Adaptive Middleware for Data Replication[⋆]

J. M. Milan-Franco[1] and R. Jiménez-Peris[1], M. Patiño-Martínez[1], and B. Kemme[2]

[1] Facultad de Informática, Universidad Politécnica de Madrid (UPM), Spain
milanjm@sip.ucm.es,{rjimenez,mpatino}@fi.upm.es
[2] McGill University, School of Computer Science, Montreal, Quebec, Canada
kemme@cs.mcgill.ca

Abstract. Dynamically adaptive systems sense their environment and adjust themselves to accommodate to changes in order to maximize performance. Depending on the type of change (e.g., modifications of the load, the type of workload, the available resources, the client distribution, etc.), different adjustments have to be made. Coordinating them is already difficult in a centralized system. Doing so in the currently prevalent component-based distributed systems is even more challenging. In this paper, we present an adaptive distributed middleware for data replication that is able to adjust to changes in the amount of load submitted to the different replicas and to the type of workload submitted. Its novelty lies in combining load-balancing techniques with feedback driven adjustments of multiprogramming levels (number of transactions that are allowed to execute concurrently). An extensive performance analysis shows that the proposed adaptive replication solution can provide high throughput, good scalability, and low response times for changing loads and workloads with little overhead.

1 Introduction

Tuning an information system in order to provide optimal performance for a specific application is a challenging task. So far, human administrators have traditionally outperformed software based solutions due to their in-depth knowledge of the application. However, more and more applications cannot be described anymore with static characteristics but are dynamic in nature. For instance, depending on the time of the day, the workload submitted to a database can be sometimes update intensive, sometimes read intensive. Also, the type of users connected to the database at a given time, or the current state of the application (e.g., in workflows) will have an influence on which parts of a database are accessed more frequently. This dynamic behavior requires prompt and frequent adjustments of the underlying information system. This, however, will be difficult to achieve when relying on humans to turn the right knobs in the system configuration. Instead, dynamic applications require an adaptive infrastructure. Adaptive systems use context awareness to perceive the surrounding context and the effect of its reactions on it. Based on this monitoring information, the system modifies itself to accommodate to the new conditions in the computing environment in which it operates,

[⋆] This work has been partially funded by the European Commission under the Adapt project, IST-2001-37126, and by the Spanish Research Council, MCyT, under project TIC-2001-1586-C03.

and thus, behaves in an optimal fashion according to some target performance metrics. The most ambitious goal of an adaptive system is autonomy, in which a system adapts itself automatically to ambient changes [18].

In this paper, we focus on the automatic adaptation of a replicated database. Database replication is used for scalability (read-only queries can be executed at any replica; hence, the more replicas the more queries can be served), fast response times (adding new replicas reduces the resource consumption on each replica), and fault-tolerance (the data is available as long as one replica is accessible). Hence, replicating databases over a cluster of workstations has become attractive for current web-based information systems that are used for read-intensive applications like online bookstores or auction systems (more than 50% reads). The challenge is replica control, i.e., changes of update transactions have to be applied at all replicas in a consistent way. Database replication at the middleware level has received considerable attention in the last years [14, 1, 2, 26, 27, 19], since it can be developed without the need to change the underlying database system. In such systems, transactions are submitted to the middleware which then forwards it to the appropriate database replica(s) (both to provide replica control as for load-balancing). For locality and fault-tolerance, the middleware is usually also replicated. Transactions can then be submitted to any middleware replica.

Target performance metrics for database systems are throughput (rate of executed transactions) and response time. These metrics depend on the workload (mix of transaction types), load (rate of submitted transactions), cache hit ratio, etc. In order to have optimal resource utilization in a replicated database, transactions have to be equally distributed among all replicas. Another important aspect of the database system to work well under a given workload is the *multiprogramming level* (MPL), i.e., the number of transactions that are allowed to run concurrently within the database system. Initially, when resources are freely available, then a high MPL boosts throughput. Also, if some transactions are I/O bound, concurrent transactions might keep the CPU busy while I/O takes place. However, when resources are highly utilized or a single resource becomes the bottleneck (e.g., the log), increasing the MPL will only increase context switches, and hence, put even more restraint on the resources. Performance is then lower than in a system with less concurrent transactions. Also, if conflict rates are high, additional transactions will only lead to higher abort rates, and hence, wasted execution time.

In dynamic environments, workload and/or the load can change over time. As a result, the system configuration has to be adapted dynamically, i.e., the MPL and the distribution of transactions across replicas must be adjusted. An additional dynamic behavior is the crash of individual components. If a node fails, the other nodes must not only be able to continue execution but should also take over the load of the failed node.

The contribution of this paper lies in providing a hierarchical approach with two levels of adaptation for a replicated database. At the local level, the focus is on maximizing the performance of each individual replica by adjusting the MPL to changes in the load and workload. At the global level, the system tries to maximize the performance of the system as a whole by deciding how to share the load among the different replicas. The challenge of performing these kinds of adaptation at the middleware level is the reduced information that is available about the changes in behavior and internals of the database making it hard to detect bottlenecks.

At the local level, each middleware instance has a pool of active connections to the local database replica. This determines the MPL since each transaction requires its own connection. If there are more transaction requests than active connections, transactions are enqueued at the middleware. In this way, each local replica is able to handle intermediate periods of high load. In order to adapt to changes in workload and load, the number of active connections has to be adjusted dynamically. We use an approach adapted from [11] that only requires to monitor the database load and the achieved throughput in order to adapt the MPL appropriately. Other approaches that we are aware of [20, 6] require a deeper knowledge of the database internals, and hence are not appropriate. But even [11] is designed for database internal adaptation. Hence, we had to extend their approach to work at the middleware level. Using this local approach, each replica adapts its MPL individually so that it can execute its load in the most efficient way without any specific knowledge of the workload nor of the database internals.

However, if the number of transactions to be executed at a replica continuously exceeds the optimal MPL, the set of transactions enqueued at the middleware becomes bigger and bigger, and the response time for those transactions deteriorates. Increasing the MPL would only worsen the situation. Instead, some of the load assigned to this replica should be redirected to other replicas with free capacity. In order to detect replicas that are overloaded or have free capacity, the local load of each replica has to be monitored. Our approach achieves this by keeping track of the number of transactions enqueued at each middleware server. If a significant imbalance is detected, the load will be redistributed. Load balancing takes place continuously as a background process in order to capture workload changes. In order to not consume too many resources, we use a greedy-algorithm with little computation overhead. Other load-balancing algorithms we are aware of (e.g., [22, 3, 2]) have a quite different transaction execution model (i.e. only deal with queries), and hence, result in different solutions.

We have evaluated our approach extensively for local and global adaptation, both in isolation and combined, in a cluster of workstations under various workloads and loads. Our results show that hierarchical adaptation provides evenly loaded replicas, avoids deterioration of individual database replicas, and is able to handle intermediate periods of overload. Furthermore, it is able to handle various workloads and database sizes, and adapts smoothly to changes in the environment.

