Managing Transaction Conflicts in Middleware-based Database Replication Architectures *

F. D. Muñoz-Escoí - J. Pla-Civera - M. I. Ruiz-Fuertes - L. Irún-Briz - H. Decker
Instituto Tecnológico de Informática
Univ. Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia, SPAIN
email: {fmunyoz,jpla,miruifue,lirun,hendrik}@iti.upv.es

J. E. Armendáriz-Iñigo - J. R. González de Mendívil
Depto. de Matemática e Informática
Univ. Pública de Navarra
Campus de Arrosadía
31006 Pamplona, SPAIN
email: {enrique.armendariz,mendivil}@unavarra.es

Abstract

Database replication protocols need to detect, block or abort part of conflicting transactions. A possible solution is to check their writesets (and also their readsets in case a serialisable isolation level is requested), which however burdens the consumption of CPU time. This gets even worse when the replication support is provided by a middleware, since there is no direct DBMS support in that layer. We propose and discuss the use of the concurrency control support of the local DBMS for detecting conflicts between local transactions and writesets of remote transactions. This allows to simplify many database replication protocols and to enhance their performance.

1 Introduction

Many database replication protocols use the constant interaction approach [19] for update propagation among replicas; i.e., they only propagate their updates by a fixed number of multicasts to the rest of replicas (a single multicast at the end of each transaction is most usual). In such protocols, transactions are executed following a scheme which is similar to the passive replication model: a dedicated master replica directly receives the transaction sentences, and once the commit procedure begins or terminates (depending on the protocol class [7] being either eager or lazy, respectively), the master replica gets the transaction writeset and multicasts it to the rest of the replicas. Once this writeset is received in the rest of replicas, it has to be applied. There are different techniques for applying the updates in this writeset, which depend on the writeset contents [10]: either standard SQL sentences or some specific data structure, such as the usual contents of a write-ahead redo log.

Whenever such writeset contents are applied in the local database replica, concurrency control has to be taken care of, since these updates may conflict with previous accesses made by other local concurrent transactions. In some cases, such conflicts may entail that the writesets must remain blocked until those local transactions have terminated, or that the latter must be aborted. In concurrent systems, transactions that end up terminating with abortion ultimately are consuming time and resources in vain before aborting. These blocked resources could otherwise be made use of advantageously for processing other, successful transactions. The more time passes before the concurrency control finally aborts a transaction, the more processing power is consumed in vain. Thus, a mechanism able to detect abort conditions as early as possible should be included in the concurrency control.

Such a concurrency control support for database replication can be achieved by a simple extension of the DBMS core, to the effect that appropriate locks are requested before the writeset is applied. On the other hand, details of

*This work has been partially supported by the Spanish grant TIC2003-09420-C02.

such extensions will always depend on the specific DBMS at hand, possibly entailing a lot of problems related to maintenance, interoperability and migration. Thus, instead of proprietary DBMS extensions, a product-independent middleware solution may be deployed for supporting database replication. However, the above-mentioned concurrency control checks may be less straightforward in a middleware-based solution than in a DBMS extension, or, at least, more costly.

In this paper, we describe a technique for managing concurrency control which combines the simplicity of using DBMS core support while maintaining the product independence of a middleware solution. Transaction conflicts are detected swiftly and replication protocols can better focus on their main targets. Instead of having to request and wait for termination, conflicting transactions may be immediately aborted. By reducing the abortion delay, the system becomes ready faster for processing other active transactions. The resulting performance improvement of replication protocols is corroborated by some tests presented in the remainder.

We have implemented and tested our approach in PostgreSQL. Our solution needs to scan the system's locking tables. Similar tables are used in virtually all DBMSs, (e.g., the `V$LOCK` view in Oracle 9i, the `DBA_LOCK` in Oracle 10g r2, the `sys.syslockinfo` table of Microsoft SQL Server 2000 –converted into a system view in SQL Server 2005–, etc.) so that this scheme is seamlessly portable to all of them, since only standard SQL constructs are used.

The remainder is structured as follows. Section 2 describes a scheme for detecting transaction conflicts with the help of the DBMS. Section 3 describes a specific protocol [12] and some variations that are used for testing the benefits of our conflict detection scheme. Section 4 provides some performance results that compare the approaches discussed in the previous section. Finally, sections 5 and 6 discuss related work and conclude the paper.

2 A Scheme for Conflict Detection

The functionality described in this section is part of the bottom layer of the MADIS middleware, explained in more detail in [9]. It is easily portable to other middleware architectures since it does not depend on other components of our architecture. Its aim is to detect conflicting pairs of transactions.

MADIS has been implemented in Java, with a JDBC interface to applications. Its current communication support is provided by the Spread group communication system [1, 4]. Since, in this paper, nothing beyond the manner in which the detection of conflicting concurrent transactions is supported by the middleware architecture, no additional details about the group communication system nor about MADIS are relevant. Moreover, we remark that the description of protocols in the next section is not complete, in as far as nothing is said about their recovery procedures. Yet again, the modularity of our approach guarantees that details about these procedures are irrelevant for discussing conflict detection (although they are of course of key importance for the main purpose of replication, viz., enabling high availability).

Notice, however, that our system has been designed for supporting full replication, i.e., each machine in the network covered by the middleware has a complete database replica. All replication protocols considered in this paper adhere to this kind of replication. Moreover, each of them can be classified as eager, with constant interaction and non-voting termination, according to [19]. Full replication with eager non-voting replication support is a preferable option for middleware-based wide-area distributions of OLTP systems and many other database applications, e.g., e-commerce transactions.

