algorithm to produce candidates TEPs and the way to select the best TEP and execute it. Finally, we illustrate the transaction routing process on our running example.

5.1. *Algorithm for choosing candidate TEPs*

A TEP must specify whether preventive or optimistic replication is to be used. Preventive replication always preserves data consistency but may increase contention as the degree of concurrency increases. On the other hand, optimistic replication performs replica synchronization after the transaction commitment at specified times. If a conflict occurs during synchronization, since the transaction has been committed, the only solutions are to either compensate the transaction or notify the user or administrator. Thus, optimistic replication is best when T is disjoint from all the transactions that accessed data in optimistic replication mode since the last synchronization point, and when the chance of conflict is low.

The algorithm to choose the candidate TEPs proceeds as follows. The input is a transaction policy consisting of transaction T, conflict description (*i.e.* which transactions commute or not with T), required precision (*Imp* value), and update description (*maxChange* property). The output is a set of candidate TEPs, each specifying a node N for executing T and how. The algorithm has two steps. First, it finds the candidate TEPs with preventive replication to prevent the occurrence of non resolvable conflicts. This involves assessing the probability of conflict between T and all the committed transactions that have accessed data in optimistic mode since the last synchronization point. In case of potential conflict at a node, a TEP with preventive replication is built.

In case of non-conflicting transaction T (second step), the algorithm finds the candidate nodes with optimistic replication for data accessed by T. If the execution

of T requires accessing consistent data not available at a node, the algorithm adds the necessary synchronization before processing T as follows. For each node N and each data D, it computes the imprecision *Imax(D,N)* of processing T at N. Imax is the sum of the maximum changes of all transactions t such that (i) updates of t and T are not disjoint, (ii) t was processed at a node M ≠ N, (iii) t updates are not yet propagated to N. If N can process T with the required consistency (*i.e.* Imax(D,N) < Imp(N) for all D accessed by T) then a TEP (T,N) is built. Otherwise, the minimal synchronization for N is specified and added to the TEP.

5.2. *Choice of optimal node and transaction execution*

Choosing the optimal node is an optimization problem with the objective of minimizing response time. The cost of a given TEP(T, N, sync) includes the synchronization cost (for propagating all updates to N) and the processing cost of T at N. After an optimal node is selected, the transaction router triggers execution as follows. First, it performs synchronization if necessary. If preventive replication has been selected, it sends the transaction to the receiver module of the replication manager. Otherwise, it sends the transaction directly to the DBMS. By default, the transaction router checks precision requirements of T before processing it, assuming that they are still valid at the end of T. This is not always true if a concurrent update occurs during the processing of T. To ensure that the precision requirements of T are met at the end of T, a solution is to process T by iteration until the precision requirements are reached. If the precision requirements cannot be reached, T must be aborted. Otherwise, T can be committed. Another solution is to forbid the execution of concurrent transactions that would prevent T from reaching its consistency requirements.

In order to compute the necessary synchronization before processing T at node N, the transaction router maintains for each node the list of all transactions to be propagated. Then, the algorithm for synchronization is the following : (i) if synchronization yields no conflicts, the transaction router executes T on N, (ii) otherwise, replica synchronization is delegated to the conflict manager.

To have detailed information about the current state of replicas, the conflict manager reads the DBMS log (which keeps the history of update operations) of the nodes to synchronize. Then, it resolves the conflicts based on priority, timestamps, or user-defined reconciliation rules. If automatic conflict resolution is not possible, the conflict manager sends a notification alert to the user or the administrator with details about the conflicting operations.

5.3. *Example of transaction routing*

Let us now illustrate the previous algorithms on the transactions of the example of Section 4.1 and show how TEPs are produced from the TP sent by the policy manager. We assume that the TPs are received in order (T1, T2, Q), that data at nodes N1 and N2 is accessed in optimistic mode and that no other transaction is running and conflicting with T1, T2 or Q. We first consider a case where integrity constraint C is not taken into account. Then we show how C influences transaction routing.

