Conflict-Aware Load-Balancing Techniques for Database Replication

Vaidė Zuikevičiūtė Fernando Pedone

Faculty of Informatics
University of Lugano
6900 Lugano, Switzerland

University of Lugano
Faculty of Informatics
Technical Report No. 2006/01
July 2006

Abstract

Middleware-based database replication protocols require few or no changes in the database engine. As a consequence, they 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. Our algorithms strive to assign transactions to replicas so that the number of conflicting transactions executing on distinct servers is reduced and the load over the servers is equitably distributed. A performance evaluation using a prototype of our system running the TPC-C benchmark shows that aborts can be reduced with no penalty in response time.

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. Even if the database internals are accessible, modifying them is usually a complex operation. As a consequence, middleware-based database replication has received a considerable amount of attention in the last years [4, 6, 11, 13, 19, 20, 26]. Such solutions can be maintained independently of the database engine, and can potentially be used in heterogeneous settings. The downside of the approach is that middleware-based database replication protocols usually have limited information about the data accessed by the transactions, and 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 server's local 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.

Designing load-balancing techniques that exploit observations (a) and (b) efficiently is not trivial. Concentrating conflicting transactions on a few replicas will reduce the abort rate, but may leave many replicas idle and overload the others. Ideally, we would like to both *minimize* the number of conflicting transactions executing on distinct replicas and *maximize* the parallelism between transactions. But these are opposite requirements. We address the problem by introducing two conflict-aware load-balancing algorithms: Minimizing Conflicts First (MCF) and Maximizing Parallelism First (MPF). MCF and MPF strive to address the two requirements above, but each one prioritizes a different one. Both algorithms are independent of the nature of the load balancer used in the system, static or dynamic.

To demonstrate the applicability of MCF and MPF we introduce the *Multiversion Database State Machine* (vDBSM). The vDBSM is a middleware extension of the DBSM [18], a kernel-based optimistic replication protocol. We compare MCF and MPF experimentally using the vDBSM running the TPC-C benchmark. The results show that while scheduling transactions with the sole goal of maximizing parallelism (i.e., "pure" MPF) is only slightly better than randomly assigning transactions to replicas, scheduling transactions in order to minimize conflicts only (i.e., "pure" MCF) can reduce aborts due to lack of synchronization from $\approx 20\%$ to $\approx 3\%$. The improvements are obtained at the expense of an increase in the response time since the load balancing is unfair. A hybrid approach, combining MCF and MPF, reduces aborts and increases transaction throughput with no penalty in response time.

The paper's main contribution is to introduce and evaluate conflict-aware load-balancing techniques for middleware-based database replication protocols. Our techniques allow database administrators to trade even load distribution for low transaction aborts in order to increase throughput with no degradation in response time. A secondary contribution of the paper is the vDBSM, a novel middleware-based replication protocol.

2 Background

2.1 Database model

We consider a distributed system composed of database clients, $c_1, c_2, ..., c_m$, and servers, $S_1, S_2, ..., S_n$. Each server has a full copy of the database. Servers are assumed to be fail-stop: a server halts in response to a failure and its halted state can be detected by operational servers. Failed servers may resume normal execution after recovering from the failure.

Database servers execute *transactions* according to strict two-phase locking (2PL)[15]. Transactions are sequences of read and write operations followed by a commit or an abort operation. A transaction is called *read-only* if it does not contain any write operation; otherwise it is called an *update* transaction.

The database *workload* is composed of a set of transactions $\mathcal{T} = \{T_1, T_2, ...\}$. To account for the computational resources needed to execute different transactions, the database administrator can assign to each transaction T_i in the workload 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 [21]. Its key concept is that all replicas receive and process the same sequence of requests in the same order. Consistency is guaranteed if replicas behave deterministically, that is, when provided with the same input (e.g., a request) each replica will produce the same output (e.g., state change). Servers interact by means of a total-order broadcast, a group communication abstraction built on top of a lower-level message passing communication system. 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.

The Database State Machine (DBSM) [18] 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)*, that is, every concurrent execution is equivalent to some serial execution of the same transactions using a single copy of the database.

At certification, the transaction's readsets, writesets, and updates are broadcast to all replicas. The *readsets* and the *writesets* identify the data items read and written by the transactions; they do not contain the values read and written. The transaction's *updates* can be its redo logs or the rows it modified and created. All servers deliver the same transactions in the same order and certify the transactions deterministically. Notice that the DBSM does not require the execution of transactions to be deterministic; only the certification test and the application of the transaction updates to the database are implemented as a state machine.

3 Multiversion Database State Machine

In this section, we introduce the Multiversion Database State Machine (vDBSM), a middleware extension to the DBSM. A correctness proof can be found in the Appendix.

The vDBSM assumes pre-defined, parameterized transactions. Each transaction is identified by its *type* and the *parameters* provided by the application program when the transaction is instantiated. From its type and parameters, the transaction's readset and writeset can be estimated, even if conservatively, before the transaction is executed. Pre-defined transactions are common in many database applications (e.g., application server environments).