Coming back to conflict detection, the main advantage of our approach is the use of the concurrency control support of the underlying DBMS. Thereby, the middleware is enabled to provide a row-level control (as opposed to the usual coarse-grained table control), while all transactions (even those associated to remote writesets) are subject to the underlying concurrency control support. Its implementation is based on the following two elements:

- The database schema is enhanced by the stored function getBlocked(). It looks up blocked transactions in the DBMS metadata (e.g., in the *pg_locks* view of the PostgreSQL system catalogue). It returns a set of pairs consisting of the identifiers of a blocked transaction and of the transaction that has caused the block. If there is no conflict when this function is called, it returns the empty set.

  In short, getBlocked() reads a system catalogue table in which the DBMS keeps information about transaction conflicts. Such a table is maintained by most DBMSs. Thus, this function is easily portable to most of them. Moreover, these DBMSs only provide read access to this system table. So, reading such views or tables does not compromise the regular activity of the DBMS core nor the activity of other transactions.

- An execution thread per database is needed that cyclically calls getBlocked(). Its cycle is commonly set to values between 300 and 1000 ms. It runs on the middleware layer.

  Once this thread has received a non-empty set of conflicting pairs of transactions, it may request the abortion of one of them. For this purpose, each transaction has a priority level assigned to it. By default, it aborts the transaction with smaller priority but takes no action if both transactions have the same priority level.

This mechanism should be combined with a transaction priority scheme in the replication protocol. For instance, we might define two priority classes, with values 0 and 1. Class 0 is assigned to local transactions that have not started their commit phase. Class 1 is for local transactions that have started their commit phase and also for those transactions associated to delivered writesets that have to be locally applied. Once a conflict is detected, if the transactions have different priorities, then the one with the lowest priority will be aborted. Otherwise, i.e., when both transactions have the same priority, no action is taken and they remain in their current state until the lock is released. Similar, or more complex approaches might be followed in other replication protocols that belong to the update everywhere with constant interaction class [19], as described in the next section.

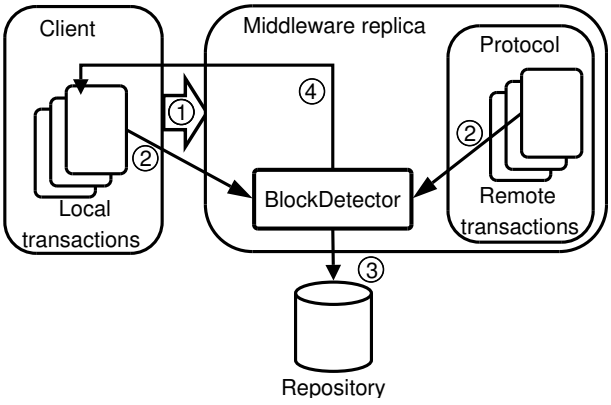


Figure 1. Steps in the block detection procedure

Note that this conflict detection strategy is intentionally very simple and quite flexible, as shown in fig. 1. Nothing but the set of pairs of conflicting transaction identifiers needs to be known, which can be straightforwardly obtained from the locking table, as described above. To begin with, a client application uses the middleware for creating local transactions (step 1). The middleware is able to filter such requests and sets the transaction identifier and priority level in an array managed by the monitoring thread (step 2). Such task is made for both local and remote (i.e., writeset-application) transactions. This information is then processed by the monitoring thread each time it reads the system catalogue table (step 3). Once a conflict arises, and depending on the transactions' priority, one of two conflicting transactions can be aborted immediately. The transaction to be cancelled is in many replication protocols the blocking (i.e., active) one, not the blocked one. Thus, its cancellation can be requested by the middleware since it has access to all active connections; i.e., the middleware uses the context information of such an active blocking transaction to request its own rollback (step 4). Subsequently, the blocked transaction (usually the application of the writeset of a remote transaction) is reactivated without further delay and thus may terminate faster and successfully.

3 A Case Study: The SI-Rep Protocol

We have selected the SI-Rep protocol described in [12] for a case study of our conflict detection mechanism. There are several reasons that justify this selection: (i) It is a good protocol for guaranteeing the *snapshot* isolation level [2] (at least, its 1CSI variation as described in [12]), as shown by its performance measurements. (ii) Its description includes a lot of implementation optimisations that improve its overall performance. So, the following three implementations variants of such a protocol can be tested: (a) an optimised version without our conflict detection technique, (b) the optimised version with our conflict detection technique, (c) the non-optimised version with our conflict detection technique. Such three implementations allow us to test the advantages and inconveniences of our block detector.

This protocol implements the snapshot isolation level. According to [2], this isolation level needs to assign two timestamps to each transaction. The first one is its start timestamp, to be set when the transaction initiates its first database access. The second one is its commit timestamp, to be assigned when the transaction successfully commits. Both usually are logical timestamps, i.e., implemented as counters. In order to commit a transaction T, a single rule has to be respected in a centralised implementation of this isolation level: *No other transaction T' with a commit timestamp in the interval [$start(T)$, $commit(T)$] wrote data that T also wrote.* This correctness rule is needed for understanding the SI-Rep protocol as described below, and for justifying that our blocking detection technique does not violate the correctness criteria of the original protocol. Some additional requirements on this isolation level for replicated databases have been discussed and formalised in [6], and they have been followed by all the implementations discussed here.