Case 1 : no integrity constraint

Upon receiving TP (*T1*, *type* = trans., *priority* = null, *compatible* = (), *update-mode* = no-compensate, *IC* = (C), *max-change* = {(Stock.quantity, 15), (Stock, 1)}), the transaction router does the following:

computes the set of candidate nodes {N1, N2};

detects that T1 is not conflicting with any running transaction, thus the candidate nodes are {N1, N2};

sends T1 to the least loaded node (say N1) with T1 as synchronization, which means that N1 must send T1 to the other node as a synchronizing transaction;

infers Imax(Stock, N1)=1, which means that at most one tuple can be modified at N1 before synchronization.

Upon receiving TP (*T2*, *type* = trans., *priority* = null, *compatible* = ((T1, commut.)), *update-mode* = no-compensate, *IC* = (C), *max-change* = {(Stock.quantity, 10), (Stock, 1)}), the transaction router does the following :

computes the set of candidate nodes {N1, N2};

detects that T2 is conflicting with T1 but commutes with it;

sends T2 to the least loaded node (assume N2) with T2 as synchronization. As T1 and T2 are commutable, the order in which they will be executed at N1 (resp. N2) does not matter;

infers Imax(Stock, N2) = 1.

Upon receiving TP (*Q*, *type* = query., *priority* = null, *compatible* = ((T1, no-commut.), (T2, no-commut.)), query-mode = ((imprecision = 5 unit), (time-bound = no), (priority = time))), the transaction router does the following:

computes the set of candidate nodes {N1, N2};

detects that Q is conflicting with both T1 and T2;

from the current values of Imax(Stock, N1) and Imax(Stock, N2), it computes that executing Q at either N1 or N2 would yield a result with an imprecision of at

most one unit. As the query mode imposes an imprecision of at most 5 units, Q is sent to the least loaded node (say N1)

In the case the query mode of Q was not allowing any imprecision, the router would have waited for the next synchronization of N1 and N2 to send Q.

Case 2 : with integrity constraint

The transaction router would detect that both T1 and T2 are likely to violate C and are not compensatable. Sending T1 and T2 to different nodes could lead to the situation where C is not violated at either N1 or N2, but is violated during synchronization. Since T1 and T2 are not compensatable, this situation is not acceptable and T2 must be sent to the same node as T1. Then we have Imax(Stock, N1) = 0 and Imax(Stock, N2) = 2. Upon receiving Q, the transaction router may still choose the least loaded node to execute it. Since the priority is given to time in the query mode, the least loaded node is chosen : N2. Had the priority been given to precision, N1 would have been selected by the transaction router.

6. Implementation

In this section, we briefly describe our current implementation on a cluster of PCs under Linux. We plan to first experiment our approach with the Oracle DBMS. However, we use standards like LDAP and JDBC, so the main part of our prototype is independent of the target environment.

6.1 *Transaction load balancer*

The transaction load balancer is implemented in Java. It acts as a JDBC server for the application, preserving the application autonomy through the JDBC standard interface. Inter-process communication between the application and the load balancer uses RmiJdbc open source software[3]. To reduce contention, the load balancer takes advantage of the multi- threading capabilities of Java based on the Linux's native threads. For each incoming transaction, the load balancer delegates transaction policy management and transaction routing to a distinct thread. The transaction router sends transactions for execution to DBMS nodes through JDBC drivers provided by the DBMS vendors. To reduce latency when executing transactions, the transaction router maintains a pool of JDBC connections to all cluster nodes.

[3] see www.objectweb.org/rmijdbc

6.2 *Conflict manager*

The conflict manager is composed of three modules :

Log analyzer : reads the DBMS log to capture the updates made by the local transactions;

Conflict solver : analyzes updates made on each node in order to detect and solve conflicts;

Synchronizer : manages synchronization processes.