The rest of the paper is structured as follows. Section 2 introduces the architecture of Middle-R [14] that was used as our base system. Section 3 discusses the local adaptation of MPL, and Section 4 is devoted to global load balancing adaptation. In Section 5 we show some of the results of our extensive performance evaluation. Related approaches are compared in Section 6. We conclude in Section 7.

2 Middleware-based Database Replication

Middle-R is a cluster based database replication tool that serves as target and base system for our adaptability analysis. [14] describes the system in more detail. The system consists of N nodes (machines), each node hosts a database system and a Middle-R server. Each database system stores a full copy of the database (replica).

The database application programs are written in the usual way, using one of the standard database interfaces to interact with the database (the current system is based on C programs with Embedded SQL). Given a transaction in form of an application program, Middle-R can identify which data objects are accessed by the transaction. Object granularity can be a table, or any application specific granularity level.

Middle-R is responsible for concurrency and replica control. It uses a group communication system to disseminate information among the replicas. The group communication system, Ensemble [10], provides support for group maintenance and reliable multicast. One of the available multicast primitives provides total order delivery, i.e., although different servers might multicast messages concurrently, the group communication system will deliver to each server all messages in exactly the same order (a sending server will also receive its own message in total order). This order is used as execution order for conflicting transactions that want to access the same objects. In order to achieve this, Middle-R servers maintain a lock table, and requests locks in the total order in which transactions are received.

The system applies asymmetric transaction processing to boost scalability [16, 15]. Each update transaction (consisting of at least one update operation) is only executed at one replica. The other replicas do not re-execute the transaction (neither read nor write operations) but simply change the affected records in an efficient manner. This spare capacity can be used to process additional transactions. Several analyses have shown that asymmetric processing can outperform symmetric processing [16, 14, 15]. Furthermore, symmetric processing is not feasible for database systems with triggers or non-deterministic behavior. Hence, most commercial systems use asymmetric replication approaches. In order to use asymmetric processing at the middleware layer, the underlying database system has to provide a function to get the changes performed by a transaction (the *write set*), and a second that takes the write set as input and applies it without re-executing the entire SQL statements. Such an extension was implemented for PostgreSQL and is currently being implemented for MySQL and Microsoft SQL Server. Oracle uses such mechanism for its own replication protocol [21].

In order to share the load among all the replicas, we follow a primary copy approach. Each set of data objects that can be accessed within a single transaction is assigned a primary replica that will be in charge of executing programs that access this specific set. For instance, replica $N1$ might be primary of object set $\{O1\}$ and replica $N2$ of object sets $\{O2\}, \{O1, O2\}$. That is, overlapping sets might be assigned to different replicas. With this we allow transactions to access arbitrary object sets. Disallowing overlapping object sets to be assigned to different replicas means that we partition the data among the replicas, each replica being responsible for transactions accessing object sets within its partition. As a result, we would disallow object sets spanning two or more partitions, and hence disallow transaction to access arbitrary objects. The primary of a multiple object set (for transactions spanning multiple object sets) is selected by selecting one of the primaries of the basic object sets. This is done through a deterministic function (e.g. a hash function such as SHA-1) applied to the basic object sets of a transaction. In this way, each transaction has a primary and it is only needed to assign primaries to each basic object set (instead of selecting a primary for every possible combination of basic object sets). The conflict-aware scheduling algorithm and its correctness can be found at [24].

Assigning object sets to primary nodes determines which transactions are executed at which nodes. Given a static workload, an optimal distribution of object sets can easily be found. However, when the workload characteristics change over time, a reassignment is necessary. This is the task of our global adaptation (dynamic load balancing).

Let us first have a look at *update transactions* that perform at least one update operation. The client can submit an *update request* to any Middle-R server which multicasts it to all middleware servers using total order multicast. Upon receiving a request to execute transaction T delivered in total order, all servers append the lock requests for objects accessed by T into the corresponding queues of the lock table (a form of conservative 2PL locking). The primary executes T when T's locks are the first in all queues. It starts a database transaction, executes T's code, and retrieves the write set from the database. Then it commits T locally and multicasts (without ordering requirement) the write set to the other Middle-R servers which apply it at their databases. The Middle-R server which originally received the client request returns the commit confirmation once it receives the write set (or after local commit if it was the primary of the transaction).

For queries (read-only transactions), there exist several alternatives. Firstly, they could always be executed locally at the server they are submitted avoiding communication. However, this disallows any form of load balancing, and if all requests are submitted to one server, this server will quickly become a bottleneck. Since communication in a local area network is usually not the bottleneck, an alternative is to also execute queries at the primary. Apart of load balancing issues this might lead to a better use of the main memory of the database system since each replica is primary only of a subset of the data. Hence, we can expect higher cache hit ratios at each replica [3] than if each replica executes any type of query. In the primary approach, a query request is forwarded to all replicas; the primary then executes the query, returns the result to the submitting Middle-R server and notifies the end of the query to all replicas. In this paper, we assume a primary approach for queries. Independently of whether a local or primary approach is used the executing Middle-R server might not need to acquire locks for queries but immediately submit them for execution if the database uses snapshots for queries (as is done by PostgreSQL or Oracle). The approach provides 1-copy-serializability because all replicas decide on the same execution order of conflicting transactions due to the total order multicast. Even if the primary fails after committing but before sending the changes, a new primary will take over and re-execute the transaction in the same order due to the total order multicast.

At the start of Middle-R, a pool of connections to the database is created as common for application servers (since connection creation is very expensive). Initially all connections are marked as free. Whenever a transaction is able to execute or a write set can be applied, the system looks for a free connection. If there is none, execution is delayed. Otherwise the system takes one of the free connections, marks it as busy, and submits the necessary statements over this connection to the database. Once the transaction terminates, the connection is again marked as free, and waiting requests are activated. Dynamically adjusting the size of the connection pool to the workload characteristics is the task of our local adaptation protocol.

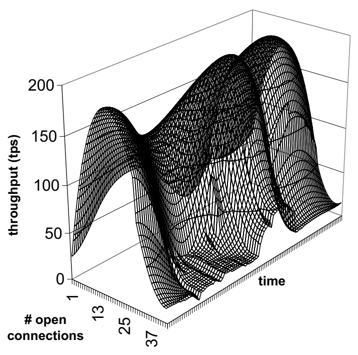


Fig. 1. Throughput as a function of the MPL over time

3 Local Level Adaptation

At the local level, each middleware server is configured to maximize the performance of its local database replica. Measurements have shown that Middle-R servers are light-weight while the database servers are the first to be the bottleneck [14]. Hence, controlling the MPL is an important step in dynamic performance optimization, and is done by limiting the connection pool to the database replica.