This protocol uses an atomic multicast [8], i.e., a reliable multicast with total order delivery, and thus it ensures that the writesets being multicast by each replica at commit time are delivered in all replicas in the same order. It uses two data structures for dealing with writesets: *ws_list*, which stores all the writesets known (i.e., delivered) until now, and *tocommit_queue*, which holds those writesets locally certified but not yet applied in the local database replica. Moreover, for each transaction, the attributes *cert* and *tid* hold something similar to the transaction start and commit timestamps, respectively.

The steps that define this protocol are shown in Fig. 2

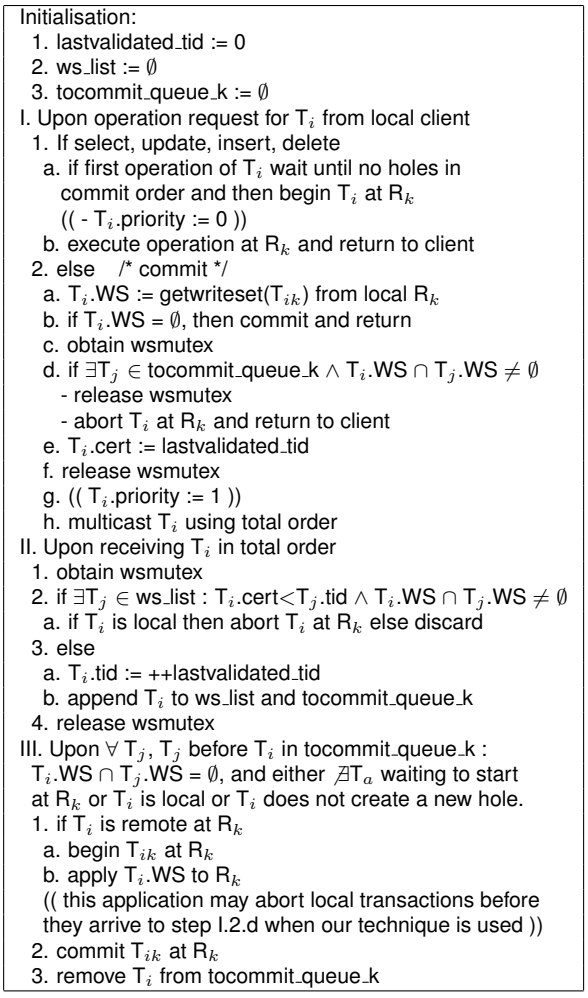


Figure 2. SI-Rep Algorithm at replica R_k (SIR-BD **extensions in parentheses)**

(further details can be found in [12]). In this figure, we have added in parentheses the extensions needed to include our blocking detection mechanism in SI-Rep. Such extensions must be ignored in the following descriptions, since they provide the basis of our SIR-BD implementation that will be discussed later.

As can be seen in that figure, several conflict detection checking variants are used in this protocol. The first one in step I validates locally the transaction before its writeset is multicast. Basically, it is checked whether the correctness rule described at the start of this section is satisfied. However, there may be other concurrent transactions whose writeset is delivered before that of the analysed transaction, which may also lead to its abortion. Those transactions are checked in the global validation action of step II. Finally, the writeset application procedure in step III also needs to check for conflicts among writesets for deciding when each writeset may be securely applied. This last checking enables the concurrent execution of non-conflicting writesets, i.e., it serves to optimise the protocol performance but is not strictly necessary for the correctness of the protocol. Note that also the checks in step I are, in a sense, redundant, since they would be made in step II anyway, but doing them ahead of step II may reduce the amount of needed multicasts, since some transactions that otherwise would be aborted in II can be aborted earlier.

Note that we have tacitly assumed that the underlying database system is supposed to be able to check for conflicts, and to abort transactions the access patterns of which violate the *snapshot* isolation level rules.

Based on this SI-Rep protocol, we have developed the following three implementations:

- **SIR** (*SI-Rep*): In this case, our SI-Rep implementation follows all steps of the protocol described in [12] (i.e., the algorithm presented in fig. 2 without any of the extensions in parentheses). It scans the writesets for deciding if they have a non-empty intersection. At worst, an item per item comparison would be necessary, detecting a conflict once a matching pair of items is found; if no match arises, the writesets do not conflict. However, as we will see in the following section, this conflict check can be easily optimised.

- **SIR-BD** (*SI-Rep with Blocking Detection*): In this second alternative, based on the previous implementation, we have added our blocking detection mechanism. None of the checks in the previous alternative have been removed. We have assigned the following priorities to the transactions. All transactions are initialised with a 0 priority level. They get level 1 when they are multicast in their local node or when their writeset is delivered in their remote nodes. This ensures the correctness of this alternative, since our blocking detection mechanism aborts a transaction only if all of these conditions are satisfied. Otherwise, no particular action is taken.

  - The transaction to be aborted is local.
  - It has not locally requested its commit; i.e., its writeset has not been multicast.
  - The transaction that causes its abortion has been generated for applying a remote writeset.

