Dealing with Writeset Conflicts in Database Replication Middlewares

Technical Report ITI-ITE-05/11

J. E. Armendáriz, F. D. Muñoz, M. I. Ruiz,
J. R. González de Mendívil, L. Irún
enrique.armendariz@unavarra.es, fmunyoz@iti.es, miruifue@iti.es,
mendivil@unavarra.es, lirun@iti.es

Dealing with Writeset Conflicts in Database Replication Middlewares *

Technical Report ITI-ITE-05/11

J. E. Armendáriz[1], F. D. Muñoz[2], M. I. Ruiz[2], J. R. González de Mendívil[1], L. Irún[2]

(1) Depto. de Matemática e Informática
Univ. Pública de Navarra
Campus de Arrosadía
31006 Pamplona, SPAIN
Phone: +34 948 168 056
Fax: +34 948 169 521
email: {enrique.armendariz,mendivil}@unavarra.es

(2) Instituto Tecnológico de Informática
Univ. Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia, SPAIN
Phone: +34 963 877 069
Fax: +34 963 877 239
email: {fmunyoz,miruifue, lirun}@iti.es

Abstract

Many database replication protocols have to deal with finding out whether the writesets of some concurrent transactions have a non-empty intersection, blocking or aborting part of these conflicting transactions in that case. To this end, a possible solution consists in checking these writesets programmatically and this requires some time and computing power. This situation is even worse if the replication support is being provided by a middleware, since there is no help from the DBMS in that layer. This paper analyzes the effect of using the concurrency-control support of the local DBMS for detecting conflicts between local transactions and writesets being applied for remote transactions. This allows some simplifications and performance enhancements in several database replication protocols.

1 Introduction

Many database replication protocols are using the constant interaction approach [20] for update propagation among replicas; i.e., they only propagate their updates using a constant number of multicasts to the rest of replicas (usually a single multicast at the end of the transaction being executed). In these protocols the transaction is executed following a scheme similar to the passive replication model; i.e., one of the replicas directly receives the transaction sentences, and once the commit procedure begins or terminates (depending on the protocol class [8], either eager or lazy, respectively), this master replica gets the transaction writeset and multicasts it to the rest of replicas. Once this writeset is received in those cohort replicas, it has to be applied. There are different techniques for applying the updates included in this writeset, depending on the writeset contents [12]: either SQL sentences or some kind of specially tailored data structure, such as the usual contents of a writeahead log.

When such writeset contents are being applied in the local database replica, some concurrency control issues have to be considered; i.e., these updates may collide with some previous accesses

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

made by other local concurrent transactions, and in some cases this will imply that these local transactions must be aborted or that the writesets may remain blocked until those local transactions have been completed. This is easily achieved if the database replication support has been provided with some extensions to the DBMS core, since the appropriate locks can be trivially requested before the writeset is applied. On the other hand, these checks may be difficult or, at least, costly, if the database replication support is being provided by a middleware.

This paper describes a simple technique for managing concurrency control at the middleware layer with a minimal help from the underlying DBMS. This permits an easy detection of writeset conflicts in the middleware and makes some optimizations possible when we are programming database replication protocols at that layer. To this end, our technique allows the immediate abortion of one of the conflicting transactions, without needing that such transaction requests its termination. As a result, in many replication protocols this will improve their performance, since transactions are aborted sooner and the system becomes ready for processing other active transactions. The performance results in some tests presented in the following sections corroborate these improvements.

Our solution needs to scan some of the system catalog tables in the underlying DBMS, so it is only convenient for DBMSs based on multi-versioned concurrency control (or MVCC, for short). However, this kind of concurrency control is being used in many DBMSs, and one of them (PostgreSQL) has been chosen in our architecture to implement this support.

The rest of this paper is structured as follows. Section 2 describes a scheme for detecting conflicts among writeset applications and local transactions with the help of the DBMS. Later, section 3 describes one protocol [14] and some variations that are used as an example to test the benefits of this conflict detection scheme. Section 4 provides some initial performance results that compare the approaches discussed in the previous section. Finally, sections 5 and 6 present some related work and the conclusions of this paper.

2 A Scheme for Conflict Detection

The support that is described in this section is part of our MADIS architecture [10, 11], and has been included in its bottom layer. Its aim is to detect conflicts between pairs of transactions being managed by our middleware, and it partially depends on some characteristics of the PostgreSQL DBMS that is being used in MADIS. Concretely, it needs that the underlying DBMS uses a MVCC mechanism, since it has to read the contents of one of the system catalog tables, and this may need read locks in non-MVCC DBMSs that will block other concurrent transactions each time such tables are modified by the latter. However the overall mechanisms are easily portable to other MVCC database systems.

