Conflict-Aware Load-Balancing Techniques for Database Replication

Vaidė Zuikevičiūtė
University of Lugano (USI)
CH-6904 Lugano, Switzerland
vaide.zuikeviciute@lu.unisi.ch

Fernando Pedone
University of Lugano (USI)
CH-6904 Lugano, Switzerland
fernando.pedone@unisi.ch

ABSTRACT

Middleware-based database replication protocols are more portable and flexible than kernel-based protocols, but have coarser-grain information about transaction access data, resulting in reduced concurrency and increased aborts. This paper proposes conflict-aware load-balancing techniques to increase the concurrency and reduce the abort rate of middleware-based replication protocols. Experimental evaluation using a prototype of our system running the TPC-C benchmark showed that aborts can be reduced with no penalty in response time.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed databases; H.2.4 [**Systems**]: Transaction processing; Distributed databases

General Terms

Performance, Design

Keywords

Replication, load balancing, transactions scheduling

1. INTRODUCTION

Database replication protocols can be classified as kernel- or middleware-based, according to whether changes in the database engine are required or not. Kernel-based protocols take advantage of internal components of the database to increase performance in terms of throughput, scalability, and response time. For the sake of portability and heterogeneity, however, replication protocols should be independent of the underlying database management system. As a consequence, middleware-based database replication has received much attention in the last years (e.g., [4, 5, 7, 9, 10, 12, 13]). The downside of the approach is that middleware-based database replication protocols usually have limited information about the data accessed by the transactions, and may result in reduced concurrency or increased abort rate or both.

This paper focuses on load-balancing techniques for certification-based replication protocols placed at the middleware layer. In such protocols, each transaction is first executed locally on some server. During the execution there is no synchronization between servers. At commit time, update transactions are broadcast to all replicas for certification. The certification test is deterministic and executed by each server. If the transaction passes certification, its updates are applied to the server's database. If two conflicting transactions execute concurrently on distinct servers, one of them may be aborted during certification to ensure strong consistency (e.g., one-copy serializability). Our load-balancing techniques build on two simple observations: (a) If conflicting transactions are submitted to the same server, the local replica's scheduler serializes the conflicting operations appropriately, reducing aborts. (b) In the absence of conflicts, however, performance is improved if transactions execute concurrently on different replicas.

Ideally, we would like the load balancer to both *minimize* the number of conflicting transactions executing on distinct replicas and *maximize* the parallelism between transactions, but unfortunately these are often opposite requirements. For example, concentrating conflicting transactions on a few replicas will reduce the abort rate, but it may overload some replicas and leave others idle. We address the problem by introducing conflict-aware load-balancing: a hybrid technique that strives to address the two requirements above.

We analyze the proposed technique experimentally using the Database State Machine replication [11] running the TPC-C benchmark. The results show that scheduling transactions with the sole goal of maximizing parallelism already doubles the throughput obtained with random assignment of transactions. Scheduling transactions in order to minimize conflicts only can reduce aborts due to lack of synchronization, but the improvements are obtained at the expense of an increase in response time since the load balancing is unfair. A hybrid approach allows to trade even load distribution for low transaction aborts in order to increase throughput with no degradation in response time.

2. BACKGROUND

2.1 Model

We consider an asynchronous distributed system composed of database clients, $c_1, c_2, ..., c_m$, and servers, $S_1, S_2, ..., S_n$. Communication is by message passing. Servers can also interact by means of a total-order broadcast, described below. Servers can fail by crashing and subsequently recover. If a server crashes and never recovers, then operational servers eventually detect the crash.

Total-order broadcast is defined by the primitives broadcast(m) and deliver(m), and guarantees that (a) if a server delivers a message m then every server delivers m; (b) no two servers deliver any two messages in different orders; and (c) if a server broadcasts message m and does not fail, then every server eventually delivers m.