Our solution to control the MPL is based on the feedback control approach proposed in [11]. Since it does not require database internal information like conflict rate, memory and other resource consumption, etc. it is suitable for a middleware-based system. In a feedback system, one uses the output of the system as an indicator whether the input of the system should be changed. [11] proposes to take the transaction throughput as output parameter. In a system without control on the number of concurrent transactions, the throughput of the database system usually rises with increasing the number of transactions until the system saturates at a throughput peak. If the number of concurrent transactions increases further, the database enters the thrashing region in which the throughput falls very fast until it stabilizes at some low residual value. Fig. 1, adjusted from [11], illustrates this behavior. The x-axis depicts the MPL, the y-axis depicts the throughput achieved by the system with it, and the z-axis shows how the curve changes over time (assuming the workload changes over time). For instance, the percentage of update transactions has a significant influence on the maximum throughput since all update transactions must access the log. This means, e.g., that a read-only workload could have a high throughput peak at a high number of concurrent transactions, while update transactions could have a considerably smaller peak at a smaller number of concurrent transactions. When now the workload moves from a read intensive workload to a write intensive workload, so does the dependency between MPL and throughput.

Hence, we have two goals. Firstly, at any given time with a given workload, we have to determine the optimal MPL, i.e., to deny newly submitted transactions to execute whenever this would lead to a load that cannot be handled anymore by the database. That is, we should set a MPL such that the database system is never in the thrashing region. The *optimal MPL* is now defined as the MPL allowing for the maximum achievable throughput. The second goal is to provide dynamic adaptability, that is, to adjust the MPL when the workload changes such that it is never higher than the optimal MPL. [11] approximates the relationship between concurrent transactions and throughput at each time point with a parabola. In order to estimate the coefficients, the system periodically measures the number of concurrent transactions and the throughput. In order to capture the time dependency of the parabola, more recent measurements are given a higher weight than older measurements[1]. After each measurement period, the optimal MPL is set to the number of concurrent transactions achieving the highest throughput. The approach also addresses some stability problems. If the load is very stable, too few different data points are collected to correctly estimate the parabola. If results are imprecise one can get phenomena like inverted parabolas. Finally, if the workload changes too fast, adaptation can lead to a ping-pong effect. For all of them, [11] proposes several counter-measures which are all implemented in our system.

3.1 Adaptive MPL Implementation

Middle-R implements this approach with all proposed optimizations at the middleware layer by controlling the connection pool. Each Middle-R server creates a pool of $cmax$ open connections to the database at startup. We assume that $cmax$ is chosen big enough to be at least as large as the largest optimal MPL for any given workload (otherwise $cmax$ could be increased at runtime). At any time, out of the $cmax$ connections, at most $mplmax \leq cmax$ connections can be used. Additionally, Middle-R keeps track of $mplcurrent$, the number of connections that are currently used to execute transactions. $mplcurrent$ is always smaller or equal to $mplmax$. If $mplcurrent < mplmax$, then there are less requests available for execution than Middle-R would allow to execute. If $mplcurrent = mplmax$, then all allowed connections are used, and Middle-R will queue any further requests until a currently active transaction has terminated. r depicts the number of waiting requests. If r is zero, then $mplcurrent$ reflects the current load submitted to the replica. If r is greater than zero, then the middleware is reducing the load of the database by queuing requests.

Middle-R now periodically measures $mplcurrent$, and the database load and throughput. From there, it estimates the load/throughput parabola, and $mplopt$, the optimal MPL for the given workload. $mplmax$ is now constantly adjusted. However, $mplmax$ is not simply set to $mplopt$ but takes into account that during the last period the load submitted to the system might have been smaller than the maximum achievable throughput. This is the case if, for the last period, $mplcurrent$ was smaller than the estimated $mplopt$. In such case, it is better to keep $mplmax$ closer to the actual load, $mplcurrent$, in order to guarantee that once $mplmax$ is changed it will actually have the desired effect. As an example why this is needed, assume the type of workload has changed (e.g., many more updates) requiring to decrease the MPL. A decrease in $mplmax$ will not have an immediate effect if $mplmax$ is bigger than $mplcurrent$. As such, we can distinguish the following cases.

– The system is in underload (the system is in the increasing slope of parabola).

[1] Parabola coefficients are estimated using a recursive least-square estimator with exponentially fading memory.

    - If $mplcurrent = mplmax$, then $mplmax$ is increased. It is set to a value between its current value and the newly calculated $mplopt$. The larger r (waiting requests), the closer the new value will be to $mplopt$.
    - If $mplcurrent < mplmax$, then $mplmax$ is decreased to $mplcurrent$.
    - If $mplcurrent \geq mplmax$, nothing is done.
- The system is at peak throughput ($mplmax$ equals the newly calculated $mplopt$)
    - If $mplcurrent \geq mplmax$, nothing is done.
    - If $mplcurrent < mplmax$, then $mplmax$ is decreased to $mplcurrent \geq mplmax$.
- The system is in thrashing region ($mplmax > mplopt$)
    - If $mplcurrent = mplmax$, $mplmax$ is decreased by one whenever an active transaction terminates and as long as $mplmax > mplopt$.
    - The case $mplcurrent < mplmax$ cannot occur in thrashing region.

If the number of submitted requests is consistently higher than the throughput that can be handled by the database, the queues at the Middle-R server will become longer and longer. Middle-R can handle this for a certain time period. After this, the system performance degrades. The solution is to perform a global adaptation in order to redistribute some of the load of the overloaded replica to other nodes. If all replicas are overloaded, the system should disallow new client requests until active transactions have terminated.

4 Global Level Adaptation

A replicated database might potentially improve its throughput as more replicas are added to the system [15]. However, this potential throughput is only reached under an even load in which all replicas receive the same amount of work (assuming a homogeneous setting), which in practice might never happen. If the load is concentrated at one replica, the throughput will be the throughput of a single replica or even worse due to the overhead of the replication protocol to ensure consistency among replicas. Load balancing is aimed to correct situations in which some replicas are overloaded, while others have still execution capacity. This is done by redistributing the load as evenly as possible among the replicas. Therefore, any load balancing algorithm requires a means to estimate the current load at each replica.

Each Middle-R server knows the total number of concurrent active transactions in the system, since requests are multicast to all servers. All servers acquire locks for the objects accessed by transactions that are kept until the transaction terminates locally. Hence, looking at its lock table, each server has a good estimate of the total number of active transactions. For some of them the server is the primary copy. We call these the local transactions of the server. Local transactions might either be executing or waiting for locks or waiting for a free connection. For others, the server is not the primary. We call them remote transactions. If it is an update transaction the server is waiting for the write set (it is still active at the primary), or currently applying the write set or waiting for a free connection (the transaction is committed at the primary). If it is a query, the write set message is empty and used as an indication of the end of the query.

Each server can estimate the number of local active transactions of any other server S by looking at its lock table and the active transactions for which S is primary and for which it has not yet received the write set message. This calculation can be done without any additional communication overhead.

