Integrity Dangers in Certification-Based Replication Protocols

M. I. Ruiz, F. D. Muñoz, H. Decker, J. E. Armendáriz, J. R. González de Mendívil

Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia, Spain

Depto. de Ing. Matemática e Informática
Univ. Pública de Navarra
Campus de Arrosadía, s/n
31006 Pamplona, Spain

{miruifue,fmunyoz,hendrik}@iti.upv.es{enrique.armendariz,mendivil}@unavarra.es

Technical Report TR-ITI-ITE-08/13

Integrity Dangers in Certification-Based Replication Protocols

M. I. Ruiz, F. D. Muñoz, H. Decker, J. E. Armendáriz, J. R. González de Mendívil

Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia, Spain

Depto. de Ing. Matemática e Informática
Univ. Pública de Navarra
Campus de Arrosadía, s/n
31006 Pamplona, Spain

Technical Report TR-ITI-ITE-08/13

e-mail:
{miruifue,fmunyoz,hendrik}@iti.upv.es{enrique.armendariz,mendivil}@unavarra.es

May, 2007

Abstract

Database replication protocols check read-write and/or write-write conflicts. If there are none, then protocols propagate transactions to the database, assuming they will eventually commit. But commitment may fail due to integrity constraints violations. Also, the read actions of integrity checking may give raise to new conflicts. Thus, some more care must be taken if, in addition to the consistency of transactions and replicas, also the consistency of integrity constraints is to be maintained. In this paper, we investigate how certification-based replication protocols can be adapted to correctly and transparently deal with the built-in integrity support provided by the underlying DBMS. Also, we experimentally demonstrate the negative effects that an incorrect management of integrity constraints may cause in a database replication distributed system.

1 Introduction

Many database replication protocols have been proposed [3, 4, 10, 13, 23] over the years. None of these proposals has assessed the support of semantic consistency as postulated by integrity constraints. In general, the ignorance of problems related to integrity checking in concurrent, distributed and replicated systems is in good company. Even the most well-known authors in the field of transaction processing are accustomed to assume that all transactions are programmed in such a way that they preserve integrity when executed in isolation, and therefore, integrity preservation is also guaranteed by serializable schedules of concurrent executions[9, 3].

Unfortunately, this assumption does not always apply. And even if it would, its consequence does not necessarily hold in replicated databases. Concurrent transactions may start and execute in different nodes, and proceed without problems until they request commitment. Upon receipt of the commit request of a transaction, the replication protocol validates it and guarantees its commit if there is no read-write or write-write conflict among concurrent transactions. Complications may arise if constraints are checked in deferred mode, i.e., at effective commit time, i.e., only after conflict validation by the replication protocol. Then, integrity checking may diagnose constraint violations by transactions that are already successfully validated by the protocol, i.e., the protocol has already sanctioned those transactions to commit. Moreover, integrity checking may increase the transaction's readset, remaining these new accesses unnoticed by the protocol.

On the other hand, if the replication protocol guarantees a serializable isolation level and all checks are immediate, all accesses made during integrity checking will be contained in the readset of the transaction. Thus, other concurrent transactions are prevented from accessing the same items, even if the commitment of both transactions does not violate any integrity constraint. However, a deferred checking still presents the problems mentioned above.

Read actions for checking integrity need to be taken into account by a scheduler of a set of concurrent transactions just as any other action of the given transactions. Hence, the sequentializability of a set of concurrent transactions may change if also actions for integrity checking are taken, concurrently. It seems that there are two possibilities for taking integrity checking actions into account: either as a separate concurrent transaction, or as actions by which the checked transaction is augmented. In middleware-based replication protocols, integrity checking actions are treated as if they would need no scheduling. In particular, it is not clear how "immediate" checks are scheduled. Each change of a schedule can, however, destroy the sequentializability of the schedule. For this reason, and also because the semantics of "immediate" usually is proprietary, "immediate" checks can not be considered as legal actions in general.

On the other hand, if "deferred" checking means that all checks are scheduled after all actions of a given transaction have been processed, then these deferred checks still need to be scheduled with regard to concurrent actions, for avoiding sequentializability problems.