MADIS has been implemented in Java and provides a JDBC interface to applications. Its current communication support is being provided by the Spread group communication system [1, 6]. Since this paper is only interested in how MADIS provides support for conflict detection among concurrent transactions, no additional details about the system model are needed. Moreover, the protocols described in the next section are not complete, since nothing has been said about their recovery procedures. Again, these procedures are important for ensuring that the replicated databases implemented using this middleware are actually highly available, but they are not needed for discussing the conflict detection mechanisms.

Besides this, MADIS has been designed for providing support for fully replicated databases; i.e., we have a replica of the database in each machine where MADIS runs and each node has a complete replica of such a database. The replication protocols described in this paper follow this kind of replication. Additionally, all of them can be classified as eager, with constant interaction and non-voting

termination, according to [20].

Coming back to the scheme for conflict detection, it is basically composed by these two elements:

- A function named getBlocked() put into the database schema, that looks for blocked transactions in the *pg_locks* view placed in the PostgreSQL system catalog. It returns a set of pairs composed by the identifier of a blocked transaction and the identifier of the transaction that has caused such a block. If there is no conflict between any pair of transactions when this function is called, it returns an empty result set.

- An execution thread per database that periodically (its period is configurable, and is commonly set to values between 500 and 1000 ms) uses the function described above. Take into account that in multi-versioned DBMSes the read-only operations cannot be blocked. This thread runs at the middleware layer.

  Once this thread has received a non-empty result set it may request the abortion of one of the transactions being involved in that conflict. For this purpose, each transaction may have a priority level assigned to it. By default, this thread takes no action if both transactions have the same priority level, but it aborts the lowest priority transaction otherwise.

This mechanism can be combined with a transaction priority scheme in the replication protocol. The O2PL [4] BULLY variation described in [2] uses this priority scheme as follows. Two priority classes are defined, 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 for those transactions associated to delivered write sets that have to be locally applied. Once a conflict is detected, if the transactions have different priorities, then that with the lowest priority will be aborted. Otherwise, i.e., when both transactions have the same priority, a second priority function is applied for transactions at level 1, avoiding thus deadlocks, but nothing is done with transactions belonging to level 0. Similar approaches may be followed in other replication protocols that belong to the update everywhere with constant interaction class [20], as we will describe in the next section.

Finally, it is worth noting that this scheme is only valid for active transactions; i.e., for transactions that are running when such a conflict detection is being made. This may be an advantage when the conflict arises between a local transaction that has not arrived yet to its commit request, and a writeset that belongs to a remote transaction that has to be committed. In these cases, the regular solution to this conflict requires the abortion of the local transaction and this may be easily completed in our scheme. On the other hand, other protocols may require to check for conflicts between different remotely received writesets that belong to transactions executed in different nodes. In that case, our scheme is not valid until those writesets are applied to the local database replica, and the conflict may only happen if both writesets have been concurrently put into the database replica, but several protocols are not using this second way of handling remote writesets. Moreover, the first situation described in this paragraph may occur in all replication protocols, and as a result our technique may be appropriate for many of them.

3 A Case of Study: The SI-Rep Protocol

We have selected the SI-Rep protocol described in [14] as a case of study for our conflict detection mechanism. There are several reasons that justify this selection: (i) It is a good protocol for guaranteeing the *snapshot* isolation level [3] (at least, its 1CSI variation as described in [14]), as it has been proven in its performance measurements. (ii) Its description includes a lot of implementation

optimizations that improve its overall performance. So, we may be able to test our conflict detection technique on the optimized version of this protocol, comparing it to the same implementation using another detection approach, and with the non-optimized version with our detection approach.

According to [3], the snapshot isolation level needs two timestamps to be assigned to any transaction. The first one is its start timestamp, that must be set when the transaction starts its first database access. The second one is its commit timestamp, that has to be assigned when such a transaction successfully commits. Both timestamps are usually logical timestamps implemented as counters. In order to commit a transaction T_1, a single rule has to be respected in a centralized implementation of this isolation level: *No other transaction T_2 with a commit timestamp in T_1's interval [start(T_1), commit(T_1)] wrote data that T_1 also wrote.* We need this correctness rule for understanding the SI-Rep protocol that we describe on the sequel, and to justify that our blocking detection technique does not violate the correctness criteria of the original protocol.