  This approach satisfies the correctness criteria of the *snapshot* isolation level, since the writeset is associated to a transaction that has successfully passed its global validation phase. It already has a commit timestamp which of course is in the range of the $[start, commit]$ interval of the local transaction, since

the latter has not yet requested its commit. Moreover, it is able to abort local transactions even before SIR does, although SIR is seen as a highly optimised protocol for early conflict detection.

```
Initialisation:
 1. lastvalidated_tid := 0
 2. lastcommitted_tid := 0
 3. ws_list := ∅
 4. tocommit_queue_k := ∅
I. Upon operation request for T_i from local client
 1. If select, update, insert, delete
  a. if first operation of T_i
   - T_i.cert := lastcommitted_tid
   - T_i.priority := 0
  b. execute operation at R_k and return to client
 2. else   /* commit */
  a. T_i.WS := getwriteset(T_ik) from local R_k
  b. if T_i.WS = ∅, then commit and return
  c. T_i.priority := 1
  d. multicast T_i using total order
II. Upon receiving T_i in total order
 1. obtain wsmutex
 2. if ∃ T_j ∈ ws_list : T_i.cert < T_j.tid ∧
  T_i.WS ∩ T_j.WS ≠ ∅
  a. release wsmutex
  b. if T_i is local then abort T_i at R_k else discard
 3. else
  a. T_i.tid := ++lastvalidated_tid
  b. append T_i to ws_list and tocommit_queue_k
  c. release wsmutex
III. T_i := head(tocommit_queue_k)
 1. if T_i is remote at R_k
  a. begin T_ik at R_k
  b. apply T_i.WS to R_k
  c. ∀ T_j : T_j is local in R_k ∧ T_j.WS ∩ T_i.WS ≠ ∅
   ∧ T_j has not arrived to step II
   (this is analysed by our conflict detector,
   concurrently with the previous step III.1.b)
   - abort T_j
 2. commit T_ik at R_k
 3. ++lastcommitted_tid
 4. remove T_i from tocommit_queue_k
```

Figure 3. SIR-SBD **algorithm at replica R_k**

- **SIR-SBD** (*SI-Rep, Simplified and with Blocking Detection*): This last implementation uses a simplified version of the SI-Rep protocol, where all writeset checks being made at steps I and III have been removed. Figure 3 shows its actual algorithm. In this variant we have eliminated the possibility of applying non-conflicting writesets concurrently. Our aim is only to prove that, by combining the mandatory writeset check of step II and our blocking detection mechanism, we obtain acceptable results. Of course, adding the possibility of concurrent execution among non-conflicting writesets yields better performance results in heavily loaded systems.

These protocols have been implemented in the current version of our middleware. It is still a prototype and does not perform well yet for writeset collection. Due to this, the performance results discussed in the next section are impaired. However, note that this disadvantage is common to each protocol considered in the comparison.

4 Performance Results

Based on a comparison of the three implementations of the SI-Rep protocol described in the previous section, we are going to show that our blocking detection strategy provides better results than any other middleware layer approach with the same conflict granularity. Our SIR-BD implementation embodies a regular configuration of our conflict detection technique, whilst SIR models a strongly optimised version of other techniques (as far as we know, other middleware architectures for database replication only achieve class granularity, where classes are usually associated to tables or fragments of tables that comprise many rows). The third protocol attempts to model a "simplified" version of the original protocol, which shows that our technique may be useful not only for middleware-based replication protocol design but also for concurrency control simplification.

To accomplish the comparison, we use PostgreSQL as the underlying DBMS, and a database with a single table with two columns and 10000 rows. One column is declared as primary key, containing natural numbers 1 to 10000 as values. The simplicity of this database enables a fast detection of conflicts with row granularity at the middleware layer, and in particular for the SIR protocol (i.e., one that does not use our automatised conflict detection). Since the primary key values form a contiguous interval, we only need two values for each writeset: its lower bound, i.e., the value f of its first row, and its upper bound, i.e., the writeset size ws. Thus, two writesets collide if and only if the f or f+ws value of one is contained in the [f,f+ws) interval of the other.

Note that the efficiency of this conflict checking for the SIR implementation does not generalise without further ado to any protocol. In general, an item-per-item scanning as discussed above is needed. Further note that we have chosen the very simple database above for granting a favour to SIR for our comparison. With a more complicated sample database, the performance of SIR would in fact be worse.

All protocols have been tested using our middleware with either 2, 4, or 8 replica nodes. Each node has a 2.8 GHz CPU with 1 GB of RAM running Linux Fedora Core 2 with PostgreSQL 7.4.12 and Sun Java 1.4.2. They are interconnected by a 1 Gbit/s Ethernet. In each replica, there are 10 concurrent clients, and 10 sequential transactions are executed via each client connection, without any pause between each pair of consecutive transactions. The monitoring thread that cyclically calls our GetBlocked() function

uses in these experiments a cycle of 1000 ms. Other experiments have been made using cycles of 300, 500, and 700 ms, but their results were practically the same as those presented here.

Each transaction is composed of two sentences, separated by a configurable delay for modelling a thinking user. As suggested by standard benchmarks, like TPC-C [18], we assume an interactive application that displays some information to the user and later an update generated by that user. The first sentence reads part of the table (with a *select for update* sentence). The second is an *update* SQL sentence which explicitly specifies the range of rows to be updated with a *where* clause. Different transactions with varying delay (1 or 4 seconds) and various amounts (5, 10, 15, 20, 40, 100) of items modified in the second sentence have been generated. Below, we refer to transactions with a delay of 1 second as *short transactions*, and to the rest as *long transactions*.

The work load in these tests consists of an uninterrupted sequence of transactions executed by each client (recall: 10 clients per replica). Hence, we think that this is adequate for bringing out the benefits of our conflict detection approach.

Based on this setting, the average completion time for committed and aborted transactions is shown below. The main advantage of our conflict detection technique is that it aborts sooner the transaction that would abort later anyway. This advantage becomes evident when comparing the completion time of aborted transactions in SIR and SIR-BD. Thus, our technique significantly contributes to decrease redundant resource consumption and hence to improve the overall transaction processing ratio.