Each server has a full copy of the database and executes transactions according to strict two-phase locking (2PL)[3]. The database *workload* is composed of a set of pre-defined parameterized transactions $\mathcal{T} = \{T_1, T_2, ...\}$. To account for the computational resources needed to execute different transactions, each transaction T_i in the workload can be assigned a *weight* w_i. For example, simple transactions could have less weight than complex transactions.

2.2 Database state-machine replication

The state-machine approach is a non-centralized replication technique [14]. Its key concept is that all replicas receive and process the same sequence of requests in the same order, ensured by total-order broadcast. Consistency is guaranteed if replicas behave deterministically.

The Database State Machine (DBSM) [11] uses the state-machine approach to implement deferred update replication. Each transaction is executed locally on some server and during the execution there is no interaction between replicas. Read-only transactions are committed locally. Update transactions are broadcast to all replicas for certification. If the transaction passes certification, it is committed; otherwise it is aborted. Certification ensures that the execution is *one-copy serializable* (1SR) [3].

We say that two transactions T_i and T_j *conflict*, denoted $T_i \sim T_j$, if they access some common data item, and at least one transaction writes it. If T_i and T_j conflict and are executed concurrently on different servers, certification will abort one of them. If they execute on the same replica, however, the replica's local scheduler will order T_i and T_j appropriately, and thus, both can commit.

3. CONFLICT-AWARE LOAD BALANCING

In the DBSM transactions can execute at any server. If transactions with similar access patterns execute on the same server, the local replica's scheduler will serialize conflicting transactions and decrease the number of aborts. Thus, instead of randomly choosing replicas for transaction execution, we assign transactions to *preferred servers* based on the transaction types, their parameters, and their conflict relation. As a consequence, we reduce the number of certification aborts.

Assigning transactions to preferred servers is an optimization problem. It consists in distributing the transactions over the replicas $S_1, S_2, ..., S_n$. When assigning transactions to database servers, we aim at (a) minimizing the number of conflicting transactions executing at distinct replicas, and (b) maximizing the parallelism between transactions. If the workload is composed of many conflicting transactions and the load over the system is high, then (a) and (b) become opposite requirements: While (a) can be satisfied by concentrating transactions on few database servers, (b) can be fulfilled by spreading transactions on multiple replicas. But if only few transactions conflict, then maximizing parallelism becomes the priority.

3.1 MPF and MCF

We propose a hybrid load balancing technique which allows to give more or less significance to minimizing conflicts or maximizing parallelism. We call it *Maximizing Parallelism First*(MPF). MPF prioritizes parallelism between transactions. Consequently, it initially tries to assign transactions in order to keep the servers' load even. If more than one option exists, the algorithm attempts to minimize conflicts.

The load of a server is given by the aggregated weight of the transactions assigned to it at some given time. To compare the load of two servers, we use factor $f, 0 < f \leq 1$. Servers S_i and S_j have similar load at time t if the following condition holds: $f \leq w(S_i,t)/w(S_j,t) \leq 1$ **or** $f \leq w(S_j,t)/w(S_i,t) \leq 1$. For example, MPF with $f = 0.5$ allows the difference in load between two replicas to be up to 50%. We denote MPF with a factor f as MPF f. MPF works as follows:

1. Consider replicas S_1, S_2, ..., S_n. To assign each transaction T_i in the workload to some server at time t execute steps 2–4, if T_i is an update transaction, or step 5, if T_i is a read-only transaction.
2. Let $W(t) = \{S_k \mid w(S_k,t) * f \leq \min_{l \in 1..n} w(S_l,t)\}$ be the set of replicas with the lowest aggregated weight $w(S_l,t)$ at time t, where $w(S_l,t) = \sum_{T_j \in S_l^t} w_j$.
3. If $|W(t)| = 1$ then assign T_i to the replica in $W(t)$.
4. If $|W(t)| > 1$ then let $C_W(T_i,t)$ be the set of replicas containing conflicting transactions with T_i in $W(t)$: $C_W(T_i,t) = \{S_k \mid S_k \in W(t)$ and $\exists T_j \in S_k$ such that $T_i \sim T_j\}$.
   (a) If $|C_W(T_i,t)| = 0$, assign T_i to the S_k in $W(t)$ with the lowest aggregated weight $w(S_k,t)$.
   (b) If $|C_W(T_i,t)| = 1$, assign T_i to the replica in $C_W(T_i,t)$.
   (c) If $|C_W(T_i,t)| > 1$, assign T_i to the replica S_k in $C_W(T_i,t)$ with the highest aggregated weight of transactions conflicting with T_i; if several replicas in $C_W(T_i,t)$ satisfy this condition, assign T_i to any of these.
   More formally, let $C_{T_i}(S_k^t)$ be the subset of S_k^t containing conflicting transactions with T_i only: $C_{T_i}(S_k^t) = \{T_j \mid T_j \in S_k^t \wedge T_j \sim T_i\}$. Assign T_i to the replica S_k in $C_W(T_i,t)$ with the greatest aggregated weight $w(C_{T_i}(S_k^t)) = \sum_{T_j \in C_{T_i}(S_k^t)} w_j$.
5. Assign read-only transaction T_i to the replica S_k with the lowest aggregated weight $w(S_k,t)$ at time t.

The choice of f depends heavily on workload characteristics: the number of conflicting transactions, their complexity and the load over the system.

We call *Minimizing Conflicts First*(MCF) a special case of MPF with a factor $f = 0$. MCF attempts to minimize the number of conflicting transactions assigned to different replicas. The algorithm initially tries to assign each transaction T_i in the workload to the replica containing conflicting transactions with T_i. If there are no conflicts, the algorithm tries to balance the load among the replicas.

3.2 A simple example

Consider a simple example with the workload of 10 transactions, $T_1, T_2, ..., T_{10}$, running in a system with 4 replicas. Transactions with odd index conflict with transactions with odd index; transactions with even index conflict with transactions with even index. Each transaction T_i has weight $w(T_i) = i$. All transactions are submitted concurrently to the system and the load balancer processes them in decreasing order of weight.

MPF 1 will assign transactions T_{10}, T_3, and T_2 to S_1; T_9, T_4, and T_1 to S_2; T_8 and T_5 to S_3; and T_7 and T_6 to S_4. MCF will assign $T_{10}, T_8, T_6, T_4, T_2$ to S_1; T_9, T_7, T_5, T_3, T_1 to S_2; and no transactions to S_3 and S_4. MPF 0.8 will assign T_{10}, T_4, and T_2 to S_1; T_9 and T_3 to S_2; T_8 and T_6 to S_3; and T_7, T_5, and T_1 to S_4. MPF 1 creates a balanced assignment of transactions: $w(S_1) = 15, w(S_2) = 14, w(S_3) = 13$, and $w(S_4) = 13$. Conflicting transactions are assigned to all servers however. MCF completely concentrates conflicting transactions on distinct servers, S_1 and S_2, but the aggregated weight distribution is poor: $w(S_1) = 30, w(S_2) = 25, w(S_3) = 0$, and $w(S_4) = 0$, that is, two replicas would be idle. MPF 0.8 is a compromise between the previous schemes. Even transactions are assigned to S_1 and S_3, and odd transactions to S_2 and S_4. The aggregated weight is fairly balanced: $w(S_1) = 16, w(S_2) = 12, w(S_3) = 14$, and $w(S_4) = 13$.

4. ANALYSIS OF THE TPC-C BENCHMARK

In this section we show how the TPC-C benchmark can be mapped to our transactional model, and provide a detailed analysis of MCF and MPF when applied to TPC-C.