Hereafter, we denote the replica where T_i executes, its readset, and its writeset by $server(T_i)$, $readset(T_i)$, and $writeset(T_i)$, respectively. The vDBSM protocol works as follows:

1. We assign to each data item in the database a version number. Thus, besides storing a full copy of the database, each replica S_k also has a vector V_k of version numbers. The current version of data item d_x at S_k is denoted by $V_k[x]$.

2. Both read-only and update transactions can execute at any replica.

   During the execution, the versions of the data items read by an update transaction are collected. We denote by $V(T_i)[x]$ the version of each data item d_x read by T_i. The versions of the data items read by T_i are broadcast to all replicas together with its readset, writeset, and updates at commit time.

3. Upon delivery, update transactions are certified. Transaction T_i passes certification if all data items it read during its execution are still up-to-date at certification time. More formally, T_i passes certification on replica S_k if the following condition holds:

$$\forall d_x \in readset(T_i): \; V_k[x] = V(T_i)[x]$$

4. If T_i passes certification, its updates are applied to the database, and the version numbers of the data items it wrote are incremented. All replicas must ensure that transactions that pass certification are committed in the same order. How this is ensured is implementation specific and discussed in Section 6.

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 one transaction reads the item and the other writes it. If T_i and T_j conflict and are executed concurrently on different servers, certification may abort one of them. If they execute on the same replica, however, the replica's scheduler will order T_i and T_j appropriately, and thus, both can commit.

Therefore, 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. Based on the transaction types, their parameters and the conflict relation, we assign transactions to *preferred servers*.

The vDBSM ensures consistency (i.e., one-copy serializability) regardless of the server chosen for the execution of a transaction. However, executing update transactions on their preferred servers can reduce the number of certification aborts. In the next section we explain how this is done.

4 Conflict-aware load balancing

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 to (a) minimize the number of conflicting transactions in distinct replicas, and (b) maximize the parallelism between transactions. These are opposite requirements. While the first can be satisfied by concentrating transactions on few database servers, the second is fulfilled by spreading transactions on multiple replicas. In Sections 4.1 and 4.2, we present two greedy algorithms that assign transactions to preferred servers. Each one prioritizes a different requirement.

Our load-balancing algorithms can be executed *statically*, before transactions are submitted to the system, or *dynamically*, during transaction processing, for each transaction when it is submitted. Static load balancing requires knowledge of the transaction types, the conflict relation, and the weight of transactions. Dynamic load balancing further requires information about which transactions are in execution on the servers. In Sections 4.3 and 4.4 we discuss our load-balancing algorithms in each context, and in Section 4.5 we contrast them.

4.1 Minimizing Conflicts First (MCF)

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 more than one option exists, the algorithm strives to distribute the load among the replicas equitably, maximizing parallelism.

1. Consider replicas S_1, S_2, ..., S_n. With an abuse of notation, we say that transaction T_i belongs to S_k^t at time t, $T_i \in S_k^t$, if T_i is assigned at time t or before to be executed on server S_k.

2. For each transaction T_i in the workload, to assign T_i to some server at time t execute steps 3–5.

3. Let $C(T_i, t)$ be the set of replicas containing transactions conflicting with T_i at time t, defined as $C(T_i, t) = \{S_k \mid \exists T_j \in S_k^t \text{ such that } T_i \sim T_j\}$.

4. If $|C(T_i, t)| = 0$ then assign T_i to the replica S_k with the lowest aggregated weight $w(S_k, t)$ at time t, where $w(S_k, t) = \sum_{T_j \in S_k^t} w_j$.

5. If $|C(T_i, t)| \geq 1$, then assign T_i to the replica in $C(T_i, t)$ with the highest aggregated weight of transactions conflicting with T_i; if several replicas in $C(T_i, t)$ satisfy this condition, assign T_i to any one 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 $C(T_i, t)$ with the greatest aggregated weight $w(C_{T_i}(S_k^t)) = \sum_{T_j \in C_{T_i}(S_k^t)} w_j$.

4.2 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 $f, 0 < f \leq 1$. We denote MPF with a factor f as MPF f. 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** $f \leq w(S_j,t)/w(S_i,t) \leq 1$.

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.

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 minimal load at time t, where $w(S_l,t)$ has been defined in step 4 in Section 4.1.

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 S_k in $C_W(T_i,t)$ with the highest aggregated weight $w(S_k,t)$; if several replicas in $C_W(T_i,t)$ satisfy this condition, assign T_i to any one of these.

4.3 Static load balancing

A static load balancer executes MCF and MPF offline, considering each transaction in the workload at a time in some order—for example, transactions can be considered in decreasing order of weight, or according to some time distribution, if available. Since the assignments are pre-computed, during the execution there is no need for the replicas to send feedback information to the load balancer. The main drawback of this approach is that it can potentially make poor assignment decisions.

We now illustrate static load balancing of MCF and MPF. Consider a workload with 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 $w(T_i) = i$.