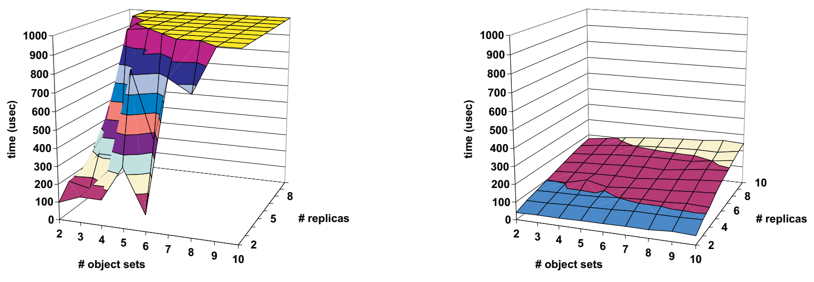
The number of local active transactions at a server is a good estimate of the load at this server. The higher this number, the higher the load of this server. If it is known that different transaction types have different execution times, then this load metrics can be made more precise by weighting the number of local active transactions with the observed average execution time [2]. The degree of balance (or imbalance) is captured by the variance among the number of local active transactions at each node. A variance with a value of zero means that the load is evenly distributed (balanced) among all replicas. On the other extreme, the maximum variance is achieved when the entire load is concentrated on a single replica.

4.1 Dynamic Load Balancing Implementation

The load balancing algorithm is in charge of finding a primary assignment of object sets to servers that minimizes the variance. Our first algorithm uses a branch-and-bound mechanism. For each object set it defines the load of this object set as the number of active transactions that want to access this set. For each replica it defines the current load as the sum of the loads of object sets for which that replica is the primary. At the start all object sets have to be assigned and the load of each replica is zero. Object sets are now assigned to primaries in the following way. At each step the algorithm selects the most loaded object set and assigns it to all servers yielding a set of partial solutions. The algorithm then traverses all partial solutions and prunes those that will not yield in a better solution than the current one (initially there are no computed solutions, hence all partial solutions are explored). The pruning in branch & bound algorithms is based on an estimation function. This function is applied to a partial solution and provides a lower bound of the variance of the optimal solution that might be found searching from this partial solution. The input of the estimation function is the total load of all object sets that have not yet been assigned, and a partial solution. The function assigns the transactions in that unassigned load to the less loaded servers in the partial solution in order to minimize the variance. This assignment, however, does not take into consideration the object sets accessed by these transactions. That is, two transactions accessing the same object set might be assigned to different servers. Hence, the function provides a lower bound of the actual possible variance. The algorithm provides an optimal assignment due to it performs an exhaustive search in the solution space. However, its use is limited to small number of object sets and replicas since its computation time grows exponentially.

Our alternative is an inexpensive greedy algorithm. It uses the same definitions of object set load and current replica load as the branch-and-bound algorithm. The greedy algorithm assigns at each step an object set to a replica. It selects the unassigned object set with the highest load and assigns it to the replica with the smallest current load. It proceeds recursively until all object sets are assigned to a replica.

The cost of both algorithms in terms of CPU time as a function of the number of replicas and the number of object sets is shown in Fig. 2. The branch-and-bound algo-



(a) Branch-and-bound (b) Greedy

Fig. 2. Computation time of the two load balancing algorithms

rithm goes beyond 1 ms of computing time with a number of object-sets and replicas higher than 5. The greedy algorithm takes a tenth of millisecond to compute a primary assignment for 10 replicas and 10 object sets. It is worth to note that the number of object sets is usually higher than the number of replicas. Fortunately, as can be seen in the graph, the computational cost of the greedy algorithm grows at a lower pace with the number of object sets than with the number of replicas.

A Middle-R server S responsible for load balancing periodically calculates the load variance. If the variance exceeds a given threshold, the load balancing algorithm calculates a new primary assignment to balance the load. The new primary assignment is not applied immediately. Instead S checks whether the current imbalance will be improved by the new configuration by a significant percentage. This prevents performing redistribution when the system performance cannot be improved appreciably. If the new configuration is better than the previous one, S multicasts a load balancing message m_l in total order to all Middle-R servers to inform about the new primary assignment. Transaction requests that are received after m_l are executed according to the new configuration. Reconfiguration for each object set takes place once all transaction requests accessing that object set that were received before m_l have been executed. For object sets that are not currently accessed by any transaction, reconfiguration can take place immediately. The time of the switch is different for each object set, depending on the number of transactions requesting access to it.

5 Experimental Results

5.1 Experiment Setup

The experiments were run in a cluster of 10 homogeneous machines. Each machine is equipped with two processors AMD Athlon 2GHz, 512 MB of RAM, and a 60 GB disk (with a 30 GB partition used for the experiments). The nodes were interconnected through a 100-MBit switch. The database used is PostgreSQL 7.2 enriched with a couple of services that enable asymmetric processing at the middleware level [14].

We used two different database sizes in the experiments. One is very small and easily fits into main memory (a 10MB database). After the warm-up it was guaranteed that all data was kept in the database cache and that no access to disk was needed to read data. The other is a medium-sized database (1 GB) that was set up very scattered forcing I/O on every access with a high probability. In any case, the database consisted of 36 tables with equal number of tuples (3,000 and 300,000 tuples, respectively).

Three workloads have been chosen to evaluate the adaptation. UPD8 is a pure update transaction that performs 8 SQL update statements on the same table. Each statement changes one tuple that is determined by indicating the value of its primary key. Hence, access to this tuple is through the B+-tree of the primary key without reading any other tuples of the table. SELUPD is an update statement that queries a table in order to perform a single update. It represents a mixed workload of read and write accesses. LQ1 is a read only query that queries a full table. That is, each transaction accesses a single table, and there exist 36 object sets each containing one table[2]. The first three experiments analyze local adaptation, and hence, are run on a single server. The remaining experiments use up to 9 machines to evaluate global adaptation. In all experiments an additional machine was used to run the client simulator.

All experiments (except the temporal ones, 1.3 and 2.1) consisted of three phases: 1) A warm-up phase (500 transactions), during which the load was injected but no measures were taken; 2) a measurement phase (1000 transactions) in which the end-to-end throughput and response time are measured; 3) and finally, a cold-down phase (500 transactions) in which the load was kept without taking any measurements. Each experiment was repeated at least 3 times.

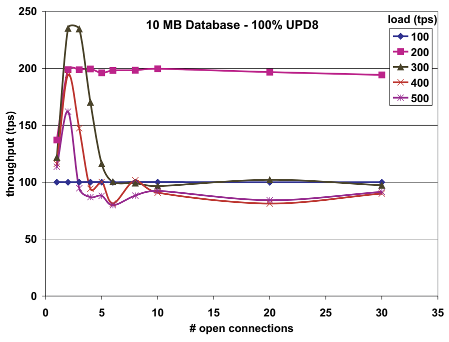
5.2 Local Adaptation

Experiment 1.1: Optimal MPL This experiment aims to motivate that there is an optimal value for $mplmax$, i.e., the number of connections that are made available at the middleware server, and this optimal is different under different workloads.