4.1 Overview of the TPC-C benchmark

TPC-C is an industry standard benchmark for online transaction processing (OLTP) [15]. It represents a generic wholesale supplier workload. TPC-C defines five transaction types: *New Order*(NO), *Payment*(P), *Delivery*(D), *Order Status*(OS) and *Stock Level*(SL). Since only update transactions count for conflicts—read-only transactions execute at preferred servers just to balance the load—there are only three types to consider: D, P and NO. These three transaction types compose 92% of TPC-C workload. We define the workload of update transactions as:

$$\mathcal{T} = \{D_i, P_{ijkm}, NO_{ijS} \mid \begin{array}{l} i,k \in 1..\#\text{WH}; j,m \in 1..10; \\ S \subseteq \{1, ..., \#WH\}\} \end{array}$$

where #WH is the number of warehouses. D_i stands for a *Delivery* transaction accessing districts in warehouse i. P_{ijkm} relates to a *Payment* transaction which reflects the payment and sales statistics on district j and warehouse i and updates the customer's balance. In 15% of the cases, the customer is chosen from a remote warehouse k and district m. Thus, for 85% of transactions of type P_{ijkm}: $(k = i) \wedge (m = j)$. NO_{ijS} is a *New Order* transaction referring to a customer assigned to warehouse i and district j. For an order to complete, some items must be chosen: 99% of the time the item chosen is from the home warehouse i and 1% of the time from a remote warehouse. S represents a set of remote warehouses.

To assign a particular update transaction to a replica, we have to analyze the conflicts between transaction types. Throughout the paper, our analysis is based on the warehouse and district IDs only. We define the conflict relation $\sim$ between transaction types as follows:

$$\begin{aligned} \sim \; = \; & \{(D_i, D_x) \mid (x = i)\} \cup \\ & \{(D_i, P_{xykm}) \mid (k = i)\} \cup \\ & \{(D_i, NO_{xyS}) \mid (x = i)\} \cup \\ & \{(P_{ijkm}, P_{xyzq}) \mid (x = i) \vee ((z = k) \wedge (q = m))\} \cup \\ & \{(NO_{ijS}, NO_{xyZ}) \mid \\ & ((x = i) \wedge (y = j)) \vee (S \cap Z \neq \emptyset)\} \cup \\ & \{(NO_{ijS}, P_{xyzq}) \mid (x = i) \vee ((z = i) \wedge (q = j))\} \end{aligned}$$

For instance, two *Delivery* transactions conflict if they access the same warehouse.

4.2 Scheduling TPC-C

We are interested in the effects of our conflict-aware load-balancing algorithms when applied to the TPC-C workload. For illustrative purposes in this section we present a static analysis of the benchmark. We analyze the behavior of our algorithms as if all TPC-C transaction types are submitted to the system simultaneously. To keep our characterization simple, we will assume that the weights associated with the workload represent the frequency in which transactions of some type may occur in a run of the benchmark.

We studied the load distribution over the servers and the number of conflicting transactions executing on different replicas. To measure the load, we use the aggregated weight of all transactions assigned to the replica. To measure the conflicts, we use the *overlapping ratio* $O_R(S_i, S_j)$ between database servers S_i and S_j, defined as the ratio between the aggregated weight of update transactions assigned to S_i that conflict with transactions assigned to S_j, and the aggregated weight of all update transactions assigned to S_i.

We have considered 4 warehouses (i.e., #WH = 4) and 8 database replicas in our analysis. The load balancer processes the requests sequentially in a random order. We compared the behavior of MCF, MPF 1 and MPF 0.5 with a random assignment of transactions to replicas (dubbed Random). The results are presented in Figure 1.

Random and MPF 1(not shown in the graphs) behaves similarly and results in a fair load distribution but has very high overlapping ratio. MCF minimizes significantly the number of conflicts, but update transactions are distributed over 4 replicas only; the other 4 replicas execute just read-only transactions. This is a consequence of TPC-C and the 4 warehouses considered. A compromise between maximizing parallelism and minimizing conflicts can be achieved by varying the f factor of the MPF algorithm. With $f = 0.5$ the overlap ratio is much lower than Random (and MPF 1).