By considering transactions in decreasing order of weight, MCF will assign transactions T_{10}, T_8, T_6, T_4, and T_2 to S_1; T_9, T_7, T_5, T_3, and T_1 to S_2; and no transactions to S_3 and S_4. MPF 1 will assign 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. 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. The resulting scheme is such that $w(S_1) = 15, w(S_2) = 14, w(S_3) = 13$, 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 $w(S_4) = 13$.

4.4 Dynamic load balancing

Dynamic load balancing can potentially outperform static load balancing by taking into account information about the execution of transactions when making assignment choices. Moreover, the approach does not require any pre-processing since transactions are assigned to replicas on-the-fly, as they are submitted. As a disadvantage, a dynamic scheme requires feedback from the replicas with information about the execution of transactions. Receiving and analyzing this information may introduce overheads.

MCF and MPF can be implemented in a dynamic load balancer as follows: The load balancer keeps a local data structure $S[1..n]$ with information about the current assignment of transactions to each server. Each transaction in the workload is considered at a time, when it is submitted by the client, and assigned to a server according to MCF or MPF. When a replica S_k finishes the execution of a transaction T_i, committing or aborting it, S_k notifies the load balancer. Upon receiving the notification of termination from S_k, the load balancer removes T_i from $S[k]$.

4.5 Static vs. dynamic load balancing

A key difference between static and dynamic load balancing is that the former will only be effective if transactions are pre-processed in a way that resembles the real execution. For example, assume that a static assignment considers that all transactions are uniformly distributed over a period of time, but in reality some transaction types only occur in the first half of the period and the other types in the second half. Obviously, this is not an issue with dynamic load balancing.

Another aspect that distinguishes static and dynamic load balancing is membership changes, that is, a new replica joins the system or an existent one leaves the system (e.g., due to a crash). Membership changes invalidate the assignments of transactions to servers. Until MCF and MPF are updated with the current membership, no transaction will be assigned to a new replica joining the system, for example. Therefore, with static load balancing, the assignment of preferred servers has to be recalculated whenever the membership changes. Notice that adapting to a new membership is done for performance, and not consistency since the certification test of the vDBSM does not rely on transaction assignment information to ensure one-copy serializability; the consistency of the system is always guaranteed, even though out of date transaction assignment information is used.

Adjusting MCF and MPF to a new system membership using a dynamic load balancer is straightforward: as soon as the the new membership is known by the load balancer, it can update the number of replicas in either MCF or MPF and start assigning transactions correctly. With static load balancing, a new membership requires executing MCF or MPF again for the complete workload, which may take some time. To speed up the calculation, transaction assignments for configurations with different number of "virtual replicas" can be done offline. Therefore, if one replica fails, the system switches to a pre-calculated assignment with one replica less. Only the mapping between virtual replicas to real ones has to be done online.

5 Analysis of the TPC-C benchmark

5.1 Overview of the TPC-C benchmark

TPC-C is an industry standard benchmark for online transaction processing (OLTP) [24]. It represents a generic wholesale supplier workload. The benchmark's database consists of a number of warehouses, each one composed of ten districts and maintaining a stock of 100000 items; each district serves 3000 customers. All the data is stored in a set of 9 relations: *Warehouse, District, Customer, Item, Stock, Orders, Order Line, New Order,* and *History.*

TPC-C defines five transaction types: *New Order*, *Payment*, *Delivery*, *Order Status* and *Stock Level*. *Order Status* and *Stock Level* are read-only transactions; the others are update transactions. Since only update transactions require assignment to preferred servers—read-only transactions can execute on any replica—there are only three transaction types to consider: *Delivery* (D), *Payment* (P), and *New Order* (NO). In the following we define the workload of update transactions as:

$$\mathcal{T} = \{D_i, P_{ijkm}, NO_{ijn} \mid i, k, n \in 1..\#\text{WH}; j, m \in 1..10\}$$

where #WH is the number of warehouses considered. 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 the 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_{ijn} 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: 90% of the time the item chosen is from the home warehouse i and 10% of the time from a remote warehouse n. Thus, 90% of transactions of type NO_{ijn} satisfy $n = i$.

To assign a particular transaction to a replica, we have to analyze the conflicts between transaction types. Our analysis is based on the warehouse and district numbers only. For example, *New Order* and *Payment* transactions might conflict if they operate on the same warehouse. We define the conflict relation $\sim$ between transaction types as follows:

$$
\begin{aligned}
\sim \;=\; & \{(D_i, D_x) \mid (x = i)\} \cup \\
& \{(D_i, P_{xykm}) \mid (k = i)\} \cup \\
& \{(D_i, NO_{xyn}) \mid (x = i) \vee (n = i)\} \cup \\
& \{(P_{ijkm}, P_{xyzq}) \mid \\
& (x = i) \vee ((z = k) \wedge (q = m))\} \cup \\
& \{(NO_{ijn}, NO_{xyz}) \mid \\
& ((x = i) \wedge (y = j)) \vee (z = n)\} \cup \\
& \{(NO_{ijn}, P_{xyzq}) \mid (x = i) \vee ((z = i) \wedge (q = j))\}
\end{aligned}
$$

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

Notice that we do not have to consider every transaction that may happen in the workload in order to define the conflict relation between transactions. Only the transaction types and how they relate to each other should be taken into account. 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.