Table 1 summarises all abortion rates obtained in our tests. As can be seen, our conflict detection technique does not introduce any significant abortion rate variation, which clearly accounts for its efficiency and scalability.

Figures 4 to 6 show the mean transaction processing time in six different scenarios, obtained by varying the numbers of replicas and updated items. For each combination of parameters, 100 samples have been run. All protocol plots consist of two curves, for committed and aborted transactions. Subsequently, we are going to discuss these curves for varying numbers of replicas.

Figure 4 shows performance results when only two replicas are used. Of all cases studied, this is the worst for our conflict detection technique, since the transaction throughput ratio increases with the number of replicas. More precisely: since our technique accelerates the detection of conflicts between remote writeset applications and local transactions, and since the number of such conflicts increases with the number of replicas, it follows that the relative overhead of our technique is amortised better by a larger number of replicas.

In spite of this, figure 4 shows that the results obtained

Table 1. Abortion rates in all experiments

Number of Updated Items	SIR Protocol					
	Replicas					
	2		4		8	
	Delay (sec.)					
	1	4	1	4	1	4
5	0.8	1.2	1.8	3.0	3.6	5.9
10	1.2	2.7	2.1	6.3	6.7	11.5
15	3.4	4.1	6.5	9.5	10.9	15.8
20	3.0	6.2	7.5	10.6	16.5	20.3
40	8.3	10.7	17.9	21.5	32.3	36.9
100	25.0	26.8	43.4	43.6	61.0	60.9

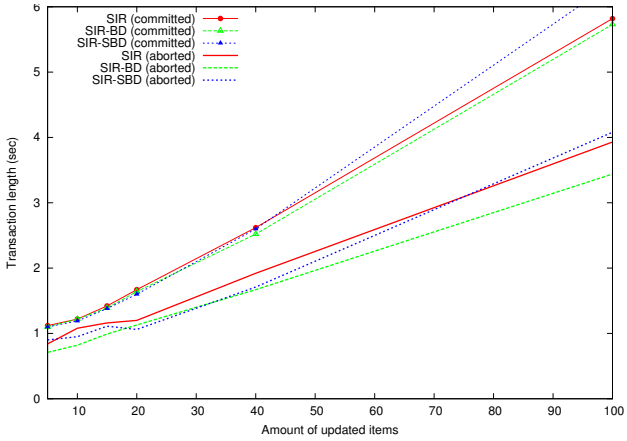
Number of Updated Items	SIR-BD Protocol					
	Replicas					
	2		4		8	
	Delay (sec.)					
	1	4	1	4	1	4
5	0.9	1.5	1.5	2.7	4.6	6.2
10	1.1	3.1	2.3	5.5	7.2	11.6
15	2.9	3.9	6.0	8.8	11.7	16.4
20	3.1	5.2	7.8	11.4	16.3	21.0
40	9.2	9.9	18	21.3	33.7	36.0
100	24.9	27.6	44.0	44.6	61.0	62.1

Number of Updated Items	SIR-SBD Protocol			
	Replicas			
	2		4	
	Delay (sec.)			
	1	4	1	4
5	0.8	1.9	1.6	2.9
10	1.1	2.2	2.1	5.4
15	2.3	4.6	4.4	7.9
20	3.6	5.2	6.3	10.8
40	6.7	11.4	17.4	20.5
100	21.1	23.6	44.1	41.8

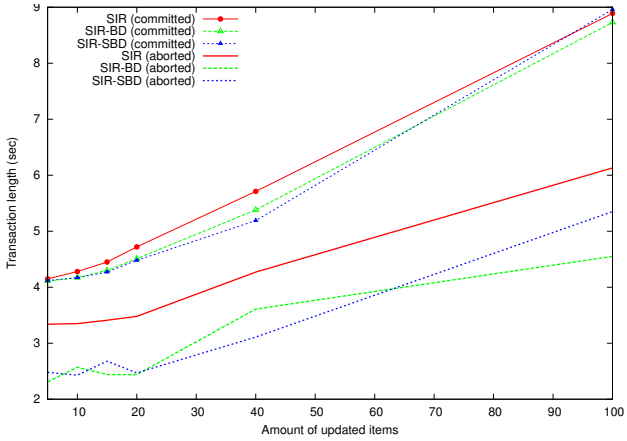
with SIR-BD are better than those of SIR. With short transactions (fig. 4.a), the average response time of SIR-BD is 15% shorter than that of SIR for aborted transactions. This reduces the overall system load and results in a 2% shorter response time for SIR-BD as compared to SIR for committed transactions. This shows that early abortions are directly beneficial for successfully committing transactions. This becomes even more evident with longer transactions. As can be seen in fig. 4.b, a delay of 4 seconds between the two accesses of each transaction results in a performance gain of around 30% of SIR-BD over *SIR* for aborted transactions and around 3% for committed transactions.

Recall that SIR-SBD is always impaired by its lack of SIR optimisations. Thus, its performance for short transactions is similar to that of SIR-BD whenever writesets are small, but it degrades with more than 40 items per writeset, in particular for committed transactions, since the SI-Rep optimisations are related to the concurrent application of remote writesets, and writeset delivery and application in the local database is included in the transaction completion phase. However, with long transactions, SIR-SBD's performance is intermediate to the other two implementations.