5. PERFORMANCE RESULTS

We have built a prototype of the DBSM in Java 1.5. The experiments were run in a cluster of Apple Xservers equipped with a dual 2.3 GHz PowerPC G5 (64-bit) processor, 1GB RAM, and an 80GB HDD. Each server runs Mac OS X Server 10.4. The servers are connected through a switched 1Gbps Ethernet LAN. We used MySQL 5.0 with InnoDB storage engine as our database server configured to run transactions at the serializable isolation level. Each server stores a TPC-C database, populated with data for 8 warehouses. In all experiments clients submit transactions as soon as the response of the previously issued transaction is received. The assignment of transactions is computed on-the-fly based on currently executing transactions at the

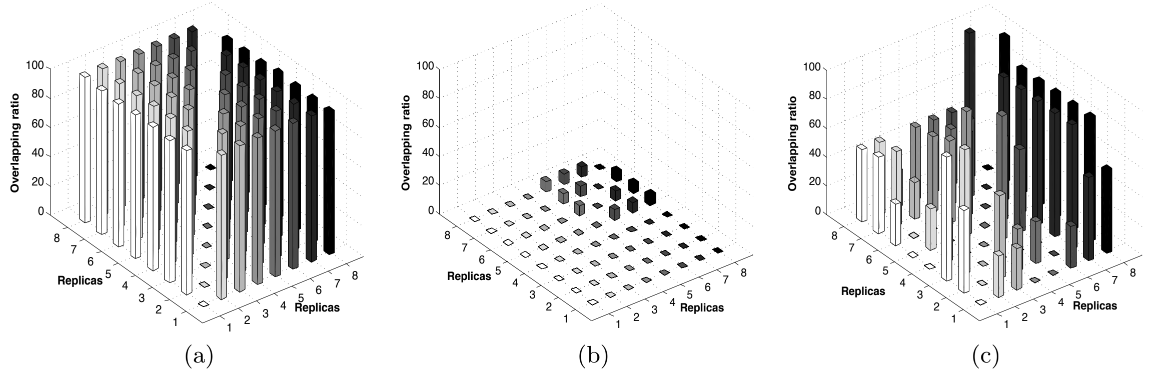

Figure 1: Overlapping ratio, (a) Random (b) MCF (c) MPF 0.5

replicas. Our load balancer is lightweight: CPU usage at the load balancer is less than 4% throughout all experiments.

5.1 Throughput and response time

Following TPC-C, we present the results based only on *New Order* transactions, which represent $\approx$ 45% of total workload. *Payment* transactions, accounting for another 43% of the workload, perform analogously. We report results of MPF with $f = 1$ and $f = 0.5$ as the most representative for the setup considered. We further assume that all transactions in the workload have the same weight.

Figures 2(a) and 2(b) show the achieved throughput and response time of committed *New Order* transactions on a system with 4 replicas under various load conditions. MCF, which primarily takes conflicts into consideration, suffers from poor load distribution over the replicas and fails to improve the throughput. Even though the aborts due to lack of synchronization are reduced significantly, the response time grows fast. Response time increases as a consequence of all conflicting transactions executing on the same replica and competing for the locks on the same data items. Prioritizing parallelism (MPF 1) doubles the achieved throughput when compared to Random. Although Random assigns transactions equitably to all replicas, differently from MPF 1, it does not account for the various execution times of transactions. Under light loads MPF 1 and a hybrid load-balancing technique, such as MPF 0.5, which considers both conflicts between transactions and the load over the replicas, demonstrate the same benefits in performance. If the load is low, few transactions will execute concurrently and minimizing the number of conflicting transactions executing at distinct replicas becomes less effective. However, once the load is increased, MPF 0.5 clearly outperforms MPF 1.