This protocol uses an atomic multicast [9], 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 already delivered 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 the following (further details can be found in [14]):

I. When an operation for transaction T_i arrives, if it is the first one, it is blocked until no hole appears in the local *tocommit_queue*. On the other hand, if it is a commit request, the writeset for that transaction is retrieved (if it is empty, the transaction can be committed immediately and no other checking is needed), and checked for conflicts with all the current contents of *tocommit_queue*. If some conflicting writeset is present, T_i is aborted, otherwise its *cert* attribute is assigned to the current value of the counter of globally validated transactions and its writeset is multicast to all replicas.

    NOTE: There are two optimizations in this first step. The first one is related to the presence of holes in *tocommit_queue*, that will be caused because writesets may be concurrently applied to the underlying database as soon as they are delivered and globally checked. If one of the writesets is successfully applied before all its predecessors in that queue, a hole is generated, and this prevents local transactions from being started, since they may read an invalid snapshot. The second optimization is related to the scanning of *tocommit_queue* before multicasting the transaction writeset. Thus, the protocol may find out as soon as possible if conflicts arose between this transaction and those that are trying to commit now. Take into account that the writesets of this queue have been delivered before T_i's writeset is multicast, so they will have a commit timestamp in the [start(T_i), commit(T_i)] interval and will lead to the abortion of T_i.

II. When T_i writeset is delivered in total order, its writeset contents are checked with those of all transactions in *ws_list* and T_i will be rejected (i.e., aborted in its local node, and discarded in all others) if exists some transaction T_j whose *tid* is greater than T_i.cert, and their writesets have a non-empty intersection. Otherwise, if this checking has been successfully passed, the counter of globally validated transactions is increased and the *tid* attribute of this transaction receives its value. Additionally, T_i's writeset is appended to the current contents of both *tocommit_queue* and *ws_list*.

III. Finally, when no conflicting transaction precedes T_i in *tocommit_queue* and it is local or no local transaction is trying to start or no new holes will appear, then T_i is locally applied and committed in the underlying database replica and it is removed from *tocommit_queue*.

As it can be seen in the previous description, there are different conflict detection checkings in this protocol. The first one is needed in step I for validating locally the transaction before its writeset is multicast. Basically, it is checking 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 analyzed transaction and that may also lead to its abortion. Those transactions are checked in the global validation action performed in step II. Finally, the writeset application procedure described in step III also needs to check conflicts among writesets for deciding when each writeset may be securely applied. This last checking is only needed for allowing the concurrent execution of non-conflicting writesets; i.e., for optimizing the protocol performance. Additionally, those checks made in step I are also optional, since the checking being made in step II is a superset of them, but they have been implemented in that paper since they are able to reduce the amount of needed multicasts, aborting earlier some transactions that otherwise would have been aborted in step II.

Finally, the underlying database system is also able to check for conflicts, aborting some transactions if their access patterns violate the rules of the *snapshot* isolation level provided that it is supported by that DBMS. To this end, we have chosen the PostgreSQL DBMS that provides the *serializable* ANSI SQL isolation level using a multi-versioned concurrency control mechanism that is actually only able to ensure the *snapshot* isolation level if we follow the stricter *serializable* definition proposed in [3].

Additionally, the checks natively made by the underlying DBMS may detect also deadlocks and incorrect sequences of updates that might lead to the abortion of a remote transaction, instead of a local one (that would have been finally aborted by the protocol). So, the remote writeset application has to be retried if the underlying DBMS aborts it.

Our conflict detecting approach will be able to abort local conflicting transactions before they are checked in step I; i.e., before a transaction requests its commit. For this purpose, the protocol will assign the highest priority to the application of a writeset, and if such an application collides with a local transaction that has not arrived to its commit request, that local transaction will be aborted. This satisfies the correctness rule described above, since the writeset is associated to a transaction that has successfully passed its global validation phase and that already has a commit timestamp that is trivially included into the [start, commit] interval of the local transaction, since the latter has not yet requested its commit.

We take this SI-Rep protocol as a base to develop its following three implementations:

- **SIR** (*SI-Rep*): In this case, our SI-Rep implementation follows all the steps of the protocol described in [14], scanning the writesets for deciding if they have a non-empty intersection. However, as we will see in the following section, we have chosen in our tests a set of transactions that define a contiguous interval of values in the primary key of the tables being accessed. As a result, it will be quite easy to find out if two writesets collide, given their size and the primary key value of the first accessed row.