At each node, the log manager runs as an independent process and reads the log to detect updates performed by the local transactions. Those updates are sent to the conflict solver through time-stamped messages. The Oracle LogMiner tool can be used to implement this module, and messages can be managed with the Advanced Queuing tool. To illustrate the log analysis on our running example, the Stock update made by transaction T1 is detected by the following query on the LogMiner schema:

```
SELECT scn, sql_redo, sql_undo
FROM v$logmnr_contents
WHERE seg_name='STOCK';
```

The conflict solver receives messages from the log analyzer, analyzes them to detect conflicting writes (*e.g.* update/update or update/delete) on a same data. It then computes how conflicts can be solved. In the best case, the conflict is solved by propagating updates from one node to the other ones. In the worst case, the conflict is solved by choosing a transaction (the one with less priority) to be compensated. In both cases, synchronizing transactions (propagation or compensation) are sent to the corresponding nodes.

The synchronizer receives synchronizing transactions from the conflict solver. It may execute them either periodically or upon receiving an order from the transaction load balancer for an immediate synchronization.

The preventive replication manager is implemented as an enhanced version of a preceding implementation [11]. To implement reliable message broadcast, we plan to use Ensemble [6], a group communication software package from Cornell University.

6.3 *Directory*

All information used for load balancing (execution rules, data placement, replication mode, cluster load) is stored in an LDAP compliant directory. The directory is accessed through Java Directory Naming Interface (JDNI) which

provides an LDAP client implementation. Dynamic parameters measuring the cluster activity (load, resource usage) are stored in the directory. They are used for transaction routing. Values are updated periodically at each node. To measure DBMS node activity, we take advantage of dynamic views maintained by Oracle. For instance, the following queries collect CPU usage and I/O made by all running transactions at a node :

```
select username, se.sid, cpu_usage
from v$session ss, v$sesstat se,
v$statname sn
where se.statistic# = sn.statistic#
and name like '%CPU used by this
session%'
and se.sid = ss.sid
order by value desc
```

```
select username, os_user, process pid,
ses.sid,
    physical_reads, block_gets,
consistent_gets,
    block_changes, consistent_changes
from v$session ses, v$sess_io sio
where ses.sid = sio.sid
order by physical_reads, ses.username
```

6.4 *Planned experimentations*

The experimental cluster is composed of 5 nodes: 4 DBMS nodes + 1 application node. At each DBMS node, there is a TPC-C database (500MB to 2GB) and TPC-C stored procedures. The application node sends the transactional workload to the transaction load balancer through TPC-C stored procedures. The cluster data placement and replication (optimistic and preventive) may be configured depending on the experiment goal. We plan to first measure the cluster performance (transactional throughput) when the database is replicated on the 4 nodes and accessed in either preventive or optimistic replication mode, by varying the update rates and the probability of conflict. We also plan to measure the scalability of our approach. We will simply implement a logical 16 node cluster where each node runs 4 DBMS instances to behave like a 4 node cluster. Based on the actual performance numbers with 4 nodes, this will yield confident results.

7. Comparison with related work

The main work related to ours is replication in either large-scale distributed systems or cluster systems and advanced transaction models that trade consistency for improved performance. In synchronous replication, 2PC can be used to update replicas. However 2PC is blocking and the number of messages exchanged to control transaction commitment is significant. It cannot scale up to cluster configurations to large numbers of nodes. [7] addresses the replica consistency problem for synchronous replication. The number of messages exchanged is reduced compared to 2PC but the solution is still blocking and it is not clear whether it scales up. In addition, synchronous solutions cannot perform load balancing as we do. The common point with our preventive approach is that we both consider the use of communication services to guarantee that messages are delivered at each

node in a specific order. [13] proposes a refreshment algorithm that assures correctness for lazy-master configurations, but does not consider multi-master configurations as we do here. Multi-master asynchronous replication [19] has been successfully implemented in commercial systems such as Oracle and Sybase. However, only the optimistic approach with conflict detection and conciliation is supported.