The results obtained for four replicas display a similar pattern to those for two replicas, but with a better per-

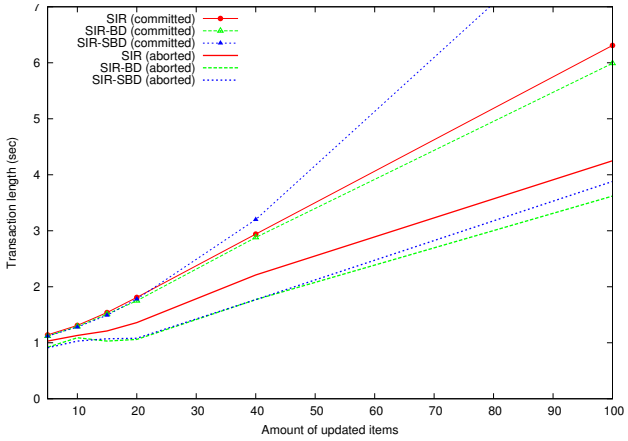


(a) Short transactions (delay=1sec)

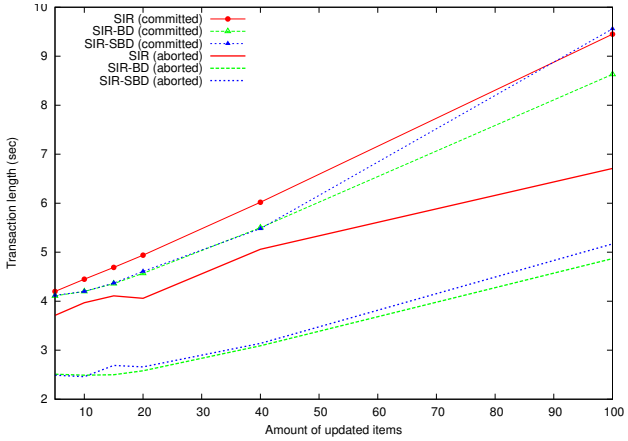


(b) Long transactions (delay=4sec)

Figure 4. Performance results for 2 replicas



(a) Short transactions (delay=1sec)



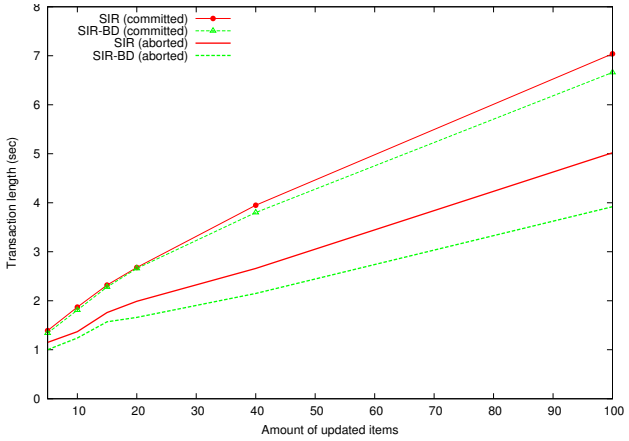
(b) Long transactions (delay=4sec)

Figure 5. Performance results for 4 replicas

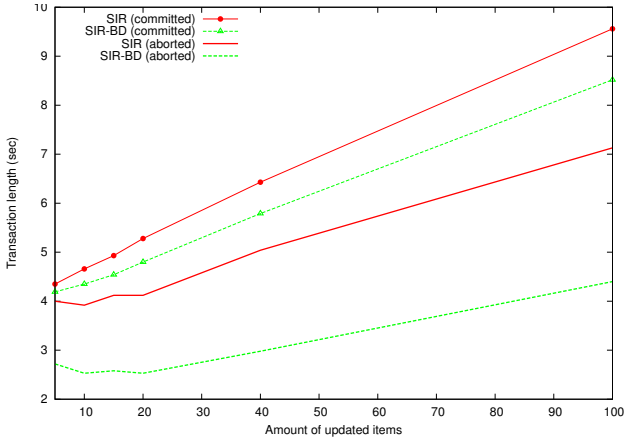
formance for our blocking detection technique. For short transactions, SIR-BD is again about 15% faster than SIR for aborted, and 3.5% faster for committed transactions, which is slightly better than with 2 replicas. Note that this positive trend for committed transactions also is indicative of the benefits obtained with increasingly large writesets.

For long transactions, the performance gain of SIR-BD over SIR is around 35% for aborted transactions and 8.3% for committed transactions (the latter is almost 3 times better than with 2 replicas). This means that the performance gains achievable by our conflict detection technique improve with an increasing number of replicas, since the entailed increase of conflicts produces a higher abortion rate. Since abortions are accomplished sooner and more rapidly, system resources that otherwise would remain blocked can be released earlier for processing other transactions, thus increasing the system throughput.

For short transactions, note that, with regard to aborted transactions, SIR-SBD yields results that are similar to those of SIR-BD, but it fails to provide good results for committed transactions. As already explained above, this is due to its lack of optimisations for concurrent writeset application. By increasing the number of replicas in our tests, the load increases proportionally, but SIR-SBD does not scale appropriately. This shows that the optimisations of the original SI-Rep protocol were appropriate for providing adequate scalability. Thus, for the test series with eight replicas, we continue without further discussion of SIR-SBD. We have shown that our conflict detection mechanism is able to simplify concurrency control at the middleware layer, and that its support is sufficient for light loads. For heavy loads, other optimisations are still needed (such as those included in the original SI-Rep, as maintained in both SIR and SIR-BD) in order to ensure the protocol's scalability. In spite of this, when long transactions are used, the results of SIR-SBD are better than those of SIR. So, our conflict detection