- **SIR-BD** (*SI-Rep with Blocking Detection*): In this second alternative, we use the previous implementation as a base, and we add our blocking detection mechanism to improve its behavior. None of the checks done in the previous case has been removed in this second implementation. This ensures the correctness of this alternative, since our blocking detection mechanism will

only abort a transaction when all these conditions are satisfied (in all other cases, no special action is taken):

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

As we have seen above, this approach satisfies the correctness criteria of the *snapshot* isolation level.

Initialization:
1. lastvalidated_tid := 0
2. lastcommitted_tid := 0
3. ws_list := $\emptyset$
4. tocommit_queue_k := $\emptyset$

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
   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 = $\emptyset$, then commit and return
   c. multicast T_i using total order

II. Upon receiving T_i in total order
1. obtain wsmutex
2. if $\exists$ T_j $\in$ ws_list : T_i.cert < T_j.tid $\wedge$ T_i.WS $\cap$ T_j.WS $\neq \emptyset$
   - release wsmutex
   - if T_{ik} is local then abort T_{ik} at R_k else discard
3. else
   - T_i.tid := ++lastvalidated_tid
   - append T_i to ws_list and tocommit_queue_k
   - 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. $\forall$ T_j : T_j is local in R_k $\wedge$ T_j.WS $\cap$ T_i.WS $\neq \emptyset$ $\wedge$ T_j has not arrived to step II
      (this is analyzed 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 1: SIR-SBD Algorithm

- **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 1 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 combining the mandatory writeset check of step II and our blocking detection mechanism we may get acceptable results. Of course, adding the possibility of concurrent execution among non-conflicting writesets provides better performance results in heavily loaded systems.

These protocols have been implemented in the current version of our middleware. Unfortunately, this version is still a prototype and provides bad performance for writeset collection. So, the performance results discussed in the next section are penalized due to this fact.

4 Performance Results

We describe two different classes of tests in the following subsections. In the first one, we analyze different scenarios and test in them all the protocols described in the previous section. In the second subsection we analyze the effects of the conflict detection timeout interval on the performance results, and we see that in the regular range no significant variations arise.

4.1 Protocol Comparison

The protocols described above have been used in some performance tests that have been carried out in our middleware. To this end, a database with only one table has been used. This table has 10000 items with two fields each one. One of these fields is the primary key of this relation and the table has been initialized using the first 10000 natural numbers into such primary key. This simple database schema allows a fast detection of conflicts with row granularity at the middleware layer. A schema with many tables is more realistic, but using it the amount of tests to be done in the middleware grows a lot, and we might obtain better results with our blocking detection mechanism, since its load mainly depends on the work being done at the core of the DBMS for dealing with access conflicts, and the DBMS provides better performance than any solution implemented in the middleware.

These performance tests have used only two replicas of the database previously described. In each replica, 10 concurrent connections have been used, and each connection is used for executing 10 sequential transactions. Each transaction is composed by a number of *update* SQL sentences that explicitly specify the row to be updated with a *where* clause. Moreover, we have tried to generate some abortion rates and transaction lengths varying the number of updates used in each transaction and a delay between these updates. For this purpose, the abortion rates have been configured to these four values: 5%, 10%, 25%, and 50%, using transactions with 20, 40, 100 and 200 updated rows, respectively. The delays between consecutive updates have been configured to the following values: 0, 25, 50, 100, 200, 400, 600, and 1000 ms; giving as a result transaction lengths that range from 0 to 20 seconds (5% abortion rate), from 0 to 40 seconds (10% abortion rate), from 0 to 100 seconds (25% abortion rate), or from 0 to 200 seconds (50% abortion rate).

Figures 2 to 5 show the mean values for the transaction lengths for the four different abortion rates previously discussed. In all figures, all protocols are plotted using two different curves: one for their committed transactions and another for their aborted ones. On the sequel, we will discuss separately the curves for committed and aborted transactions.