5.2 Statically scheduling TPC-C

We are interested in the system's load distribution and the number of conflicting transactions executing on different replicas. To measure the load, we use the aggregated weight of all update transactions assigned to each 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 update transactions assigned to S_j, and the aggregated weight of all update transactions assigned to S_i. For example, consider that T_1, T_2, and T_3 are assigned to S_i, and T_4, T_5, T_6, and T_7 are assigned to S_j. T_1 conflicts with T_4, and T_2 conflicts with T_6. Then the overlapping ratio for these replicas is calculated as $O_R(S_i, S_j) = \frac{w(T_1)+w(T_2)}{w(T_1)+w(T_2)+w(T_3)}$ and $O_R(S_j, S_i) = \frac{w(T_4)+w(T_6)}{w(T_4)+w(T_5)+w(T_6)+w(T_7)}$. Notice that since our analysis here is static, the overlapping ratio gives a measure of "potential aborts"; real aborts will only happen if conflicting transactions are executed concurrently on different servers. Clearly, a high risk of abort translates into more real aborts during the execution.

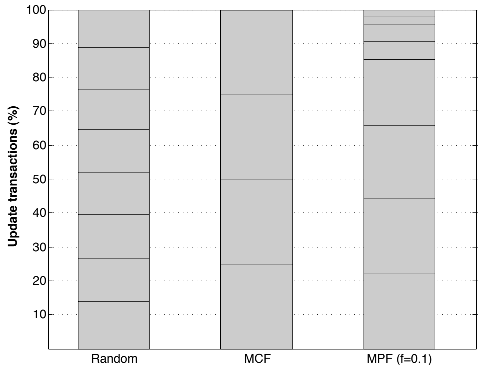


Figure 1: Load distribution over 8 replicas

We have considered 4 warehouses (i.e., #WH = 4) and 8 database replicas in our static analysis. We compared the results of MCF, MPF 1 and MPF 0.1 with a random assignment of transactions to replicas.

Random results in a fair load distribution (see Figure 1), but has very high overlapping ratio (see Figure 2). MPF 1 (not shown in the graphs) behaves similarly to Random: it distributes the load equitably over the replicas, but has high overlapping ratio.

MCF minimizes significantly the number of conflicts, but transactions are distributed over 4 replicas only. This is a consequence of TPC-C and the 4 warehouses considered. Even if more replicas were available, MCF would still strive to minimize the overlapping ratio, assigning transactions to only 4 replicas.

A compromise between maximizing parallelism and minimizing conflicts can be achieved by varying the f factor of the MPF algorithm. With $f = 0.1$ the overlap ratio is much lower than Random (and MPF 1). Finally, notice that Figure 1 shows the load of update transactions only; by carefully scheduling read-only transactions to underloaded replicas the overall load can be better balanced.

6 Prototype overview

We have implemented a prototype of the vDBSM in Java v.1.5.0 using both static and dynamic load balancing. Client applications interact with the replicated compound by submitting SQL

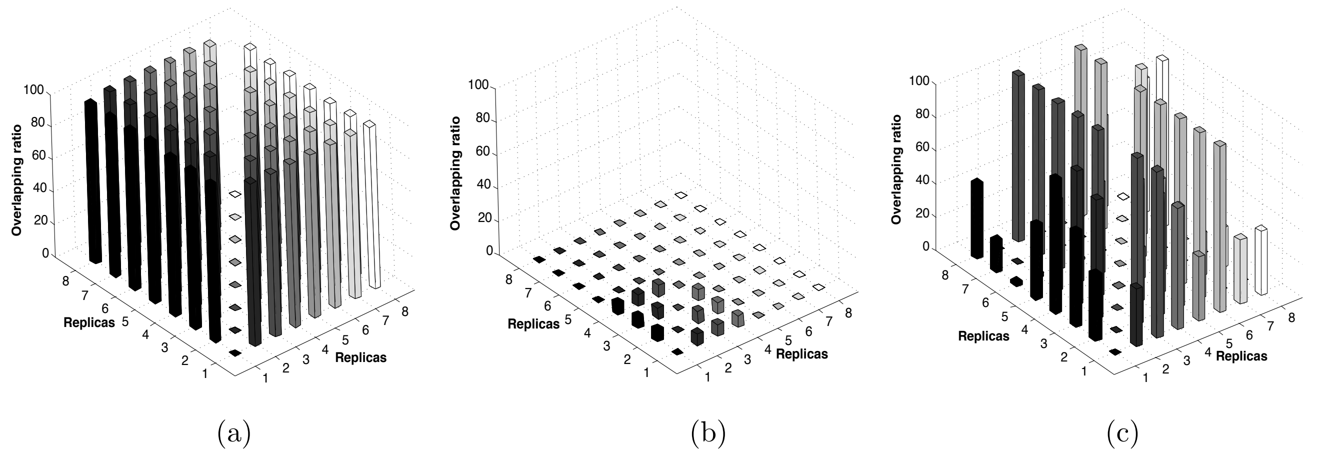


Figure 2: Overlapping ratio, (a) Random (b) MCF (c) MPF 0.1

statements through a customized JDBC-like interface. Application requests are sent directly to a database server, in case of static load balancing, or first to the load balancer and then re-directed to a server. A *replication module* in each server is responsible for executing transactions against the local database, and certifying and applying them in case of commit. Every transaction received by the replication module is submitted to the database through the standard JDBC interface. The communication between clients, replicas and the load balancer uses Unix sockets. Update transactions are broadcast to all replicas using a communication library implementing the Paxos algorithm [10].

