Conflict-Aware Load-Balancing Techniques for Database Replication

Vaidė Zuikevičiūtė Fernando Pedone

University of Lugano (USI)
CH-6904 Lugano, Switzerland
Phone: +41 58 666 4695 Fax: +41 58 666 4536
vaide.zuikeviciute@lu.unisi.ch
fernando.pedone@unisi.ch

Abstract

Middleware-based database replication protocols require few or no changes in the database engine. Thus, 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 assign transactions to replicas so that the number of conflicting transactions executing on distinct servers is reduced and the processing load is equitably distributed over the servers. 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.

Keywords: database replication, load balancing, middleware

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 much attention in the last years (e.g., [4, 12, 15, 19, 20]). 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 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.

Designing a load balancer that exploits observations (a) and (b) efficiently is not simple. Ideally, we would like 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 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 requirement.

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 scheduling transactions with the sole goal of maximizing parallelism (i.e., "pure" MPF) already doubles the throughput obtained with random assignment of transactions. Scheduling transactions in order to minimize conflicts only (i.e.,"pure" MCF) 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. Under higher loads a hybrid approach, combining MCF and MPF, outperforms "pure" MPF.

The contributions of this paper are the following: (a) we introduce conflict-aware load-balancing techniques for middleware-based database replication protocols; (b) we present the vDBSM, a novel middleware-based replication protocol; (c) we perform a detailed static analysis of the proposed algorithms behavior when applied to the TPC-C benchmark; (d) we implement the proposed conflict-aware load-balancing techniques and evaluate their performance using the TPC-C bechmark.

2 Background

2.1 System 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.

We do not make any strong assumptions about the time it takes for clients and servers to execute and for messages to be transmitted. We do assume however that the system is augmented with additional assumptions allowing total-order broadcast to be solved.

2.2 Database model

Each server has a full copy of the database. Servers execute *transactions* according to strict two-phase locking (2PL)[3]. 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, 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.

2.3 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).

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 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 and its readset by $server(T_i)$ and $readset(T_i)$, 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. Read-only transactions are local to the replica on which they execute.

   During the execution of an update transaction T_i, the versions of the data items read by T_i 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 the transaction type and its parameters 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 precisely, 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 update statements are executed against 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*, and thus reduce the number of certification aborts—but notice that regardless of the server chosen for the execution of a transaction, the vDBSM always ensures consistency (i.e., one-copy serializability).

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 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; likewise, if the load is low, few transactions will execute concurrently and minimizing the number of conflicting transactions executing at distinct replicas becomes less important.

In Sections 4.1 and 4.2 we present *Minimizing Conflits First (MCF)* and *Maximizing Parallelism First (MPF)*, two greedy algorithms that prioritize different requirements when assigning transactions to preferred servers. In Section 4.3 we give a simple example to illustrate the behavior of the algorithms.

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 there are no conflicts, the algorithm tries to balance the load among the replicas, maximizing parallelism.

1. Consider replicas S_1, S_2, ..., S_n. We say that transaction T_i belongs to S_k^t at time t, $T_i \in S_k^t$, if at time t T_i is assigned to execute on server S_k.

2. For each transaction T_i in the workload, to assign T_i to some server at time t execute step 3, if T_i is an update transaction, or step 4, if T_i is a read-only transaction.

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

   (a) 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 $w(S_k, t) = \sum_{T_j \in S_k^t} w_j$.

   (b) If $|C(T_i, t)| = 1$, assign T_i to the replica in $C(T_i, t)$.

   (c) If $|C(T_i, t)| > 1$, 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. Assign read-only transaction T_i to the replica S_k with the lowest aggregated weight $w(S_k, t)$ at time t, where $w(S_k, t)$ is defined as in step 3(a).

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 $f, 0 < f \leq 1$. 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.

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 minimal load at time t, where $w(S_l,t)$ has been defined in step 3(a) 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 $W(t)$: $C_W(T_i,t) = \{S_k \mid S_k \in W(t) \text{ and } \exists T_j \in S_k \text{ such that } T_i \sim T_j\}$.

   (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 $w(C_{T_i}(S_k^t))$ counting only transactions conflicting with T_i, $w(C_{T_i}(S_k^t))$ is formally defined in step 3 in Section 4.1; if several replicas in $C_W(T_i,t)$ satisfy this condition, assign T_i to any one of these.

5. Assign read-only transaction T_i to the replica S_k with the lowest aggregated weight $w(S_k,t)$ at time t.

Notice that the MCF algorithm is a special case of MPF with a factor $f = 0$.