4.1.1 Aborted Transactions

There are no significant differences in the four different tested abortion rates among the protocols' behavior. In all cases, both SIR-BD and SIR-SBD are able to abort sooner those transactions that

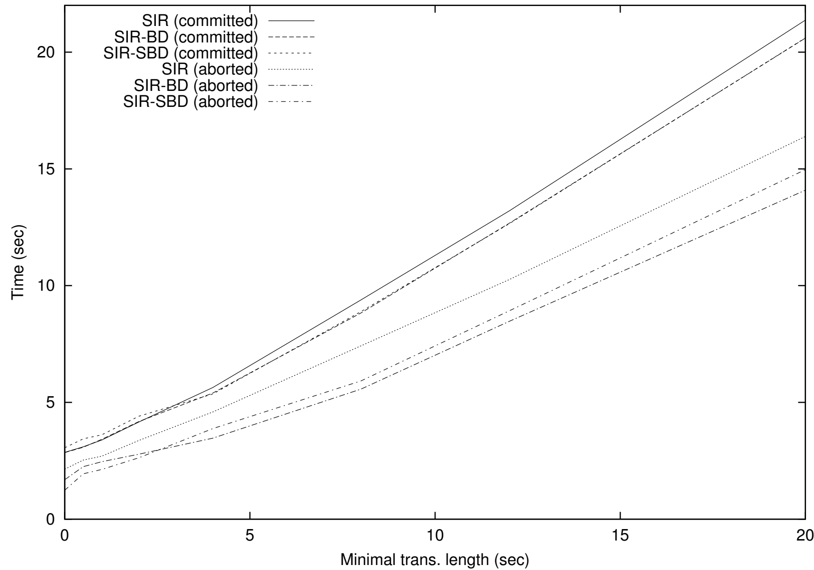

Figure 2: Performance results for a 5% abortion rate.

finally would have been aborted anyway, when they are compared with the original SIR. The SIR-BD implementation shows its best results with long transactions, where its average completion time is between 80% and 85% of the original SIR, being its gains smaller with shorter completion times. On the other hand, the SIR-SBD implementation behaves slightly better than SIR-BD with short transactions and slightly worse with long ones, but being always better than the original SIR. These differences between SIR-BD and SIR-SBD are higher with low abortion rates (i.e., 5%) but almost negligible in the rest (i.e., 10%, 25%, and 50%). In the lowest abortion rate, SIR-SBD behaves "better" than SIR-BD for short transactions due to the fact that finally committed transactions need longer completion times in the former protocol and this implies that a given transaction collides with others that would have not started yet if the latter protocol was used. This slightly enlarges the SIR-SBD's abortion rate for short transactions. Thus, it is actually 4.5% in SIR-SBD, but 3.6% in SIR and 3.2% SIR-BD for the non-delayed transaction set that was designed for generating a 5% average abortion rate. On the other hand, its abortion rate is exactly the same as that of SIR-BD for longer transactions, being both of them always smaller than SIR's.

One additional comment is remarkable regarding aborted transactions. The underlying DBMS has helped also a bit for aborting conflicting transactions. As it can be seen in the figures, once the inter-update delay is longer than 50ms, the aborted transactions last less than their minimal time in all protocols (i.e., the curves for aborted transactions are below the diagonal line in all figures). Since the underlying DBMS uses a MVCC mechanism, transactions only get blocked when they present write-write conflicts between them. If so arises, once the first blocking transaction commits, its blocked counterparts are automatically aborted by the DBMS. This has often happened in our tests, due to the design of the transactions used in them. However, even in these cases, our blocking detection mechanism is able to abort sooner those transactions that finally would have been aborted by the DBMS or by the protocol.

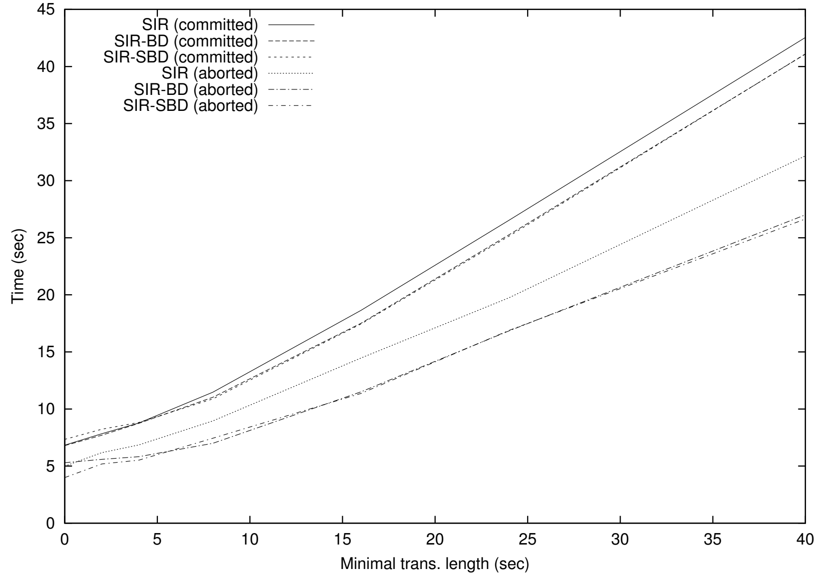

Figure 3: Performance results for a 10% abortion rate.