On delivery the transaction is enqueued for certification. While transactions execute concurrently in the database, their certification and possible commitment are sequential. The current versions of the data items are kept in main memory to speed up the certification process; however, for persistency, every row in the database is extended with a version number. If a transaction passes the certification test, its updates are applied to the database and the versions of the data items written are incremented both in the database, as part of the committing transaction, and in main memory.

To ensure that all replicas commit transactions in the same order, before applying T_i's updates, the server aborts every locally executing conflicting transaction T_j. To see why this is done, assume that T_i and T_j write the same data item d_x, each one executes on a different server, T_i is delivered first, and both pass certification test. T_j already has a lock on d_x at $server(T_j)$, but T_i should update d_x first. We ensure correct commit order by aborting T_j on $server(T_j)$ and re-executing its updates later. If T_j keeps a read lock on d_x, it is a doomed transaction, and in any case it would be aborted by the certification test later.

In the case of static load balancing, the assignment of transactions to replicas is done by the replication modules and sent to the customized JDBC interface upon the first application connect. Therefore when the application submits a transaction, it sends it directly to the replica responsible for that transaction type. The dynamic load balancer is interposed between the client applications and the replication modules. The assignment of submitted transactions is computed on-the-fly based on currently executing transactions. The load balancer keeps track of each transaction's execution and completion status at the replicas. Since all application requests are routed through the load balancer, no additional information exchange is needed between replication modules and the load balancer. The load balancer does not need to know when a transaction commits at each replica, but only at the replica where the transaction was executed.

7 Performance results

7.1 Experimental setup

The experiments were run in a cluster of Apple Xservers equipped with a dual 2.3 GHz PowerPC G5 (64-bit) processor, 1GB DDR SDRAM, and an 80GB 7200 rpm disk drive. Each server runs Mac OS X Server v.10.4.4. The servers are connected through a switched 1Gbps Ethernet LAN.

We used MySQL v.5.0.16 with InnoDB storage engine as our database server. InnoDB is a transactional database engine embedded inside MySQL. It provides MySQL with transaction-safe (ACID–compliant) tables. Connections to the database were handled by MySQL Connector/J v.3.1.12 interface. The isolation level was set to serializable throughout all experiments.

We evaluated the algorithms varying the number of servers from 2 to 8. Each server stores a TPC-C database, populated with data for 4 warehouses ($\approx$ 400MB database). The workload is created by a full-fledged implementation of TPC-C. According to TPC-C, each warehouse must support 10 emulated clients, thus throughout the experiments the workload is submitted by 40 concurrent clients. TPC-C specifies that between transactions, each client should have a mean think time between 5 and 12 seconds.

Experiments have two phases: the *warm-up phase* when the load is injected but no measurements are taken, and the *measurement phase* when the data is collected.

7.2 Static vs. dynamic load balancing

Figure 3 shows the number of update transactions assigned to each server during executions of the benchmark with a static load balancer and different scheduling techniques. MCF and MPF 0.1 implemented with a static load balancer suffer from poor load distribution over the replicas. MCF distributes transactions over four replicas only, even when more replicas are available. MPF 0.1 achieves better load balancing than MCF with 6 and 8 replicas. Random and MPF 1 result in a fair load distribution.

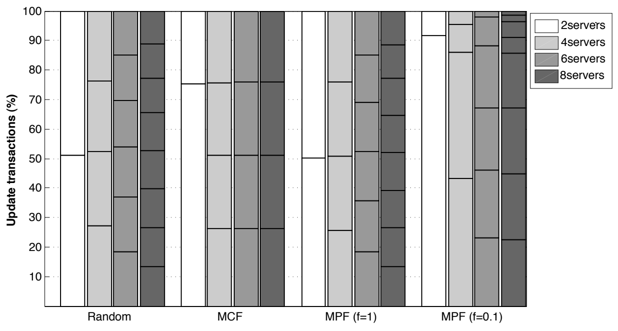


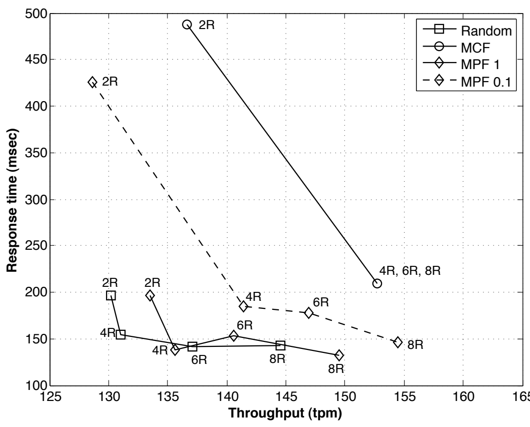
Figure 3: Real load distribution (static)