If a transaction T is augmented by integrity checks, a new transaction T' results. If all transactions T_1,...,T_n concurrent with T are also augmented by such checks, a new set T',T'_1,...,T'_n of transactions results. Then, the serializability of T',T'_1,...,T'_n is not the same as serializability of T,T_1,...,T_n. Thus, a new serializability condition is needed: A set of transactions is *serializable with regard to integrity* if its augmentation with integrity checks is serializable in the traditional sense.

Things may get even worse when isolation levels are relaxed. Recently, the *snapshot* isolation level [1] has gained popularity, since it is supported by several DBMSs (e.g., Oracle, PostgreSQL, Microsoft SQL Server, ...), though it does not avoid all isolation anomalies. Although it can support serializable executions [8], the replication protocols that support it [13, 14, 6, 17] are able to relax the final transaction validation process, since only write-write conflicts need to be checked for this isolation level. This implies that the abortion rate generated by the replication protocol is lower than using serializable isolation. That, however, may cause bigger problems for integrity maintenance, since more protocol-accepted transactions will be involved in constraint-related problems at commit time. Moreover, other relaxed isolation levels like *read committed*, which may make sense for some kinds of applications, and some replication protocols that support them [2, 20], are facing the same problem as discussed above.

This way, if the replication protocol only guarantees a snapshot isolation level (abbr., SI), only write-write conflicts will be checked during the validation phase. Thus, for SI, a state transition that preserves the integrity of the database can be guaranteed only if integrity is checked in deferred mode. For instance, suppose an integrity constraint that limits the value of the sum of two different columns x and y. Assume also two concurrent transactions A and B being executed in different nodes: A only updates x, so y is read to check integrity; in the other node, B only updates y, so x is read for integrity checking. Suppose that both integrity checkings are passed in their respective local nodes, but the updates are such that, when combined, the integrity constraint is violated. The validation phase of the protocol will not detect any write-write conflict between A and B and thus will allow both transactions to commit, which leads to an integrity inconsistent database state. Although multi-column `CHECK` constraints are not usually supported in DBMSs, the same situation can be reached with a simpler constraint for a single column x, like `CHECK SUM(x) < Threshold`, where `Threshold` is a constant. In this case, with `SUM(x) = 8` in the old state and a threshold of 10, two transactions A and B both inserting a new row with value 1 in column x will successfully pass their local integrity checkings, but cause a integrity violation when combined. For this reason, such constraints must be checked at commit time, in order to consider the updates made by previous remote transactions.

Apart from the two options considered until now, i.e., making constraint checking immediately after the update action or deferring this checking until commit time, there is a third possibility, proposed by some DBMS vendors, consisting in making constraints immediate at the end of a transaction. This is a way of checking whether a commit operation can succeed. Thus, the transaction programmer can avoid unexpected rollbacks by setting constraints IMMEDIATE as the last statement in a transaction. If any constraint fails the check, the programmer can then correct the error before requesting the commit of the transaction.

However, such techniques can be dismissed because they are non-standardized proprietary tricks, requiring an additional non-declarative programming effort for transactions. Also the constructs IMMEDIATE and DEFERRED are procedural. That is against the principle of declarativity, which means that constraints are stated without control structures, so that the evaluation (checking) of constraints is completely transparent to the transaction designer/programmer. Procedural constructs tend to be error-prone and hard to maintain, in particular if they are not standard. Moreover, although there is a substantial difference for the transaction programmer between using this trick and declaring constraints IMMEDIATE from the beginning of the transaction, from the point of view of the replication protocol there is no difference, as both checkings are done before the local commit request of the transaction. Thus, the updates made by previous remote transactions are not considered in these checkings, leading to the same problem described above.

Something similar will happen with foreign key constraints, which are supported by all DBMSs supporting SI. Suppose a table constraint like `FOREIGN KEY x REFERENCES t.y`, where `t` is the referenced table and `y` is the primary key column of `t`. If an INSERT inserts a row with value `x = v` and a concurrent DELETE deletes the row in `t` with column value `y = v`, then a violation of the foreign key constraint remains undetected by local immediate checks.

Thus, there may be transactions that are sanctioned to commit by the replication protocol but have to be aborted later, due to integrity violations detected by deferred checking. Recall that to abort after commit is against the basic principles of transaction processing.

Most modern database replication protocols use total order broadcast for propagating sentences or writesets/readsets of transactions to other replicas [23]. Among them, certification-based replication protocols provide good performance by optimised algorithms, such as [14]. In this paper, we are going to analyse the integrity checking support needed for certification-based replication protocols.