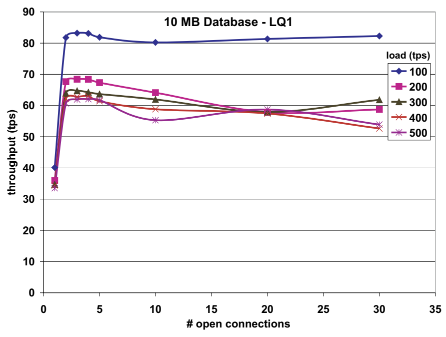
In these experiments, no adaptive algorithm was run. Instead in each test run, $mplmax$ was set to a specific value. Then a given load (in transactions per second) was submitted to the system, and the throughput measured. If the throughput was smaller than the load, the system was overloaded. Each of the Fig. 3(a-d) present the throughput of the system (y-axis) given different values for $mplmax$ (x-axis) for different loads submitted to the system (different curves). For each curve the maximum value represents the maximum achievable throughput. We can observe that this peak is achieved at different values for $mplmax$, and hence, illustrates the need to adjust $mplmax$ according to the application type.

Fig. 3(a-b) show results on a small database where computation is CPU bound, whereby (a) uses update transactions while (b) uses queries. In both figures, we can see the throughput behavior as described in Fig. 1. In (a), at low loads the maximum throughput is equal to the load submitted to the system. At 100 transactions per second (tps), a single connection can already easily handle the load, and any additional connections will probably never be used. At 200 tps, two connections help to increase the

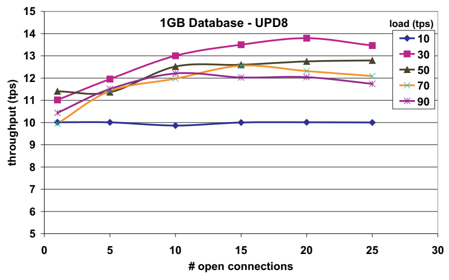
[2] We conducted experiments with transactions that accessed several tables. Response times of such workloads are generally higher due to the conflict behavior. The relative behavior of the adaptive system, however, was similar, and hence, we focus our discussion on transactions accessing a single table.



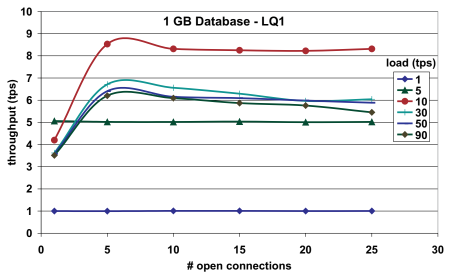
(a) CPU-bound workload 100% updates



(b) CPU-bound workload 100% queries



(c) IO-bound workload 100% updates



(d) IO-bound workload 100% queries

Fig. 3. Experiment 1.1: Optimal MPL for different workloads

throughput over one (which only can handle around 140 tps), and any additional connection is probably never used. Starting at 300 tps, the system is not able anymore to handle the entire submitted load. Two connections are able to achieve around 240 tps, but moving to three connections the system starts to deteriorate and allow a throughput of only 100 tps. All higher loads show similar behavior, however the maximum achievable throughput is even smaller, probably due to the fast growth of enqueued transactions at the middleware layer. The optimal MPL of 2 is the same as the number of CPUs available on the test machine. This means, it is optimal for the database that each CPU executes exclusively one transaction. Any increase in concurrent update transactions deteriorates performance. We assume that the reason is that all update transactions access the log to write the redo information. As far as we know, PostgreSQL does not perform any group commit (committing several transactions at the same time to reduce log induced I/O). Writing the log can easily take as long as executing the entire transaction in memory, hence, the degree of parallelism is limited by the time to flush the log to disk. For query workloads (b), the maximum throughput of 85 tps is achieved at an optimal MPL of 3-5. This MPL is higher than for update workloads. Queries do not have the log as single bottleneck. The reason that it is worth to execute more than one query on each CPU is probably due to the communication delay between application program and the database server. For queries, the query result has to be returned to the application program. The program uses complex data structures to retrieve the result, and the interaction between program and database can be quite complex. Hence, while the response is transferred from the database to the program and processed by the program, the server is free to execute additional queries.

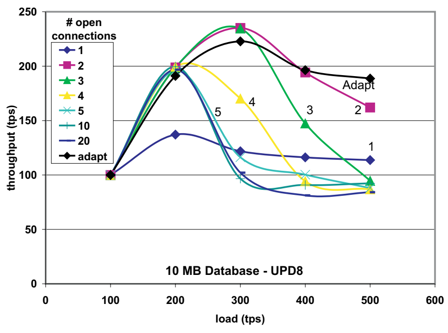
When we now move to an I/O bound configuration (Fig. 3.c-d), the system is less vulnerable to thrashing and the optimal degree of concurrency is very different. Generally, maximum achievable throughputs are much smaller than with small database sizes, since each transaction needs longer to finish. Write intensive workloads (Fig. 3.c) require a substantially larger connection number (around 20) to maximize throughput. Each transaction takes now considerable time to retrieve the 8 tuples to be updated since each operation probably requires I/O. Hence, the log is no more the bottleneck and we can take advantage of more concurrency in the system. For read intensive workloads (Fig. 3.d), once the system works at saturation (load of 10 tps), the optimal MPL is at around 5 concurrent transactions. We assume the reason why an increase in MPL does not lead to higher throughput is due to the fact that queries have higher main memory requirements in the database. The chosen query performs aggregation, and hence, some temporary results have to be stored in the database before results are returned to the application program. Hence, if too many queries run concurrently they compete for main memory and some intermediate results might be swapped to disk leading to thrashing [6].

The conclusion from this experiment is that there is no single optimal MPL that holds for every workload, but instead, each workload has a different optimal MPL. What is more, depending whether the workload is CPU-bound or IO-bound the optimal degree of concurrency is substantially different. Therefore, to attain an optimal performance on a continuous basis, an adaptive control of the number of active connections is required. Note that although the database size might be relatively fixed in a given application, the portion of the database that is accessed might change over time. For instance, in an online bookstore, during daytime there will be a lot of queries scanning huge data sets. During night, update intensive batch processing on subset of the data might be performed. When this change in workload occurs, the system should automatically adjust the number of active connections.