4.1.2 Committed Transactions

Finally, as a result of the improvement in the completion time of aborted transactions, figures 2 to 5 also show that the completion time of committed transactions is similar in the SIR and SIR-BD protocols when short transactions are considered. In these cases, the SIR-SBD shows a small overhead when compared with these two protocols, except when the abortion rate is greater than 25%. In this latter case, shown in Fig. 5, its overhead is almost 25% of the completion time in SIR and SIR-BD for those transactions with null inter-update delay. Additionally, when transactions have an inter-update delay greater than 100ms (take into account that transactions are aborted sooner in this case by SIR-BD and SIR-SBD), both the SIR-BD and SIR-SBD provide a completion time that is 5% shorter (on average) than the SIR's one.

The SIR-SBD overhead for short transactions is easily explained by its simplifications that have removed the concurrent application of writesets in step III. This provided good results for aborted transactions, since they were aborted even sooner than in the other two protocols; but it had the important drawback of increasing a bit the abortion rate, and also the increase in the completion time of committed transactions that we have analyzed here.

All these figures also show that with long transactions, higher abortion rates (i.e., higher conflict degrees) imply better performance improvements. This implies that with abortion rates smaller than 5% the improvements achieved with this technique will be also less significant.

Anyway, these tests have shown that the optimizations included in the SI-Rep (and implemented as SIR and SIR-BD) provide good results in our system, similar to those described in its original paper [14] when it is compared to a non-optimized version (SIR-SBD in our case). On the other hand, it also shows that our conflict detection mechanism is able to provide good results when transactions are long, simplifying a bit the efforts needed to implement a replication protocol in a middleware.

As a result, we may say that this conflict detection technique boosts the traditional writeset conflict management techniques at the middleware layer when they aim to provide a row-level granularity. However, our technique cannot be used in all database replication protocols, since it is only valid for replication protocols that only need to check for writeset conflicts. Take into account that there are

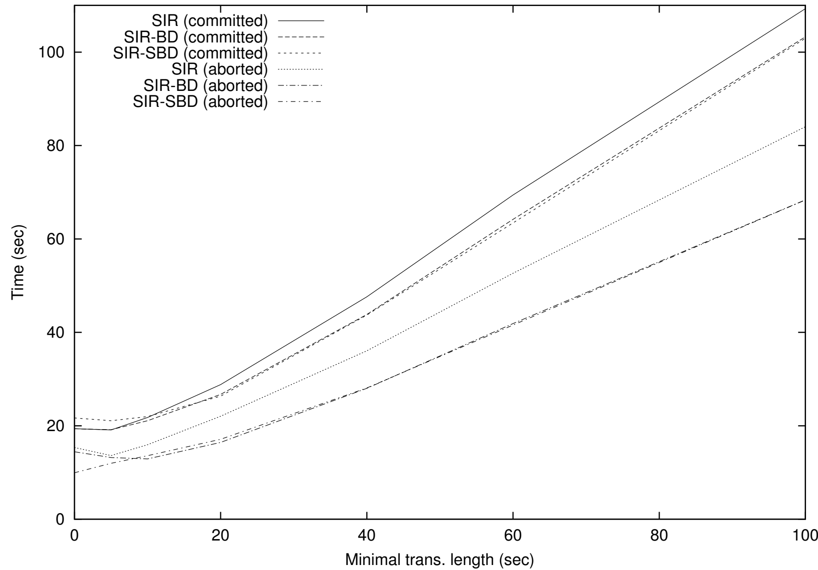

Figure 4: Performance results for a 25% abortion rate.

others that might need readsets [17, 15], or that use object versions for conflict checking [15].

4.2 Timeout Intervals

We compared four different timeout values for our blocking detector: 300 ms, 500ms, 700ms and 1000ms. To this end, we chose a configuration with an abortion rate equal to 25% –i.e., with 100 updates per transaction– and 100 ms of delay between consecutive updates. The results of this test are shown in figure 6, and they are the means of ten different experiments.