This paper follows up to [16]. As far as the authors are aware, our work is the first to study the integrity support by replication protocols. If so, then, apart from inaugurating this subfield of research, its main contribution consists in identifying how replication protocols can be adapted in order to correctly deal with integrity constraints. This paper also contains a practical study of the negative effects of improperly managing integrity constraints in a certification-based replication protocol. Our study reflects that an incorrect handling of integrity-violating transactions in a distributed system leads, at least, to a higher abortion rate. Moreover, incorrectly aborted transactions might have led, if not aborted, to the correct abortion of concurrent conflicting transactions that are then incorrectly committed.

The rest of this paper is structured as follows. In section 2 we detail the assumed system model. Section 3 briefly explains integrity constraints and section 4 explains in detail the certification-based replication and the extensions made for a proper management of integrity constraints in this type of replication protocols. In section 5 we present the experimental results obtained, detailing the tests performed and the observed protocol behavior. Section 6 analyses the problems that other works present with regard to integrity consistency. Finally, section 7 concludes the paper.

2 System Model

We assume a partially synchronous distributed system of finitely many nodes with the following characteristics. Clocks are not synchronized but message transmission time is bounded. Each node holds a replica of a given database; i.e., the database is fully replicated. Each system node has a local DBMS that is used for local transaction management. On top of the DBMS, the middleware system MADIS [12, 17] is deployed in order to provide support for replication. This middleware uses a group communication service (abbr., GCS), that provides a communication and a membership service supporting virtual synchrony [5]. The communication service uses a total order multicast for message exchange among nodes through reliable channels. The GCS groups messages delivered in views [5]. The uniform reliable multicast facility [11] ensures that, if a multicast message is delivered by a node (correct or not), then it will be delivered to all available nodes in that view. In this work, we use Spread [21] as our GCS.

3 Integrity Constraints

Integrity constraints (shortly, constraints) define what is a consistent database state, by requiring that certain conditions be invariant across updates. Consistency as defined by constraints is sometimes called *semantic consistency*. This emphasises that constraints express properties pertaining to the application domain of the database. This nomenclature also serves to distinguish semantic consistency from *transaction consistency*, which involves guarantees of atomicity and isolation, and from *replication consistency*, which requires that the states of replicated database nodes coincide on the values of their individual copies of common data items.

Constraints can be classified by well-known criteria such as declarative or procedural; static or dynamic; column, table or inter-table constraints [15]. In this paper, we only consider the effects of not correctly coordinating integrity checks, as provided by the DBMS at hand, with the actions taken by the replication protocol, regardless of the integrity constraint type.

But, since we build on existing DBMS support for integrity checking, the common concept in current SQL implementations of the *checking mode* is important. As previously commented, this mode specifies when a declarative constraint is to be checked: either *immediate*, i.e., directly upon the given update action, or *deferred*, i.e., delayed until the transaction containing the update switches to immediate-mode or, by default, right before the commit of the transaction. In general, the checking mode is controlled by the transaction programmer or, if the transaction is dynamic, by the agent who executes the transaction.

4 Certification-Based Database Replication Protocols

Certification-based database replication protocols (abbr., CBR) use a total order broadcast mechanism [5] for update propagation and replica coordination. As defined in [23], such protocols proceed along the following sequence of steps:

1. A transaction T is executed in a single replica, the delegate one.
2. When T locally requests its commit, its readset and writeset are collected.
3. A message is broadcast to all replicas in total order, propagating T's writeset and readset.
4. On T's data delivery, T is validated against concurrent transactions, looking for read-write and write-write conflicts. This validation stage is symmetrical since all replicas can hold the same history list of previously delivered readsets and writesets, i.e. no additional communication step is needed for that.
5. If the delegate replica has not found any conflict, the protocol is able to reply to the client, notifying T's success. Moreover, in all replicas, T's data is added to a list of to-be-committed transactions and also to the history list used for evaluating conflicts with incoming ones.
6. If a conflict is found, T is aborted in its delegate replica, and its data are discarded in all other replicas.
7. If no conflict is found, T can commit in each replica. To this end, its writeset is applied. At the end of this step, a local commit operation is executed in all replicas in order to commit T. If writeset application is impeded, for instance by T being involved in a deadlock and aborted by the DBMS, then writeset application is reattempted until it succeeds.