Figure 4 shows the achieved throughput of committed transactions versus the response time for both static and dynamic load balancers. For each curve in both static and dynamic schemes the increased throughput is obtained by adding replicas to the system. From both graphs, scheduling transactions based mainly on replica load, MPF 1, results in slightly better throughput than Random, while keeping the response time constant. Prioritizing conflicts has a more noticeable effect on load balancing. With static and dynamic load balancing, MCF, which primarily takes conflicts into consideration, achieves higher throughput, but at the expense of increased response times. A hybrid load-balancing technique, such as MPF 0.1, which considers both conflicts between transactions and the load over the replicas, improves transaction throughput and only slightly increases response times with respect to Random and MPF 1.

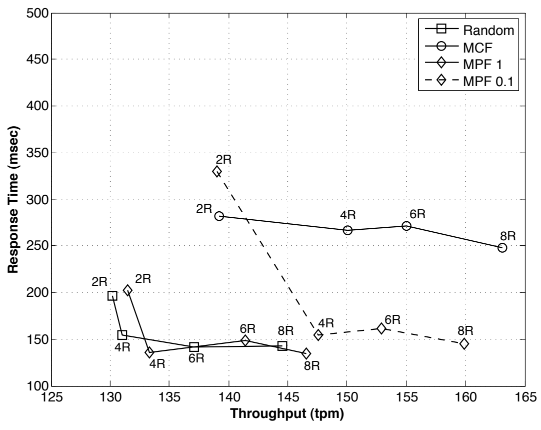
Since in static load balancing MCF uses the same transaction assignment for 4, 6 and 8 replicas (see Figure 3), the throughput does not increase by adding replicas, and that is why MCF in Figure 4 (a) only contains two points, one for 2 replicas (2R) and another one for 4, 6 and 8 replicas (4R, 6R, 8R). Static MCF strives to minimize conflicts between replicas and assigns transactions to the same 4 servers. In this case, dynamic load balancing clearly outperforms the static one, since all available replicas are used. Finally, the results also show that except for dynamic MPF 0.1, the system is overloaded with 2 servers.

7.3 Abort rate breakdown

In this section we consider the abort rate breakdown for both dynamic and static load balancing (see Figure 5). There are three main reasons for a transaction to abort: (i) it fails the certification test, (ii) it holds locks that conflict with a committing transaction (see Section 6), and (iii) it times out after waiting for too long. Notice that aborts due to conflicts are similar



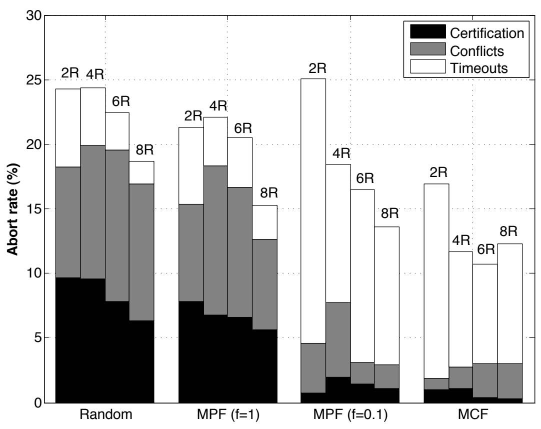
(a)



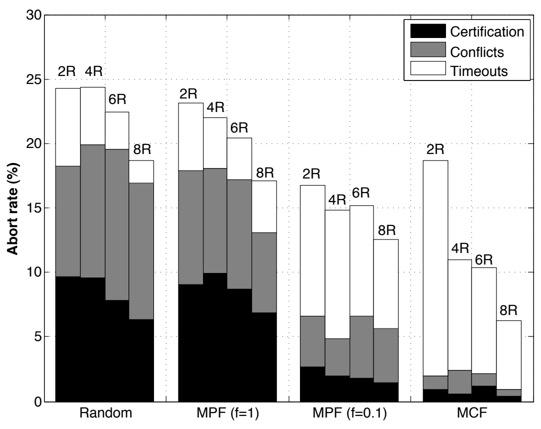
(b)

Figure 4: Throughput vs. response time (a) static load balancing (b) dynamic load balancing

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. In both static



(a)



(b)

Figure 5: Abort rates (a) static load balancing (b) dynamic load balancing

and dynamic strategies with more than 2 replicas, Random and MPF 1 result in more aborts than MCF and MPF 0.1. Random and MPF 1 lead to aborts due to conflicts and certification, whereas aborts in MCF and MPF 0.1 are primarily caused by timeouts. MCF reduces certification aborts from $\approx 20\%$ to $\approx 3\%$. However, MCF, especially with a static load balancer, results in many timeouts caused by conflicting transactions waiting for execution. MPF 0.1 with a static load balancer suffers mostly from unfair load distribution over the servers, while with a dynamic load balancing MPF 0.1 is between MPF 1 and MCF: reduced certification aborts, if compared to the former, and reduced timeouts, if compared to the latter. In the end, MCF and MPF 0.1 win because local aborts introduce lower overhead in the system than certification aborts.