4.3 A simple example

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 $w(T_i) = i$. All transactions are submitted concurrently to the system and the load balancer processes them 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 $w(S_4) = 13$.

5 Analysis of the TPC-C benchmark

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

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 10 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*(NO), *Payment*(P), *Delivery*(D), *Order Status*(OS) and *Stock Level*(SL). *Order Status* and *Stock Level* are read-only transactions; the others are update transactions. Since only update transactions count for conflicts—read-only transactions execute at preferred servers just to balance the load—there are only three update transaction types to consider: *Delivery*, *Payment*, and *New Order*. These three transaction types compose 92% of TPC-C workload.

In the following we define the workload of update transactions as:

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

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 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. 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:

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

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

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 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.

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 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. If 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 $O_R(S_j, S_i) = \frac{w(T_4)+w(T_6)}{w(T_4)+w(T_5)+w(T_6)+w(T_7)}$.

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 results in a fair load distribution but has very high overlapping ratio. MPF 1 (not shown in the graphs) behaves similarly to Random: it distributes the load over all the replicas, but has 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. Even if more replicas were available, MCF would still strive to minimize the overlapping ratio, assigning update transactions to only 4 replicas. A compromise between maximizing parallelism and minimizing conflicts can be achieved

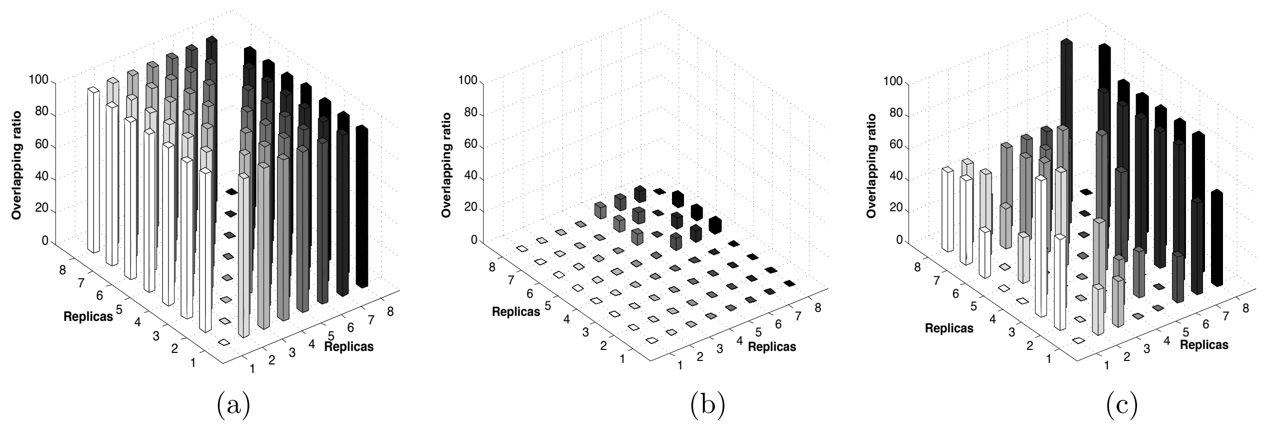

Figure 1: Overlapping ratio, (a) Random (b) MCF (c) MPF 0.5

by varying the f factor of the MPF algorithm. With $f = 0.5$ the overlap ratio is much lower than Random (and MPF 1), for example.

6 Prototype overview

We have built a prototype of the vDBSM in Java v.1.5.0. Client applications interact with the replicated compound by submitting SQL statements through a customized JDBC-like interface. Application requests are sent 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 [11].

On delivery, transactions are 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. We group the updates of several remote transactions into a single transaction in order to reduce the number of disk writes.

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 the 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.

The assignment of submitted transactions is computed on-the-fly based on currently executing transactions at the replicas. The load balancer keeps track of each transaction's execution and completion status at the replicas. No additional information exchange is needed for locally executing transactions. For better load estimation replication modules periodically inform the load balancer about the remote updates waiting to be applied at the replicas. Our load balancer is lightweight: CPU usage at the load balancer is less than 4% throughout all experiments.

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. The isolation level was set to serializable throughout all experiments.

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 (i.e., we do not use TPC-C think times).