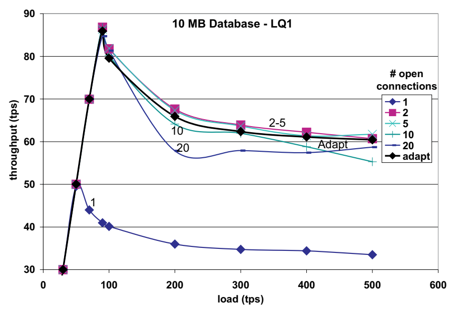
Experiment 1.2: Local adaptation under constant load This experiment is targeted to show the behavior of the local adaptation under a constant load. Although it might appear surprising at first glance, a constant load is one of worst case scenarios for adaptive algorithms. This is due to the fact that a constant load provides little information to the adaptive system [11]. In each setting of the experiment we measured the throughput for different values of $mplmax$ and compared it with the one obtained using the adaptive algorithm (local adaptation). Fig. 4 (a-d) show the achieved throughput (y-axis) when we increase the load in the system (x-axis) for different numbers of active connections (curves)[3]. Additionally, one curve indicates the throughput achieved by our adaptive algorithm.

The adaptive control exhibits a near-optimal throughput both for CPU and IO-bound workloads as well as in read and write intensive workloads. That is, for each given workload and load it dynamically determines the optimal value for $mplmax$. We want to

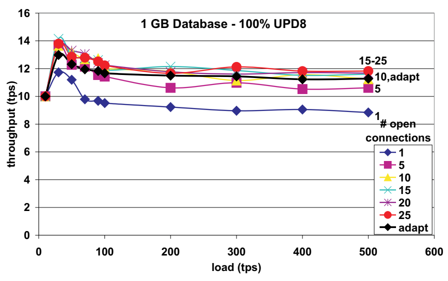
[3] The different data points can be extrapolated from Fig. 3, except for the adaptive curve.



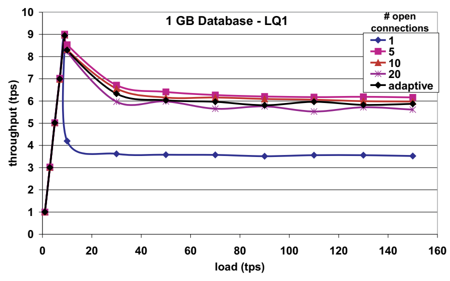
(a) 100% Updates CPU-bound workload



(b) 100% Queries CPU-bound workload



(a) 100% Updates IO-bound workload



(b) 100% Queries IO-bound workload

Fig. 4. Experiment 1.2: Adaptive number of connections vs. fixed number of connection under constant load

emphasize that this adaptation is achieved without having any knowledge of the workload or database size but is based solely on the observed throughput. It also does not need to know the reasons that might lead to the need for low or high MPLs ($mplmax$), as analyzed in the previous experiment. Hence, while manual tuning (increasing or decreasing the MPL) will require the database administrator to have knowledge of the current workload characteristics and their possible effects on concurrent transactions, the local adaptation algorithm chooses a nearly optimal concurrency level without any application or domain specific knowledge. Hence, it can be a general module of a middleware without any application specific adjustments.

Experiment 1.3: Temporal evolution of local adaptation The goal of this experiment is to show how long the local adaptation needs to determine the optimal MPL when the workload characteristics change. The workload chosen for the experiment is SELUPD and a 10MB database. In this case the maximum throughput was achieved with an MPL of 2. Since we want to show how long the system needs to adapt to this MPL, the experiment starts with an MPL of 20, i.e., far of being optimal. Fig. 5 shows the throughput achieved over time in intervals of 5 seconds and at a load of 200 tps. During the first two seconds of runtime, the system collects the historical information needed to adapt the MPL. Then, it increases the MPL only to determine that this does not improve

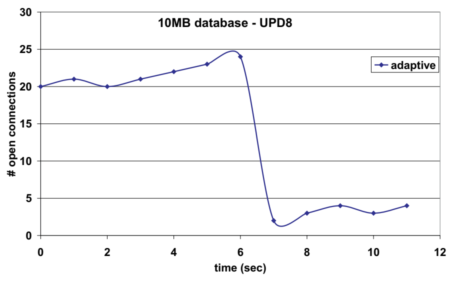
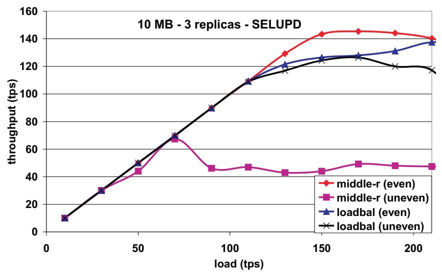
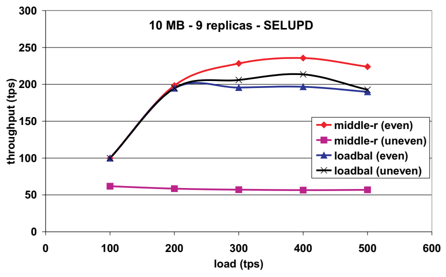


Fig. 5. Experiment 1.3: Temporal evolution of local adaptation



(a) 3 replicas



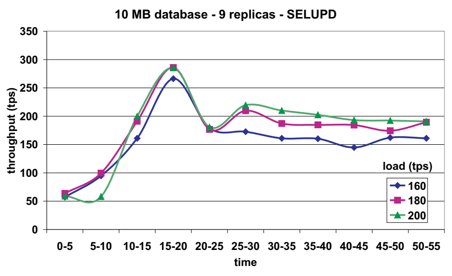
(b) 9 replicas

Fig. 6. Experiment 2.1: Global Adaptation vs. No Adaptation at 3 and 9 replicas

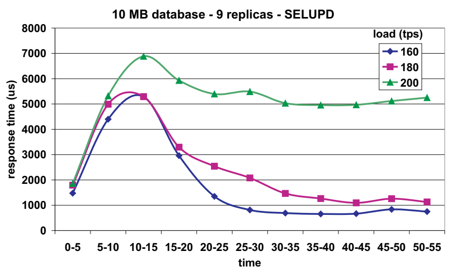
the behavior. However, it helps to build a parabola shaped transaction/throughput curve. The system now detects that it is in the downwards part of the parabola and realizes that it is in the thrashing region. At this point, the system starts to reduce drastically the MPL until it finds itself in the upwards part of the parabola. Then, the MPL stabilizes in a quasi-optimal interval, between 2-4. The adaptation of the MPL takes around 5 seconds (subtracting the 2 seconds it takes to collect the historical information). That is, in a system where workload changes do not appear every couple of seconds, our approach should not lead to any ping-pong behavior. It should be noticed that this experiment stress tests the system by imposing a very extreme change in the optimal MPL. Less extreme workload changes should lead to a quasi-optimal MPL in shorter time.

5.3 Global Adaptation

Experiment 2.1: Performance of Global Adaptation This experiment is aimed to measure the improvement in throughput provided by the load balancing algorithm. The experiment compares the performance of Middle-R with and without global adaptation for different workloads and number of replicas. Fig. 6 shows the throughput achieved with increasing loads for 3 and 9 replicas respectively. The workload consists of transactions of type SELUPD. Each graph includes four curves. One curve corresponds to



(a) Throughput



(b) Response time

Fig. 7. Experiment 2.2: Temporal evolution of the performance with dynamic load balancing

the best achievable throughput. This optimal throughput is achieved by Middle-R without load balancing with a totally even load that is, each replica is the primary of the same amount of objects (tables) and submitted transactions access these objects in a round-robin fashion. This curve provides an upper bound of the throughput for the load balancing algorithm. The lower bound is the throughput without load balancing for a totally imbalanced load. This imbalanced load consists in concentrating the entire load in a single replica (one replica is the primary for all object sets). The other two curves show the behavior of Middle-R using load balancing for even and uneven loads.