Note that readset collection and propagation can be costly if row-level granularity instead of table-level granularity is used. So, in practice, certification-based protocols are rarely used for implementing serializable isolation. On the other hand, CBR is the preferred protocol class when the *snapshot* isolation (abbr., SI) level [1] is supported, mainly because this level relies on multiversion concurrency control, and readsets do not need to be checked in the certification step. On the other hand, since such certification is based on logical timestamps and depends on the length of transactions, a list of previously accepted certified transactions is needed for certifying the incoming ones. Moreover, we have also proved recently [20] that

the *read committed* isolation level can be implemented by using the CBR protocol class, demanding a certification strategy quite similar to the one used in SI.

So, we focus on SI-oriented CBR protocols in this paper. A general protocol of this kind is displayed in figure 1a. The following symbols are used: t is the transaction being processed; R is the set of alive replicas; r_i is the local replica executing the protocol; r_d is the delegate replica of a transaction, i.e., the replica where that transaction started; c is the client process; DB is the local DBMS interface accessed by the replication protocol; and *wset*(t) is the writeset of t. Note that the $DB.abort(t)$ operation is executed in non-delegate replicas without having previously applied t's updates. If that happens, t's writeset must be discarded and, obviously, no operation will be requested to the underlying DBMS. The symbol $wslist_i$ is needed for representing the list of successfully certified writesets in replica r_i, also called the history list. A writeset should be added to that list in step 8 of this new protocol, once it has been accepted for commitment. Thus, the list might grow indefinitely. To avoid that, the list can be pruned following the suggestions given in [23]. Accesses to this list are confined within mutually exclusive zones delimited by $mutex.lock$ and $mutex.unlock$ calls. Note that a $mutex.unlock$ call has no effect if the mutual exclusion was already ended (line 13 in figure 1a has no effect for transactions successfully certified as they leave the protected section in line 8a).

As already indicated, certification-based replication gives rise to several problems with regard to integrity constraints. If there is any deferrable declarative constraint and a transaction requests deferred checking, that checking can not be done until the last step in the sequence presented at the beginning of this section. However, as we have seen, that may lead to constraint violation and abortion behind schedule. In this case, any repeated attempts to commit the transaction clearly would be in vain. Also note that in step 5 of the same sequence the transaction was assumed successful, and other transactions whose data were delivered after T may have already been aborted due to T's assumed commit. So, certification-based protocols need to be modified in order to correctly deal with deferrable constraints. We discuss such modifications below.

Replication protocols are assumed to be implemented, as usual, either in a middleware or as a direct extension of the DBMS core. In each case, the DBMS is assumed to directly provide support for integrity maintenance, by raising exceptions or reporting errors in case of constraint violation. Such exceptions and error messages are then managed by the replication protocol. Thus, they do not reach the user-level application, unless the replication protocol decides so. We also assume that the DBMS is able to support the isolation level for which the replication protocol has been conceived. Thus, the replication protocol may focus on its native purpose of ensuring replica consistency, and delegate local concurrency control to the DBMS.

1: Execute t.	1: Execute t.
2: On t commit request:	2: On t commit request:
3: TO-bcast(R, $\langle$wset(t), $r_i\rangle$)	3: TO-bcast(R, $\langle$wset(t), $r_i\rangle$)
4: Upon $\langle$wset(t), $r_d\rangle$ reception:	4: Upon $\langle$wset(t), $r_d\rangle$ reception:
5: $mutex.lock$	5: $mutex.lock$
6: $status_t \leftarrow$ certify(wset(t), $wslist_i$)	6: $status_t \leftarrow$ certify(wset(t), $wslist_i$)
7: if ($status_t$ = commit) then	7: if ($status_t$ = commit) then
8: append($wslist_i$, wset(t))	8: append($wslist_i$, wset(t))
8a: $mutex.unlock$	
9: if ($r_i \neq r_d$) then	9: if ($r_i \neq r_d$) then
10: DB.apply(wset(t))	10: DB.apply(wset(t))
11: DB.commit(t)	11: $status_t \leftarrow$ DB.commit(t)
	11a: if ($status_t$ = abort) then
	11b: remove($wslist_i$, wset(t))
12: else DB.abort(t)	12: else DB.abort(t)
13: $mutex.unlock$	13: $mutex.unlock$
14: if (r_i = r_d) then	14: if (r_i = r_d) then
15: send(c, $status_t$)	15: send(c, $status_t$)
a) SI CBR protocol.	b) Extended SI CBR protocol.