7.2 Throughput and response time

The TPC-C workload consists of 5 transactions types, each of which accounts for different computational resources. Hereafter we present the results based only on *New Order* transactions, which represent ≈ 45% of total workload. *Payment* transactions, accounting for another 43% of the workload, perform analogously. We measure throughput in terms of the number of *New Order* transactions committed per second (tps). The response time reported represents the mean response time of committed *New Order* transactions in milliseconds (msec).

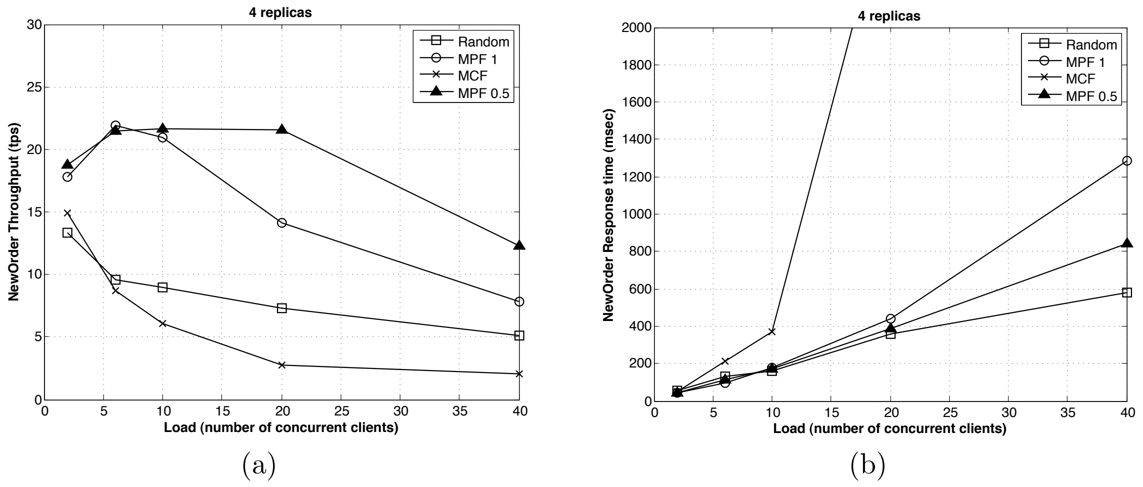

Figure 2: (a) throughput and (b) response time of *New Order* transactions; 4 replicas

We initially studied the effects of our techniques under various load conditions. Figure 2 shows the achieved throughput and response time of committed *New Order* transactions on a system with 4 replicas. 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.

We then considered how the proposed algorithms react to a varying number of replicas. Notice that adding new replicas in an update intensive environment (92% af all transactions in TPC-C are updates) will improve the availability of the system but may not increase its performance. In the vDBSM, transactions are executed and broadcast by one server, then delivered, validated, and applied by all replicas. More replicas may help spreading the load of executing transactions, but in any case, their updates will be processed by every server. Moreover, in some techniques adding more replicas increases the chance that conflicting transactions are executed on distinct replicas, consequently causing more aborts.

For the following experiments, we kept the load constant at 10 concurrently-executing clients while increasing the number of replicas from 2 to 8. Figure 3 shows that the effects of our load-balancing algorithms are sustainable. In fact, Random and MCF benefit from an increasing number of replicas. None of them reaches the performance of MPF 0.5 and MPF 1, which behave similarly for all configurations considered. From these experiments, it turns out that increasing the availability of the system with MPF can be achieved without performance penalties.