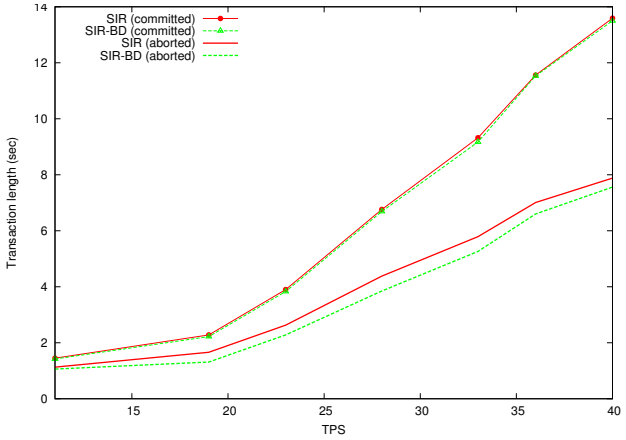


(a) Short transactions (delay=1sec)

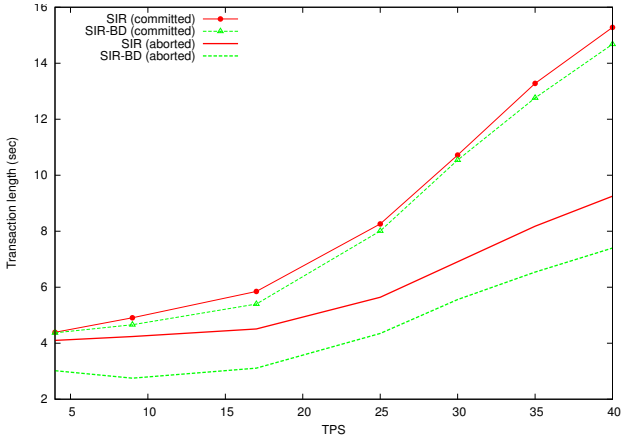


(b) Long transactions (delay=4sec)

Figure 6. Performance results for 8 replicas



Short transactions (delay=1sec)



Long transactions (delay=4sec)

Figure 7. Results varying the system load

technique seems to be good enough for this kind of scenarios where its results are comparable to all the optimisations made in the original SIR implementation.

The third set of experiments has been run on a cluster of eight replicas. It confirms the trend suggested by the previous test with four replicas; i.e., the results are the better the more replicas are added to the system.

For short aborted transactions, the performance gain of SIR-BD over SIR ranges from 10% with writesets consisting of less than 20 items to 22% for writesets of 100 items, which is slightly better than with 4 replicas. Short committed transactions have a performance gain ranging from 3.2% for writesets with less than 10 items to 5.4% for writesets of 100 items. Again, these results are better than those obtained with four replicas.

For long transactions better results are also obtained using SIR-BD. For aborted transactions, the performance gains range from 32% (small writesets) to 38% (large writesets), while for committed transactions, the gain varies between 3.7% and 11%.

All these results have shown that our conflict detection technique is able to provide better performance than other middleware-based approaches using different transaction lengths and different numbers of replicas. Moreover, it scales quite well when the number of replicas is increased. However, no result has been given varying the system load; i.e., the number of started transactions per second. To remedy this, figure 7 shows the behaviour of our technique when the system load ranges between 11 and 40 TPS for short transactions, and between 4 and 40 TPS for long transactions. To this end, we have used again the simplified database schema of the previous experiments, combined with the same kind of transactions, accessing in this case to 20 items of the single database table. Although at first sight the load introduced in these experiments seems light, we have to remark that the transactions being used are

quite long in both cases. Some standard benchmarks, like TPC-W [17], are able to introduce higher TPS values, but use short transactions that do not consider any user thinking time (and that can be finished in a few milliseconds).

As this figure shows, the SIR-BD results are better than those of the SIR implementation, in both kinds of transactions (short and long). Moreover, as it has already happened in the other figures, the differences are higher when long transactions are used. The aim of this figure is to show that such good results do not depend on the system load, as it can be seen in both parts of Fig. 7, since the differences between SIR and SIR-BD remain more or less constant in all the studied range of TPS, both for committed and aborted transactions. This also shows that the overhead introduced by our reading thread is minimal, and that it does not introduce any bad behaviour when the load is increased.

Summarising the experimental results, it is fair to say that our conflict detection technique improves traditional middleware-based writeset conflict management techniques in terms of row-level granularity. Additionally, our technique can be used in almost all replication protocols, since it is able to manage any kind of conflict between transactions. For instance, if read-write conflicts must be managed, we only need to select an underlying isolation level that blocks transactions whenever such a conflict arises. A good rule of thumb is to always use the isolation level of the underlying DBMS that is closest to the level correspondingly expected from the replication protocol.

5 Related Work

As opposed to middleware-based solutions, database replication protocols have been implemented in the DBMS core when performance has been at stake [10]. Then, there are no problems for dealing with transaction conflicts, since internal concurrency control mechanisms can be used for transaction handling. This has been a default for many replication protocols [11, 14]. In some of them, depending on the requested isolation level, readsets must also be checked [14].

The non-voting protocol described in [15] is one of the first examples of an implementation on the middleware layer. It has to check for conflicts between pairs of transactions by scanning their writesets. With mechanisms as described in this paper, its performance can be improved, as we have shown for the SI-Rep protocol.