Figure 1: SI and Extended SI certification-based protocols.

The extensions for managing integrity constraints in SI CBR protocols, as displayed in figure 1b, seem to be minor. Only a slight modification of the original line 11 is needed, for recording the result of the commit attempt. If such commit attempt of some transaction t failed due to integrity violation (but not if failure is due to other causes, since such abortions are indefinitely reattempted), then the writeset of t should be removed from the $wslist_i$, since it has not been finally accepted. This is done in lines 11a and 11b.

However, these seemingly minor extensions may have a notable impact on system performance. Typical SI CBR protocols [6, 14, 17, 7] use some optimisations in order to achieve good performance. One such optimisation consists in minimising the set of operations to be executed in mutual exclusion (i.e., avoiding new remote writeset processing) in the part of the protocol devoted to managing incoming messages. In many protocols (e.g., [14, 17]), an auxiliary list is used for storing the writesets to be committed (the related protocol section in fig. 1a only encompasses lines 6 to 8). As a result, new certifications can be made, once the current writeset has been accepted. With our extensions, no new writeset can be certified until a firm decision on the current one has been taken. That only happens after line 11b in fig. 1b; i.e., once the writeset has been applied in the DBMS and its commit has been requested. This might take quite some time, and must be done one writeset at a time.

5 Experimental Analysis

For analysing the protocol extensions proposed in section 4 by practical experiments, we have implemented three SI CBR protocol versions: a) IntUnaware –integrity-unaware protocol– corresponds to the pseudocode shown in figure 1a with only two modifications: first, it is able to identify those transactions that raise integrity exceptions when tried to be committed and, thus, it does not indefinitely reattempt them (we introduced this extension in order to obtain a protocol that keeps liveness even though it still improperly manages integrity consistency), and second, it informs clients with the real final status of transactions in line 15; b) IntAware –integrity-aware– protocol, which corresponds to the pseudocode shown in figure 1b; and c) IntAwareOpt, an optimised version of the previous protocol.

In short, the integrity management error made by the IntUnaware protocol is to keep in the history list those transactions that were aborted due to integrity violations. At this point, the reader may ask why we analyse the IntUnaware protocol at all, since its behavior is incorrect and should be replaced by a correct version such as the IntAware protocol. The reason is that no better protocols yet exist for replicated databases. Therefore, we want to analyse the difference of performance between existing protocols such as the IntUnaware protocol and our proposal to make it correct, as embodied by the IntAware protocol. This performance analysis also is interesting because, in many situations and applications (e.g., data warehousing), proper constraint checking is often disabled, which means that using either the IntUnaware or the IntAware version will have the same (possibly incorrect) effect. Hence, the differences of performance of both protocol variants are of practical interest.

For this reason, the third protocol version was introduced in the analysis. The IntAwareOpt protocol was designed to reduce the completion time of aborted transactions of the IntAware version, as we observed that the difference in completion times between the IntUnaware and the IntAware protocols was considerable higher for aborted transactions. With this end, the IntAwareOpt protocol starts the certification of a writeset W as soon as it is delivered, without waiting for the completion of previous transactions, as does the IntAware protocol. In order to maintain correctness, this certification phase establishes dependencies for W, in such a way that the certification is only definitive when the protocol has a firm decision for all previous conflicting transactions, i.e., all previous transactions that conflict with W have been committed or aborted. This optimisation brings forward the certification decision of abortion for a transaction T, as the successful commitment of only one concurrent conflicting transaction is enough to determine that T must abort.

The last performance issue considered in this study was the accurate tunning of an important module of our middleware: the BlockDetector. This module uses the concurrency control support of the local DBMS for detecting conflicts between local transactions and writesets of remote transactions. Periodically, this module executes a stored procedure that looks up blocked transactions in the DBMS metadata. In short, it reads a system catalogue table in which the DBMS keeps information about transaction conflicts. Such

a table is maintained by most 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. This mechanism, combined with a transaction priority scheme in the replication protocol, allows the BlockDetector to early abort those local transactions that, anyway, the protocol would have also aborted due to conflicts with previous and concurrent transactions already successfully certified. More detailed information about this module, along with performance results, can be found in [17].