Let us first have a look at the results without load balancing. The throughput of the middleware without load balancing with the totally uneven load (lowest curve) is nearly the same (around 50 tps) independently of the number of replicas. This is a logical result since the throughput of the system is precisely the throughput of a single replica. Therefore, the number of replicas does not have any influence in the throughput. With an even load the 9 replicas achieve a higher maximum throughput than 3 replicas. This holds despite SELUPD is a pure update workload. The reason is the asymmetric processing of transactions where non-primary replicas only apply updates which require fewer resources. Hence, the maximum achievable throughput is higher. The performance gain is due to the fact that the primary performs many read operations that are not executed at non-primaries. As a result, for 3 replicas, the maximum throughput achievable is around 140 tps, whilst with 9 replicas it reaches 200 tps. That is, a 43% higher throughput.

When we now look at Middle-R with load balancing, we see that for an even load the achievable throughput is basically the same as without load balancing. This shows that the overhead introduced by the load balancing algorithm is negligible. When the system starts with an uneven load, we can see that the maximum achievable throughput is nearly as good as when the system starts with a balanced load. This is achieved by the global adaptation through redistribution of object sets (tables) to different primaries such that all replicas are primary of some of the accessed tables. The final distribution leads to an even load at all replicas yielding a quasi-optimal throughput.

Experiment 2.2: Temporal evolution of the global adaptation The previous experiment showed that the load balancing algorithm achieves a better performance than any of them in isolation. This experiment complements the previous one by showing how long the system needs to balance the load starting from a totally imbalanced situation (all the load is concentrated in a single replica). The experiment is run on 9 replicas and with a SELUPD workload. A configuration with 9 replicas is more sensitive to load imbalances and therefore, will better show how good the load balancing is. Three different loads have been used in the experiment. The load with which the optimal throughput is obtained (180 tps, the maximum load at which the throughput equals the load), a lower load (160 tps, underload), and a higher load (200 tps, thrashing) [4]. For all the loads, a single run of the load balancing algorithm achieved the optimal throughput. That happened in second 4 for 160 tps, and in second 2 for the other loads.

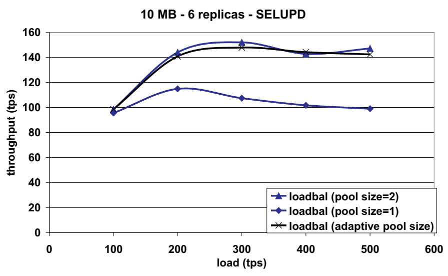
Fig.7.a and Fig.7.b exhibit the average throughput and response time as seen by clients. Fig.7.a shows that at the beginning the system provides low throughput due to the imbalance. Once the load balancing algorithm was triggered, the throughput increases very fast until it peaks at around 250 tps. This is more than the submitted load. The reason is that the achieved throughput (60 tps) was far lower at the beginning of the experiment than the submitted load what forces Middle-R servers to enqueue many transactions. Hence, once reconfiguration has taken place, the actual load submitted to the database is higher than the submitted load to Middle-R until all waiting transactions have been executed. Finally, the throughput levels off at the actually submitted load.

The response time (Fig.7.b) takes longer to reach the optimal (around 25 sec.). The reason is again that the system starts from a totally imbalanced situation which enqueues transactions at the middleware. Once the system has reconfigured itself to attain an optimal configuration in the first 2-4 seconds, there are many queued transactions. These pending transactions have high average response times (due to the long queues created by the initial imbalanced assignment) even with an optimal configuration till the system is able to catch up.

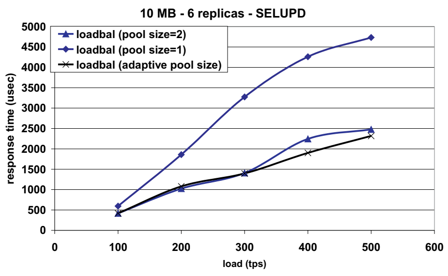
Experiment 2.3: Combining Local and Global Adaptation This experiment aims to show that the combination of local and global adaptation exhibits a performance close to the optimal. The experiment was run on six nodes and the workload used was SELUPD. Initially, for all curves, a replica is primary of all object sets (i.e. a totally imbalanced assignment). The initial number of connections in the adaptive curve is 20.

Fig. 8.a-b presents the (a) throughput and (b) response time with increasing load. Two curves are presented for comparison purposes: one with the optimal MPL (2) and one with the worst MPL (1). Then, a third curve is presented for the middleware with both dynamic load balancing and adaptive MPL. As can be seen in Fig. 8.a, the combination of local and global adaptation outperforms the load balancing with the worst fixed MPL and is very close to throughput of a fixed MPL of 2. Fig. 8.b shows that this quasi-optimal throughput is achieved without degrading the response time. The response time of the combined global and local adaptation is the same as the one of global adaptation with a fixed MPL of 2.

[4] Notice that with a load of 200 tps the system is thrashing since the achieved throughput is slightly below 200 tps. See Fig. 6.b



(a) Throughput



(b) Response time

Fig. 8. Experiment 2.3: Local+Global Adaptation vs. Global adaptation

6 Related Work

Jim Gray's paper [9] stated that traditional textbook database replication algorithms providing 1-copy-serializabilty, were not scalable. This paper, instead of closing down the area of consistent replication, opened new lines of research. The idea was to reduce the communication to the minimum needed to enforce 1-copy-serializability. Some of these efforts can be found in [1, 2, 4, 5, 7, 8, 12, 13, 16, 17, 23, 25, 27, 19, 26]. Most of the approaches implement eager replication, although a few are based on lazy replication [23, 26]. However, both of these works address the problem of inconsistencies in lazy replication. [23] provides freshness guarantees whilst [26] enforces a level of consistency similar to eager replication. Replication at the middleware level can perform symmetric processing of updates [1, 2, 7, 27, 19] or asymmetric processing [14, 26] depending on whether update transactions are run at all replicas or they are run at a single replica while the rest of them just install the resulting updates. If an update transaction performs many reads in order to update a few tuples, the amount of work saved at the rest of the replicas can be considerable. Symmetric processing can work with any database but at the cost of an inherent limited scalability. The asymmetric processing approach requires two services to get the updates from a transaction and to install them at a different replica. These services can be implemented on top of the functionality provided by commercial databases (black box approach) such as Microsoft SQL Server (such as triggers or specialized APIs) or they can be implemented within the database (gray box approach) as it has been done with PostgreSQL for this paper.