5.2 Abort rate breakdown

To analyze the effects of conflict-awareness we present a breakdown of abort rate. There are four main reasons for a transaction to abort: (i) it fails the certification test, (ii) it holds locks that conflict with a committing transaction, (iii) it times out after waiting for too long to obtain a lock, and (iv) it is aborted by the database engine to resolve a deadlock. Notice that aborts due to conflicts are similar in nature to certification aborts, in that they both happen due to the lack of synchronization between transactions during the execution. Thus, a transaction will never be involved in aborts of type (i) or (ii) due to another transaction executing on the same replica.

Figure 2(c) shows the abort rate breakdown for each technique; each vertical bar per technique represents different submitted load. Random and MPF 1 lead to aborts mainly due to conflicts and certification, whereas aborts in MCF are primarily caused by timeouts. Due to better precision in load balancing and conflict-awareness, MPF 1 also results in lower abort rates when compared to Random. MCF successfully reduces the number of aborts due to lack of synchronization. However, increasing the load results in many timeouts caused by conflicting transactions competing for locks. MPF 0.5, which tries to account for both conflicts and load, reduces the number of aborts from $\approx$ 40% to $\approx$ 11%.

6. RELATED WORK

We focus on related work in the area of database replication where some form of load balancing techniques are used.

In [8] the authors introduce a two-level dynamic adaptation for replicated databases: at the local level the algorithms strive to maximize performance of a local replica by taking into account the load and the replica's throughput to find the optimum number of transactions that are allowed to run concurrently within a database system; at the global level the system tries to distribute the load over all the replicas considering the number of active transactions and their execution times. Differently from our approach, this work does not consider transaction conflicts for load balancing.

In [6] the authors propose a load balancing technique that takes into account transactions memory usage. Transactions are assigned to replicas in such a way that memory contention is reduced. To lower the overhead of updates propagation in a replicated system, the authors also present a complementary optimization called update filtering. Replicas only apply the updates that are needed to serve the workload submitted, i.e., transaction groups are partitioned across replicas. Differently from ours, the load balancer in [6] doesn't consider conflicts among transactions. Further, if workload characteristics change, the assignment of transaction groups to replicas requires complex reconfiguration, which is limited if update filtering is used. On the contrary, our load-balancing decisions are made per transaction.

Clustered JDBC (C-JDBC) [4] uses round-robin, weighted round-robin or least pending requests first for transactions scheduling to the database replicas. Similarly, the Ganymed

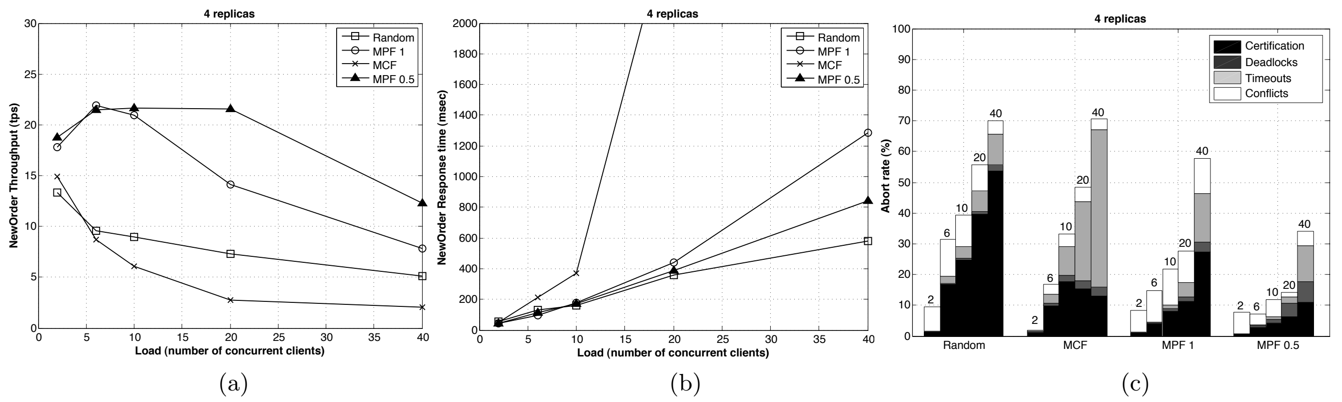

Figure 2: (a) throughput and (b) response time of *New Order* transactions, (c) abort rates; 4 replicas

scheduler [12] assigns read-only transactions to the slave replicas according to least pending requests first rule. However, none of the approaches exploit transaction conflicts information.