As already mentioned, the BlockDetector lowers the completion time for local aborted transactions. But two considerations have to be made: first, this module aborts a local low-priority transaction only if its block is related to another transaction that has been already marked by the protocol with a higher priority, as a result of a successful certification; and second, the BlockDetector is implemented as a thread that cyclically –its period is defined by a timeout– obtains information about current blocks and takes the appropriate actions. This way, the sooner the protocol successfully certifies transactions –thus increasing their priority–, and also the shorter the timeout of the BlockDetector, the earlier the local to-be-aborted transactions will actually abort. The first condition motivated the inclusion of the IntAwareOpt version in the study. The second one, led us to test performance with regard to different values for the BlockDetector timeout.

To accomplish the analysis, we use PostgreSQL [19] as the underlying DBMS, and a database with two tables, *tbl1* and *tbl2*, each with 10,000 rows and two columns. The first of each table is declared as its primary key. The second column of *tbl2* is a foreign key, set to be evaluated in *deferred* mode, that references the first column of *tbl1*. The second column of *tbl1* is an integer field that is subject to updates made by transactions.

Two types of transactions have been used: a) transactions that preserve integrity, called IntS –integrity safe– transactions; and b) transactions that violate an integrity constraint that has been satisfied before their execution. These last transactions update the foreign key column of *tbl2* with a value that has no reference value in the primary key column of *tbl1*, and are called IntV –integrity violating– transactions. In the test runs of our analysis, we have varied the proportions of IntS and IntV transactions. More precisely, we analysed test runs with 0, 10, 20, 30, 40 and 50% of IntV transactions. Although these percentages reach an artificially high level, we wanted to measure the consistency degradation as more and more transactions are improperly added to the history list. Thus, all shown graphs display this percentage in their x axis.

Both protocols have been tested using MADIS with 3 replica nodes. Each node has an AMD Athlon(tm) 64 Processor at 2.0 GHz with 2 GB of RAM running Linux Fedora Core 5 with PostgreSQL 8.1.4 and Sun Java 1.5.0. They are interconnected by a 1 Gbit/s Ethernet. In each replica, there are 4 concurrent clients, each of them executing a stream of sequential transactions, each one accessing a fixed number of 20 items for writing, with a fixed pause of 500 ms between each consecutive transaction.

In order to clearly show the differences in the decisions made by each protocol, each replica node works with a different one: IntUnaware, IntAware or IntAwareOpt protocol versions. So, for convenience, we may identifiy nodes as IntUnaware, IntAware or IntAwareOpt nodes, respectively. Moreover, when performance is not being considered, both IntAware and IntAwareOpt nodes can be simply identified as aware nodes, and the IntUnaware node can be referenced as the unaware node. Although the database states of both unaware and aware nodes can be expected to diverge, no interference appears between the three types of protocol, as a certification-based protocol takes its decisions independently from the rest of nodes. So it is possible to run these three protocols together in the same cluster. To determine the outcome of transactions in each protocol type, each of the 22,500 transactions that we issued in each execution of the test, was assigned a unique global identifier determined by the total order broadcast.

With this cluster configuration, it is easy to detect the incorrect decisions made by the IntUnaware protocol. Suppose that an IntV transaction T_v is delivered in the system, presenting no conflicts with concurrent transactions. Aware nodes try to commit it and discard it when the database notifies the integrity violation, removing T_v from the history list. The unaware node, after the certification step, includes T_v in both the history and the `tocommit` lists. Later, when T_v is sent to the database, the integrity violation is detected, so T_v is not indefinitely retried, but –and this is the error in the integrity management– it is not deleted from the history list. Problems arise when subsequent IntS transactions present write conflicts only with those IntV transactions not applied but maintained in the history list of the unaware node. The outcome for these transactions will differ from unaware to aware nodes: they will be incorrectly aborted in the unaware node, while committed in aware ones. For convenience, let us call such incorrect abortions

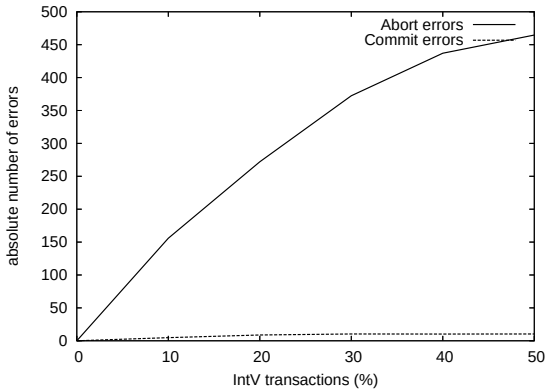


Figure 2: Errors of the IntUnaware protocol.