There are interesting projects for replicated data management in cluster architectures. The PowerDB project at ETH Zurich deals with the coordination of the cluster nodes in order to provide a uniform and consistent view to the clients. Its solution fits well for some kinds of applications, such as XML document management [5] or read-intensive OLAP queries [17]. However, it does not address the problem of seamless integration of legacy applications. The GMS project [21] uses global information to optimize page replacement and prefetching decisions over the cluster. However, it mainly addresses system-level or Internet applications (such as the Porcupine mail server). Other projects are developed in the context of wide-area networks. The Trapp project at Stanford University [8] addresses the problem of precision/performance trade-off. However, the focus is on numeric computation of aggregation queries and minimizing communication costs. The TACT middleware layer [24] implements the continuous consistency model. Despite the fact that additional messages are used to limit divergence, a substantial gain in performance may be obtained if users accept a rather small error rate. However, read and write operations are mediated individually: an operation is blocked until consistency requirements can be guaranteed. This implies monitoring at the server level, and it is not clear if it allows installation of a legacy application in an ASP cluster. In the quasi-copy caching approach [1], four consistency conditions are defined. Quasi-copies can be seen as materialized views with limited inconsistency. However, they only accept single master replication, which is not adapted to our multi-master replication in a cluster system. Finally, epsilon transactions [23] provide a nice theoretical framework for dealing with divergence control. As in the continuous consistency model [24], it allows different consistency metrics to give answers to queries with bounded imprecision. However, it requires to significantly alter the concurrency control, since each lock request must read or write an additional counter to decide whether the lock is compatible with the required level of consistency. In summary, none of the existing approaches addresses the problems of leaving databases and applications autonomous and unchanged as in our work.

8. Conclusion

In this paper, we proposed a new solution for parallel processing with autonomous databases in a cluster system for ASP, using a replicated database organization. The main idea is to allow the system administrator to control the consistency/performance tradeoff when placing applications and databases onto

cluster nodes, which is not possible when using an existing parallel DBMS. Application requirements are captured through execution rules stored in a shared directory. They are used at configuration time to choose the best organization for applications and databases. They are also used at run-time to either prevent or tolerate copy inconsistency in order to optimize load balancing.

Capitalizing on the work on relaxing database consistency for higher performance, this paper makes several contributions in the context of cluster systems. First, we defined a replicated database architecture for clusters systems that does not hurt application and database autonomy. We use non intrusive techniques by intercepting DBMS transaction calls or exploiting DBMS's log interfaces.

Second, we proposed a new preventive replication method that provides strong consistency without the overhead of synchronous replication by exploiting the cluster's high speed network. The preventive replication architecture maintains DBMS's autonomy and can support optimistic replication as well, with the addition of a conflict manager.

Third, we proposed a transaction load balancing architecture which can trade-off consistency for performance using optimistic replication and execution rules. Support for both preventive and optimistic replication provides a continuum from strong consistency to weaker consistency with different cost/performance. Execution rules can be defined at different levels of granularity (program or transaction, table or attribute or tuple) to express application semantics. The distinction between Transaction Policy (what we want) and Transaction Execution Plan (how we optimize it) eases the system's evolution (by changing rules) and load balancing decisions.

Finally, we presented an execution model to execute Transaction Execution Plans in a way that optimizes load balancing. The optimal node is selected based on the replication mode that should be used and a cost function which estimates nodes's load. We also proposed a conflict manager architecture which exploits the database logs and execution rules to perform replica reconciliation among heterogeneous databases.

We have started to implement the proposed solution on LIP6's cluster architecture running Linux and Oracle 8i. In the near future, we will experiment with the TPC-C benchmark to assess the cost/performance of preventive replication and optimistic replication (with relaxed consistency) under various workloads. We will also develop a simulation model, calibrated with our implementation, to study how our solution scales up to very large cluster configurations.