A thorough study of load balancing and scheduling strategies is performed in [2]. Conflict-aware scheduling [1] is the winning technique of all considered. The scheduler is extended to include conflict awareness in the sense that requests are scheduled to replicas that are up-to-date. Unlike in our load balancing techniques, conflict awareness is at a coarse granularity, i.e., table. Further, if the scheduler fails, the system needs to deal with a complicated recovery procedure to continue functioning correctly; whereas in our approach the load balancer is independent of the system's correctness—even if the load-balancer fails, transactions can execute at any replica without hampering consistency.

7. FINAL REMARKS

To keep low abort rate despite the coarse granularity of middleware-based replication protocols, we introduced conflict-aware load-balancing techniques that attempt to reduce the number of conflicting transactions executing on distinct database servers and seek to increase the parallelism among replicas. MCF concentrates conflicting transactions on a few replicas reducing the abort rate, but leaves many replicas idle and overloads others; MPF with the sole goal of maximizing parallelism distributes the load over the replicas, but ignores conflicts among transactions. A hybrid approach allows database administrators to trade even load distribution for low transaction aborts in order to increase throughput with no degradation in response time.

8. REFERENCES

[1] C. Amza, A. Cox, and W. Zwaenepoel. Conflict-Aware Scheduling for Dynamic Content Applications. In *Proceedings of USITS*, March 2003.

[2] C. Amza, A. L. Cox, and W. Zwaenepoel. A comparative evaluation of transparent scaling techniques for dynamic content servers. In *Proceedings of IEEE ICDE*, pages 230–241, April 2005.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[4] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *Proceedings of ATEC, Freenix track*, pages 9–18, June 2004.

[5] A. Correia, A. Sousa, L. Soares, J.Pereira, F. Moura, and R. Oliveira. Group-based replication of on-line transaction processing servers. In *LADC*, pages 245–260, October 2005.

[6] S. Elnikety, S. Dropsho, and W. Zwaenepoel. Tashkent+: Memory-aware load balancing and update filtering in replicated databases. In *Proceedings of EuroSys*, pages 399–412, March 2007.

[7] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Proceedings of ACM SIGMOD*, pages 419–430, June 2005.

[8] J. M. Milán-Franco, R. Jiménez-Peris, M. Patiño-Martínez, and B. Kemme. Adaptive middleware for data replication. In *Proceedings of Middleware*, pages 175–194, October 2004.

[9] F. D. Muñoz-Escoí, J. Pla-Civera, M. I. Ruiz-Fuertes, L. Irún-Briz, H. Decker, J. E. Armendáriz-Iñigo, and J. R. G. de Mendívil. Managing transaction conflicts in middleware-based database replication architectures. In *Proceedings of IEEE SRDS*, 2006.

[10] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Consistent Database Replication at the Middleware Level. *ACM Transactions on Computer Systems*, 23(4):375–423, 2005.

[11] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Journal of Distributed and Parallel Databases and Technology*, 14:71–98, 2003.

[12] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *Proceedings of Middleware*, pages 155–174, October 2004.

[13] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. The GlobData fault-tolerant replicated distributed object database. In *Proceedings of EurAsia-ICT*, pages 426–433, October 2002.

[14] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[15] Transaction Proccesing Performance Council (TPC). TPC benchmark C. Standard Specification, 2005.