7.4 The impact of failures and reconfigurations

We now consider the impact of membership changes due to failures in the dynamic load-balancing scheme. The scenario is that of a system with 4 replicas and one of them fails. Until the failure is detected, the load balancer continues to schedule transactions (using MPF 1 in this case) to all replicas, including the failed one. After 20 seconds, the time that it takes for the load balancer to detect the failure, only the operational three replicas receive transactions.

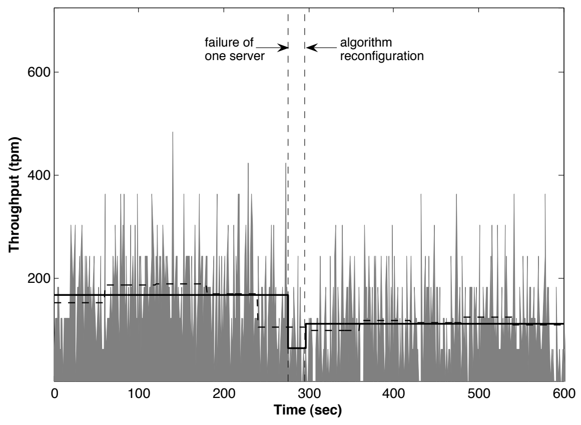


Figure 6: Algorithm reconfiguration

Figure 6 shows the replica's failure impact on committed transactions throughput with MPF 1. The solid horizontal line represents the average throughput when 4 replicas are processing transactions before the failure, during the failure, and after the failure is detected. The horizontal dashed line shows the average number of committed transactions per one-minute intervals. The throughput decreases significantly during the failure, but after the failure is detected, the load-balancing algorithm adapts to the system reconfiguration and the throughput improves. However, only 3 replicas continue functioning, so the throughput is lower than the one with 4 replicas. Our detection time of 20 seconds has been chosen to highlight the effects of failures. In practice smaller timeout values would be more adequate.

8 Related Work

In contrast to replication based on distributed locking and atomic commit protocols, group-communication-based protocols minimize the interaction between replicas and the resulting synchronization overhead. Proposed protocols differ mainly in the execution mode—transactions can be executed conservatively [9, 13] or optimistically [3, 18], and in the database correctness criteria they provide—one-copy serializability [1, 13, 14, 16, 17, 18, 20] or snapshot isolation [11, 19, 26].

The original DBSM has been previously extended in several directions. Sousa et al. investigate the use of partial replication in the DBSM [23]. In [8] the authors relax the consistency criteria of the DBSM with Epsilon Serializability. The work in [25] discusses readsets-free certification. The basic idea of the DBSM remains the same: transactions are executed locally according to strict 2PL. In contrast to the original DBSM, when an update transaction requests a commit, only its updates and writesets are broadcast to other sites. Certification checks whether the writesets of concurrent transactions intersect; if they do, the transaction is aborted. However, since such a certification test does not ensure one-copy serializability, conflict materialization techniques are adopted in the DBSM.

A number of works have compared the performance of group-communication-based database replication. In [5] Holliday et al. use simulation to evaluate a set of four abstract replication protocols based on atomic broadcast. The authors conclude that single-broadcast transaction protocols allow better performance by avoiding duplicated execution and blocking. These protocols abstract the DBSM. Another recent work evaluates the original DBSM approach, where a real implementation of DBSM's certification test and communication protocols is used [22]. In [7] the authors evaluate the suitability of the DBSM and other protocols for replication of OLTP applications in clusters of servers and over wide-area networks. All the results confirm the usefulness of the approach.

In [13] and [14] the authors present three protocols (Discor, Nodo and Reordering) which use conflict classes for concurrency control of update transactions. A conflict class, as a transaction type in our load-balancing algorithms, represents a partition of the data. Unlike the vDBSM, conflict classes are used for transaction synchronization. The vDBSM does not use any partitioning information within the replication algorithm. Load-balancing techniques use transaction types to increase the performance of the vDBSM (by reducing the abort rate), and not for correctness, as in [13] and [14].

Little research has considered workload analysis to increase the performance of a replicated system. In [12] 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.

In [2] the authors use transaction scheduling to design a lazy replication technique that guarantees one-copy serializability. Instead of resolving conflicts by aborting conflicting transactions, they augment the scheduler with a sequence numbering scheme to provide strong consistency. Furthermore the scheduler is extended to include conflict awareness in the sense that a conflicting read operation that needs to happen after some write operation is sent to the replica where the write has already completed.

9 Final remarks

To keep low abort rate despite the coarse granularity of middleware-based replication protocols, we introduced conflict-aware load-balancing techniques that minimize the number of conflicting transactions executing on distinct database servers and maximize the parallelism between replicas. Our techniques can be used with both static and dynamic load-balancing strategies. Current work is investigating variations of MCF and MPL applied to databases with consistency criterion different than serializability.

References

[1] Y. Amir and C. Tutu. From total order to database replication. In *Proc. of the International Conference on Dependable Systems and Networks*, 2002.