Another possible approach for dealing with conflict detection at the middleware layer consists in using a linear interaction principle as described in [19]. Then, all replicas receive the same updating requests in the same order and perform them according to the concurrency control of the underlying database; i.e., no additional support is needed by the middleware in this case. This technique has been used in some systems, like C-JDBC [3] and RJDBC [5]. It shares the advantages of our conflict detection mechanism, since concurrency control is delegated to the DBMS, but the linear interaction technique demands the propagation of each update sentence to all replicas, which is costly in terms of communication overhead.

With regard to performance, an ideal middleware-based conflict detection mechanism has been proposed in [13]. This solution is based on the definition of conflict classes [16], that however depend on the given application. A possible option is to assign different conflict classes to different database tables. If user transactions are implemented as stored procedures, the middleware is able to know a priori which conflict classes will be used by each transaction. So, conflicts among transactions can be easily detected since the replication protocol only needs to check if the conflict classes of the considered transactions intersect. Thus, instead of writesets, only their conflict classes need to be analysed. Transaction support of the protocols in [13] can be implemented straightforwardly using stored procedures. These protocols may even be developed without stored procedures, though the aforementioned ease of implementation then is forsaken.

Independent of this implementation detail, this approach requires a significant effort to identify the conflict classes to be used by each transaction. A sufficiently fine-grained conflict detection cannot be achieved at runtime, i.e., the solution is inadequate for applications where transaction behaviour cannot be predicted statically, e.g., interactive applications. In our solution, conflicts can be detected with a finer granularity (indeed, row-level granularity, instead of table-level) and no conflict class identification is needed. Thus, it is more flexible than the solution in [13], and may provide better performance results when the conflict rate among transactions is not very low.

6 Conclusions

The conflict detection technique described in this paper can be easily implemented on top of any DBMS as a database replication middleware module. Its main advantages are: (i) database replication protocols may delegate conflict detection to this module, by which the protocols themselves become simpler; (ii) conflict detection can be hooked up with transaction abortion, thus bringing forward in time the decision to abort and hence accelerating the overall transaction processing; (iii) conflict detection is achieved by built-in DBMS functionality, so that an additional error-prone implementation effort is avoided and row-level (instead of table-level) granularity is obtained; (iv) its overall cost is quite low, since only one execution thread is needed that periodically checks one of the system catalogue tables.

We have implemented and tested this mechanism with one of the recent replication protocols that enables the snapshot isolation level. The obtained results prove that the performance of this approach is better than a programmed check at the middleware layer, even if this check is accomplished immediately as in the examples of this paper. Our solution is easily portable from one SQL-based DBMS to another, and it is easily adapted to any database replication protocol that uses the constant interaction approach.

References

[1] Y. Amir, C. Danilov, and J. R. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proc. of the Intnl. Conf. on Dependable Systems and Networks*, pages 327–336, New York, NY, USA, June 2000.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 1–10, San José, CA, USA, May 1995.

[3] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *Proc. of USENIX Annual Technical Conference, FREENIX Track*, pages 9–18, 2004.

[4] Center for Networking and Distributed Systems. SPREAD Web Page, 2005. Accessible in URL: *http://www.spread.org/*, Johns Hopkins Univ., Baltimore, MD, USA.

[5] J. Esparza-Peidro, F. D. Muñoz-Escoí, L. Irún-Briz, and J. M. Bernabéu-Aubán. RJDBC: A simple database replication engine. In *Proc. of the Intern. Conf. on Enterprise Inf. Syst.*, pages 587–590, Porto, Portugal, Apr. 2004.

[6] J. R. González de Mendívil, J. E. Armendáriz, J. R. Garitagoitia, L. Irún-Briz, and F. D. Muñoz-Escoí. Non-blocking ROWA protocols implement generalized snapshot isolation using snapshot isolation replicas. Technical report, ITI-ITE-06/04, Instituto Tecnológico de Informática, Valencia, Spain, July 2006.

[7] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, Canada, 1996.

[8] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. ACM Press, 2nd edition, 1993. ISBN 0-201-62427-3.

[9] L. Irún-Briz, H. Decker, R. de Juan-Marín, F. Castro-Company, J. E. Armendáriz, and F. D. Muñoz-Escoí. MADIS: a slim middleware for database replication. In *Proc. of the 11th Intnl. Euro-Par Conf.*, pages 349–359, Monte de Caparica (Lisbon), Portugal, Sept. 2005. Springer.

[10] B. Kemme. *Database Replication for Clusters of Workstations*. PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, Aug. 2000.

[11] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333–379, Sept. 2000.

[12] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware-based data replication providing snapshot isolation. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, Baltimore, Maryland, USA, June 2005.

[13] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. MIDDLE-R: Consistent database replication at the middleware level. *ACM Trans. on Comp. Sys.*, 23(4):337–374, Nov. 2005.

[14] F. Pedone. *The Database State Machine and Group Communication Issues*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1999.

[15] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. Strong replication in the GlobData middleware. In *Proc. of Workshop on Dependable Middleware-Based Systems (in DSN 2002)*, pages G96–G104, Washington D.C., USA, 2002.

[16] D. Skeen and D. D. Wright. Increasing the availability in partitioned database systems. In *Proc. of Conf. on Principles of Database Systems*, pages 290–299, 1984.

[17] Transaction Processing Performance Council. TPC benchmark W (web commerce) - specification, Dec. 2003. Version 2.0r. Downloadable from: http://www.tpc.org/.

[18] Transaction Processing Performance Council. TPC benchmark C - standard specification, Dec. 2005. Revision 5.6. Downloadable from: http://www.tpc.org/.

[19] M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso. Database replication techniques: A three parameter classification. In *Proc. of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, pages 206–217, Oct. 2000.