Adaptation is receiving a growing attention since the autonomic computing vision from IBM [18]. [22] uses adaptation at different levels in a distributed middleware supporting web services. The system administrator can define utility functions that are used by the adaptive middleware to guide its decisions. This middleware provides load control, connection load balancing, and admission control. A lot of attention is paid to service differentiation that provides some QoS guarantees for different kinds of clients.

In the area of databases most work has concentrated on implementing adaptation within databases. [28] summarizes the work performed in the last years around self-tuning of memory management for data servers. Advances in this area include predictive local caching and distributed caching.

Load control has been traditionally static requiring tuning by an experienced administrator [20]. Adaptation can be used to perform this tuning automatically. In [20] adaptive load control is used to prevent data contention thrashing. The undertaken approach is enacted by monitoring the transaction conflict rate and reducing the degree of concurrency when conflicts go beyond a given threshold.

[11] uses feedback control to determine the optimal MPL independently of its nature, data or memory contention. A simulation is performed to study different strategies to provide load control. Their conclusion is that adaptation can improve notably the performance of a centralized database system under overloads. Our approach for load control extends the parabola approximation method presented in [11] in that it is able to work at the middleware level, and provides performance results of a real implementation. [6] also analyzes a feedback driven approach for determining the optimal MPL. At the same time the authors attempt to find the optimal main memory allocation for a transaction type within the database system. As such, the approach can only be applied within the database kernel.

[29] introduces a dynamic replication scheme in which the location and number of object replicas is changed dynamically depending on the access pattern. The algorithms minimize the communication overhead introduced by remote accesses by locating replicas of the accessed objects close to clients. C-JDBC [7] and [3] present a database replication middleware that performs symmetric replication. They assign incoming new queries to replicas according to three different policies: round-robin, weighted round-robin, the replica with the fewest pending queries, or the replica that recently answered a query accessing the same tables.

7 Conclusions

Database replication at the middleware level has attracted a lot of attention in the last years. One of the goals of replication is to increase the system throughput. That is, the more replicas the system has, the higher the throughput. However, if the system is not carefully tuned, the expected throughput increase will not occur.

In this paper we have shown that there are a number of factors to take into account in order to tune a replicated database. These factors include the load and workload in the system. Since these parameters typically change dynamically, the system should be able to adapt itself to the new configuration in order to maximize its throughput. We combine automatic adaptation at two levels. Local adaptation limits the number of concurrent transactions according to the workload type (but without knowing the workload type). Global adaptation performs load balancing to distribute evenly the load (queries and updates) among replicas. The conducted performance evaluation has shown that the proposed dynamic adaptation is able to achieve a throughput close to the optimal without disrupting response time.

References

1. Y. Amir and C. Tutu. From Total Order to Database Replication. In *Proc. of Int. Conf. on Distr. Comp. Systems (ICDCS)*, July 2002.
2. C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed Versioning: Consistent Replication for Scaling Back-End Databases of Dynamic Content Web Sites. In *Proc. of Middleware 03*.

3. C. Amza, A. L. Cox, and W. Zwaenepoel. Scaling and Availability for Dynamic Content Web Sites. Technical Report TR-02-395, Rice University, 2002.
4. T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool. Replication, Consistency, and Practicality: Are These Mutually Exclusive? In *ACM SIGMOD Conference*, 1998.
5. Y. Breitbart and H. F. Korth. Replication and Consistency: Being Lazy Helps Sometimes. In *Proc. of the Principles of Database Systems Conf.*, pages 173–184, 1997.
6. K. P. Brown, M. Mehta, M. J. Carey, and M. Livny. Towards Automated Performance Tuning For Complex Workloads. In *Proc. of 20th VLDB*, 1994.
7. E. Cecchet, J. Marguerite, and W. Zwaenepoel. RAIDb: Redundant Array of Inexpensive Databases. Technical Report Technical Report 4921, Inria, 2003.
8. S. Gancarski, H. Naacke, E. Pacitti, and P. Valduriez. Parallel Processing with Autonomous Databases in a Cluster System. In *Proc. of CoopIS/DOA/ODBASE*, pages 410–428, 2002.
9. J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proc. of the SIGMOD*, pages 173–182, Montreal, 1996.
10. M. Hayden. The Ensemble System. Technical Report TR-98-1662, Department of Computer Science. Cornell University, Jan. 1998.
11. H. Heiss and R. Wagner. Adaptive Load Control in Transaction Processing Systems. In *Proc. of 17th VLDB*, 1991.
12. J. Holliday, D. Agrawal, and A. E. Abbadi. The Performance of Database Replication with Group Communication. In *Int. Symp. on Fault-tolerant Computing Systems*, 1999.
13. R. Jiménez-Peris, M. Patiño-Martínez, and G. Alonso. Non-Intrusive, Parallel Recovery of Replicated Data. In *IEEE Symp. on Reliable Distributed Systems (SRDS)*, 2002.
14. R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Scalable Database Replication Middleware. In *Proc. of IEEE Int. Conf. on Distributed Computing Systems*, 2002.
15. R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Are Quorums an Alternative for Data Replication. *ACM Transactions on Databases*, 28(3):257–294, Sept. 2003.
16. B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, A new way to implement Database Replication. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, 2000.
17. B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM TODS*, 25(3):333–379, Sept. 2000.
18. J. Kephart and D. Chess. The vision of autonomic computing. *IEEE Computer*, Jan. 2003.
19. A. Kistijantoro, G. Morgan, S. K. Shrivastava, and M. Little. Component Replication in Distributed Systems: A Case Study Using Enterprise Java Beans. In *Proc. of SRDS*, 2003.
20. A. Moenkeberg and G. Weikum. Performance Evaluation of an Adaptive and Robust Load Control Method for the Avoidance of Data Contention Trashing. In *Proc. of VLDB*, 1992.
21. Oracle. *Oracle 8 (tm) Server Replication*. 1997.
22. G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef. Performance Management for Cluster Based Web Services. Technical report, IBM Technical Report, 2003.
23. E. Pacitti, P. Minet, and E. Simon. Replica Consistency in Lazy Master Replicated Databases. *Distributed and Parallel Databases*, 9(3):237–267, 2001.
24. M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable Replication in Database Clusters. In *Proc. of Distributed Computing Conf.(DISC)*, 2000.
25. F. Pedone, R. Guerraoui, and A. Schiper. Exploiting Atomic Broadcast in Replicated Databases. Technical report, Département d'Informatique, EPFL, 1996.
26. C. Plattner and G. Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In *Proc. of Middleware*, 2004.
27. L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. Strong Replication in the GlobData Middleware. In *Proc. Work. on Dependable Middleware-Based Systems*, 2002.
28. G. Weikum, A. Christian, A. Kraiss, and M. Sinnwell. Integrating Snapshot Isolation into Transactional Federations. *Data Engineering Bulletin*, 22(2), 1999.
29. O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(2):255–314, 1997.