[2] C. Amza, A. Cox, and W. Zwaenepoel. Conflict-Aware Scheduling for Dynamic Content Applications. In *Proc. of the 4th USENIX Symposium on Internet Technologies and Systems*, 2003.

[3] B.Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proc. of the 26th International Conference on Very Large Data Bases*, 2000.

[4] E.Cecchet, J.Marguerite, and W.Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *Proc. of USENIX Annual Technical Conference, Freenix track*, 2004.

[5] J. Holliday, D.Agrawal, and A. E. Abbadi. The performance of database replication with group multicast. In *Proc. of the 29th Annual International Symposium on Fault-Tolerant Computing*, 1999.

[6] R. Jiménez-Peris, M. Patino-Martínez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *Proc. of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, 2002.

[7] A. Jr., A. Sousa, L. Soares, J.Pereira, F. Moura, and R. Oliveira. Group-based replication of on-line transaction processing servers. In *Proc. of Second Latin American Symposium on Dependable Computing (LADC)*, 2005.

[8] A. C. Jr., A. Sousa, L. Soares, F. Moura, and R. Oliveira. Revisiting epsilon serializabilty to improve the database state machine (extended abstract). In *Proc. of the Workshop on Dependable Distributed Data Management, SRDS*, 2004.

[9] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proc. of 19th International Conference on Distributed Computing Systems*, 1999.

[10] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[11] Y. Lin, B. Kemme, M. Patino-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Proc. of the 2005 ACM SIGMOD International Conference on Management of data*, 2005.

[12] J. M. Milan-Franco, R. Jiménez-Peris, M. Patino-Martínez, and B. Kemme. Adaptive middleware for data replication. In *Proc. of the 5th ACM/IFIP/USENIX International Conference on Middleware*, 2004.

[13] M. Patino-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Proc. of the 14th International Conference on Distributed Computing*, 2000.

[14] M. Patino-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Consistent Database Replication at the Middleware Level. *ACM Transactions on Computer Systems (TOCS)*, 2005.

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

[16] F. Pedone and S. Frolund. Pronto: A fast failover protocol for off-the-shelf commercial databases. In *Proc. of the 19th IEEE Symposium on Reliable Distributed Systems*, 2000.

[17] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proc. of the 4th International Euro-Par Conference on Parallel Processing*, 1998.

[18] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Journal of Distributed and Parallel Databases and Technology*, 14:71–98, 2002.

[19] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *Proc. of the 5th ACM/IFIP/USENIX International Conference on Middleware*, 2004.

[20] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. The GlobData fault-tolerant replicated distributed object database. In *Proc. of the 1st Eurasian Conference on Advances in Information and Communication Technology*, 2002.

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

[22] A. Sousa, J.Pereira, L. Soares, A. Jr., L.Rocha, R. Oliveira, and F. Moura. Testing the dependability and performance of group communication based database replication protocols. In *Proc. of IEEE International Conference on Dependable Systems and Networks - Performance and Dependability Symposium*, 2005.

[23] A. Sousa, F. Pedone, F. Moura, and R. Oliveira. Partial replication in the database state machine. In *Proc. of the IEEE International Symposium on Network Computing and Applications*, 2001.

[24] Transaction Proccesing Performance Council (TPC). TPC benchmark C. Standard Specification, 2005. http://www.tpc.org/tpcc/spec/.

[25] V.Zuikeviciute and F.Pedone. Revisiting the database state machine approach. In *Proc. of VLDB Workshop on Design, Implementation, and Deployment of Database Replication*, 2005.

[26] S. Wu and B. Kemme. Postgres-R(SI):combining replica control with concurrency control based on snapshot isolation. In *Proc. of the IEEE International Conference on Data Engineering*, 2005.

A Appendix: Proof

Theorem 1 *The vDBSM ensures one-copy serializability.*

Proof: We show next that if a transaction T_i commits in the vDBSM, then it also commits in the DBSM. Since the DBSM guarantees one-copy serializability, a fact proved in [18], it follows that the vDBSM is also one-copy serializable.

To commit on site S_k in the vDBSM, T_i must pass the certification test. Therefore, it follows that for every d_x read by T_i, $V_i[x] = V(T_i)[x]$. We have to show that if $V_i[x] = V(T_i)[x]$ holds for each data item d_x read by T_i then for every transaction T_j committed at $site(T_i)$, either T_i started after T_j committed, or T_j does not update any data items read by T_i, that is, $writeset(T_j) \cap readset(T_i) \neq \emptyset$. For a contradiction assume $V_i[x] = V(T_i)[x]$ for all d_x read by T_i and there is a transaction T_k such that T_i starts before T_k commits and $writeset(T_k) \cap readset(T_i) \neq \emptyset$. Let $d_x \in writeset(T_k) \cap readset(T_i)$. Before T_i starts, the current version of d_x, $V[x]$ is collected and stored in $V(T_i)[x]$. When T_k commits, $V_i[x]$ is incremented. Since T_k commits before T_i is certified, it cannot be that $V_i[x] \neq V(T_i)[x]$, a contradiction that concludes the proof.