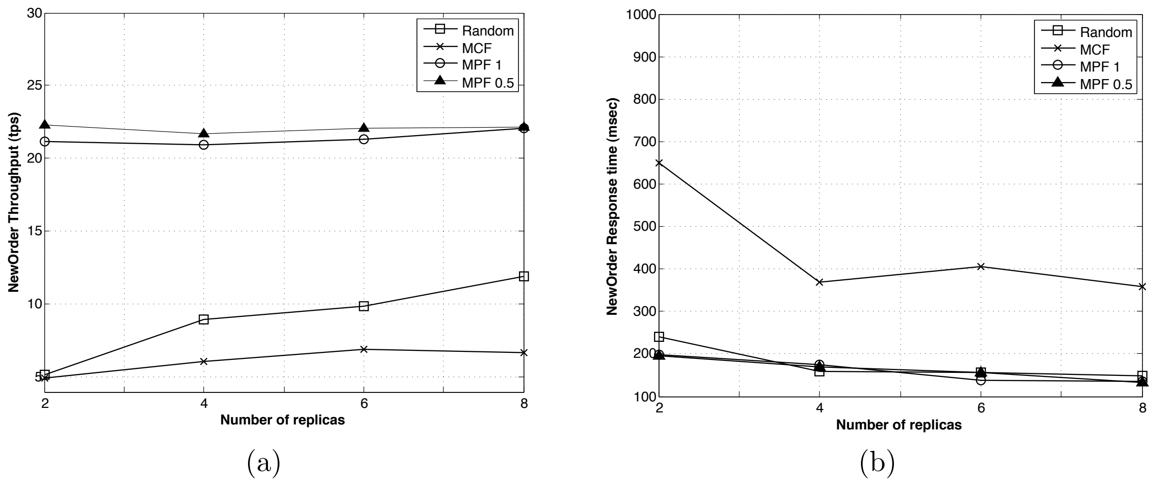

Figure 3: (a) throughput and (b) response time of *New Order* transactions; varying replicas

7.3 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 (see Section 6), (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.

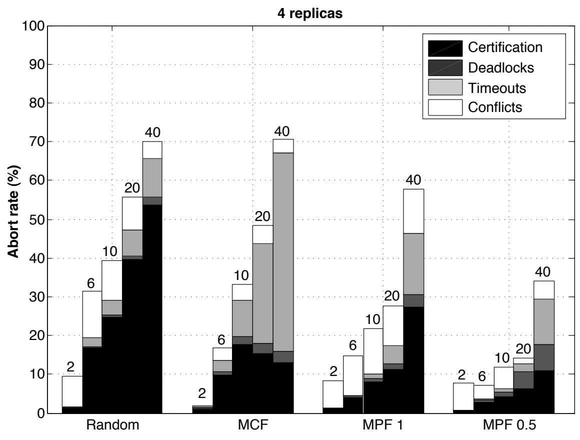

Figure 4: Abort rates

Figure 4 shows the abort rate breakdown for each of the techniques run on the system with 4 replicas; 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. A hybrid approach, MPF 0.5, which tries to account for both conflicts and load, reduces the number of aborts from ≈ 40% to ≈ 11%.

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 [10, 15] or optimistically [9, 18], and in the database correctness criteria they provide—one-copy serializability [1, 15, 16, 17, 18, 20] or snapshot isolation [12, 19, 25].

The original DBSM has been previously extended in several directions. Sousa et al. investigate the use of partial replication in the DBSM [23]. In [6] the authors relax the consistency criteria of the DBSM with Epsilon Serializability. The work in [26] 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 [8] 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 [5] 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.

Pacitti et al. [14] present preventive replication that supports multi-master and partial configurations. The consistency is guaranteed by annotating transactions with a chronological timestamp value and ensuring FIFO order among messages send by a replica. The transaction is allowed to execute on a database server only when the upper bound of the time needed to multicast a message has exceeded. Further optimizations of the protocol allows concurrent transactions execution at the replicas and eliminate the delay time.

In [15] 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 need 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 [15]. In fact the techniques proposed here could be combined with those in [15].

Little research has considered workload analysis to increase the performance of a replicated system. In [13] 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 [7] 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 [7] 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.

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. 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.

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 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, combining MCF and MPF, allows database administrators to trade even load distribution for low transaction aborts in order to increase throughput with no degradation in response time.

References

[1] Y. Amir and C. Tutu. From total order to database replication. In *Proceedings 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 *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, 2003.

[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 USENIX Annual Technical Conference, Freenix track*, 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 *Proceedings of Second Latin American Symposium on Dependable Computing*, 2005.

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

[7] S. Elnikety, S. Dropsho, and W. Zwaenepoel. Tashkent+: Memory-aware load balancing and update filtering in replicated databases. In *Proceeding of EuroSys*, 2007.

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

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

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

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

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

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

[14] E. Pacitti, C. Coulon, P. Valduriez, and M. Tamer Özsu. Preventive replication in a database cluster. *Distributed and Parallel Databases*, 2005.

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

[16] F. Pedone and S. Frolund. Pronto: A fast failover protocol for off-the-shelf commercial databases. In *Proceedings 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 *Proceedings 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 *Proceedings 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 *Proceedings 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. Correia, L.Rocha, R. Oliveira, and F. Moura. Testing the dependability and performance of group communication based database replication protocols. In *Proceedings 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 *Proceedings 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] S. Wu and B. Kemme. Postgres-R(SI):combining replica control with concurrency control based on snapshot isolation. In *Proceedings of the IEEE International Conference on Data Engineering*, 2005.

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