References

[1] R. Alonso, D. Barbará, H. Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM Transactions on Database Systems (TODS)*, 15(3), 1990.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil. A Critique of ANSI SQL Isolation Levels. In *ACM SIGMOD Int. Conf. on Management of Data*, 1995.

[3] A. Doucet, S. Gançarski, C. León, M. Rukoz. Checking Integrity Constraints in Multidatabase Systems with Nested Transactions. In *Int. Conf. On Cooperative Information Systems (CoopIS)*, 2001.

[4] S. Gançarski, H. Naacke, P. Valduriez. Load Balancing of Autonomous Applications and Databases in a Cluster System. In *4th Workshop on Distributed Data and Structure (WDAS)*, 2002.

[5] T. Grabs, K. Böhm, H.-J. Schek. Scalable Distributed Query and Update Service Implementations for XML Document Elements. In *IEEE RIDE Int. Workshop on Document Management for Data Intensive Business and Scientific Applications*, 2001.

[6] M. Hayden. The Ensemble System. Technical Report, Departement of Computer Science, Cornell University, TR-98-1662, 1998.

[7] B. Kemme, 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.

[8] C. Olston, J. Widom. Offering a Precision-Performance Tradeoff for Aggregation Queries over Replicated Data. In *Int. Conf. on Very Large Databases (VLDB)*, 2000.

[9] T. Özsu, P. Valduriez: Principles of Distributed Database Systems. Prentice Hall, 2nd edition, 1999.

[10] T. Özsu, P. Valduriez. Distributed and Parallel Database Systems - Technology and current state-of-the-art. *ACM Computing Surveys*, 28(1), 1996.

[11] E. Pacitti. Improving Data Freshness in Replicated Databases. PhD Thesis, INRIA-RR 3617, 1999.

[12] E. Pacitti, O. Dedieu. Algorithms for Optimistic Replication on the Web. *Journal of the Brazilian Computing Society*, 2002, to appear.

[13] E. Pacitti, P. Minet, E. Simon. Replica Consistency in Lazy Master Replicated Databases. *Distributed and Parallel Databases*, 9(3), 2001.

[14] E. Pacitti. Preventive Lazy Replication in Cluster Systems. Technical Report RR-2002-01, CRIP5, University Paris 5, 2002.

[15] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, G. Alonso. Scalable Replication in Database Clusters. In *14th Int. Conf. on Distributed Computing (DISC)*, 2000.

[16] D. Powel et al. Group communication (special issue). *Communication of the ACM*, 39(4), 1996.

[17] U. Röhm, K. Böhm, H.-J. Schek. Cache-Aware Query Routing in a Cluster of Databases. *Int. Conf. on Data Engineering (ICDE),* 2001.

[18] A. Sheth, M. Rusinkiewicz. Management of Interdependent Data: Specifying Dependency and Consistency Requirements. *Workshop on the Management of Replicated Data*, 1990.

[19] D. Stacey. Replication: DB2, Oracle, or Sybase. Database Programming & Design. 7(12), 1994.

[20] P. Valduriez. Parallel Database Systems: open problems and new issues. *Int. Journal on Distributed and Parallel Databases*, 1(2), 1993.

[21] G. Voelker et al. Implementing Cooperative Prefetching and Caching in a Global Memory System.In *ACM Sigmetrics Conf. on Performance Measurement, Modeling, and Evaluation*, 1998.

[22] G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Transactions on Database Systems (TODS)*, 16(1), 1991.

[23] K. L. Wu, P. S Yu, C. Pu. Divergence Control for Epsilon-Serializability. In *8th Int. Conf. on Data Engineering (ICDE)*, 1992.

[24] H. Yu, A. Vahdat. Efficient Numerical Error Bounding for Replicated Network Services. In Int. Conf. On Very Large Databases (VLDB), 2000.