These results are quite similar for all timeout values, and only slightly better for the highest one, which has been used in all the tests discussed in the previous section. With small timeouts, the thread associated to blocking detection has to be executed more often, and this introduces some overhead that might only be counterbalanced detecting sooner the conflicts, allowing a faster transaction abortion that will reduce the system load. This has not happened in this tested configuration. Besides this, the differences among tested timeouts have been minimal, and it seems appropriate to use in all tests of the previous section always the same timeout interval. Of course, the optimal timeout value may depend on the system load being supported, but this test proves that the optimization achievable with a fine tuning of the timeout interval will be almost insignificant.

5 Related Work

Database replication protocols have been implemented in the core of the DBMS when performance has been a requirement [12]. In these cases, there are no problems for dealing with writeset conflicts, since the internal concurrency control mechanisms are able to manage appropriately the requests being made for applying the incoming writesets. This has been the default behavior in many database replication protocols [13, 17]. In some of these protocols, depending on the requested isolation level, readsets must be also transmitted and checked [17].

The non-voting protocol described in [18] is one of the first examples of implementation at the

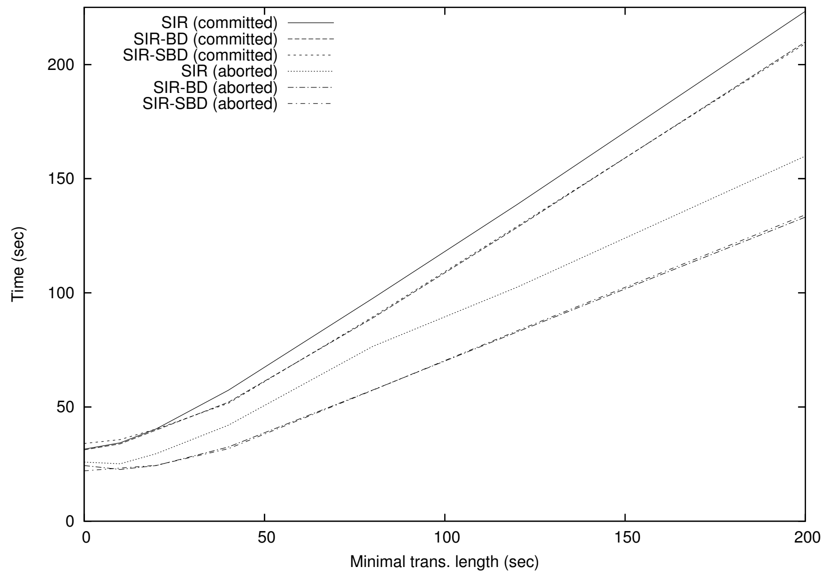

Figure 5: Performance results for a 50% abortion rate.

middleware layer and it has to check for conflicts between pairs of transactions scanning their writesets. A mechanism similar to the one described here would have improved its performance as it has been shown for the SI-Rep one.

Another possible approach for dealing with conflict detection at the middleware layer consists in using a linear interaction principle as described in [20]. In that case, all replicas receive the same updating requests in the same order and perform them following the concurrency control of the underlying database; i.e., no special support is needed by the middleware in this case. This technique has been used in some systems, like C-JDBC [5] and RJDBC [7]. This shares the advantages of our conflict detection mechanism, since concurrency control has been delegated to the DBMS, but in this case the linear interaction technique demands the propagation of each update sentence to all replicas, and this may be costly in terms of communication.

From the point of view of performance, the ideal conflict detection mechanism at the middleware level has been used in [16], since it may be easily implemented and is checked almost immediately and seamlessly. Its solution is based on the definition of conflict classes [19] that are application-dependent. A possible choice is to assign each conflict class to a different database table. If the user transactions are implemented as stored procedures, the middleware is able to know a priori wich conflict classes will be used by each transaction. So, conflicts among transactions can be trivially detected since the replication protocol only needs to check if the conflict classes of the considered transactions actually intersect; i.e., the writesets need not be analyzed, but only their conflict classes. Additionaly, the protocols described in [16] may also implement their transactions without using only stored procedures. So, this solution can be considered optimal, but needs some effort to identify the conflict classes being used by each transaction. Our solution has worst performance than this one, but conflicts may be detected with a finer granularity (indeed, row-level granularity, instead of table-level) and it does not need any assistance for identifying any conflict class, so it is slightly more flexible than the one described in [16].

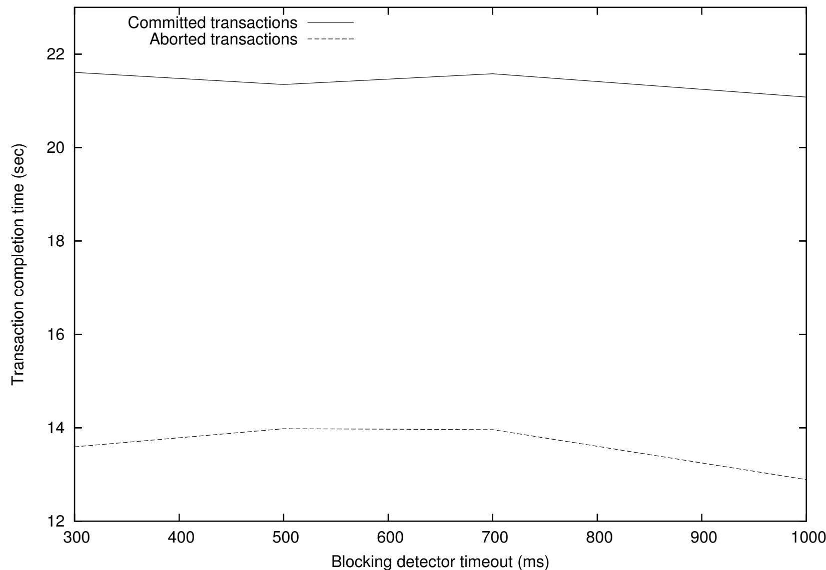

Figure 6: Comparison of blocking detector timeouts.

6 Conclusions

The conflict detection technique described in this paper can be easily implemented in MVCC-based DBMSs for providing assistance to database replication middlewares. Its main advantages are: (i) the resulting database replication protocols may delegate the conflict detection to this scheme, allowing some protocol simplifications, (ii) conflict detection can be associated to transaction abortion, advancing thus the abortion decisions and reducing in this way the overall transaction completion time (to this end, the resulting protocols must use a total-order multicast for propagating the writesets), (iii) conflict detection is actually implemented by the DBMS, so it often has row-level granularity, (iv) its overall cost is quite low, since it only needs an execution thread that periodically checks one of the system catalog tables. On the other hand, its main inconvenient is related to concurrency control, too, since this mechanism can not be used in middlewares that use an underlying lock-based concurrency control DBMS. In that case, the read accesses to the system catalog table will prevent the progression of updates requested by other concurrent transactions. Additionally, this detection mechanism only provides information for write-write conflicts. Other kinds of conflict have to be checked otherwise, so its natural target is the set of protocols with a *snapshot* isolation level.

We have tested this mechanism with one of the recent replication protocols that provides this isolation level. The obtained results prove that this approach has better performance than a programmatical check at the middleware layer, even if this check is accomplished immediately as in the examples provided here. The results can be easily transferred to many other database replication protocols that also use the constant interaction approach without propagating readsets, either with voting or non-voting termination [20].

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] J. E. Armendáriz, J. R. Juárez, I. Unzueta, J. R. Garitagoitia, F. D. Muñoz-Escoí, and L. Irún-Briz. Implementing replication protocols in the MADIS architecture. In *Proc. of the XIII Jornadas de Concurrencia y Sistemas Distribuidos*, Granada, Spain, September 2005.

[3] 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.

[4] M. J. Carey and M. Livny. Conflict detection tradeoffs for replicated data. *ACM Trans. Database Syst.*, 16(4):703–746, 1991.

[5] 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.

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

[7] 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, April 2004.

[8] 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.

[9] 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.

[10] Instituto Tecnológico de Informática. MADIS Web Site, 2005. Accessible in URL: *http://www.iti.es/madis/*.

[11] 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, September 2005. Springer.

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

[13] 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, September 2000.

[14] 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.

[15] F.D. Muñoz-Escoí, L. Irún-Briz, P. Galdámez, J.M. Bernabéu-Aubán, J. Bataller, and M.C. Bañuls. GlobData: Consistency protocols for replicated databases. In *Proc. of the IEEE-YUFORIC'2001*, pages 97–104, Valencia, Spain, November 2001.

[16] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Consistent database replication at the middleware level. *ACM Trans. on Comp. Sys.*, 2005. In press.

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

[18] 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.

[19] 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.

[20] 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, October 2000.