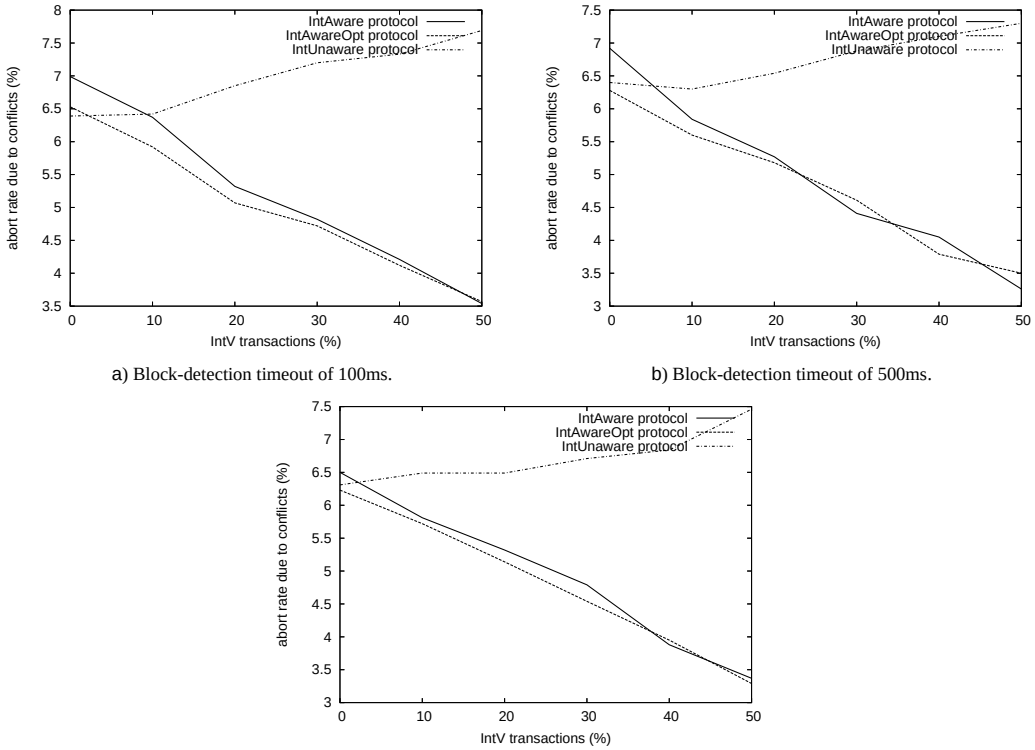
abort errors. But such erroneous abortions are not the only problem caused by the improper management of integrity. Notice that a transaction T_a, incorrectly aborted in the unaware node, is committed in aware ones. This way, it appears in the history list of aware nodes but not in the history list of the unaware one. Now suppose that a subsequent IntS transaction T_c is delivered in the nodes. If T_c only presents conflicts with T_a, T_c will be aborted in aware nodes but committed in the unaware one. This is called a commit error. Both abort and commit errors were computed in the tests and are shown in figure 2 in absolute numbers, i.e., the number of transactions with different outcome over the total of transactions issued. Notice that only IntS transactions are subjected to such errors in their outcome, as IntV transactions always end in abortion.

Notice also that transactions incorrectly included in the history list –IntV or erroneously committed IntS transactions– and transactions incorrectly missed from it –erroneously aborted IntS transactions– affect subsequent certifications. This way, it is possible that the unaware node certifies incoming transactions in a wrong way, i.e. with different certification result that aware nodes, or, even, the same final certification decision but based on conflicts with different transactions. These certification errors remain unnoticed in our tests except for those related to an IntS transaction that becomes certified with a different result in both types of nodes.

Mainly, as seen in figure 2, detected errors consist in abort errors, i.e. aborting transactions that conflict with others incorrectly included in the history list. Commit errors are less usual as transactions in an unaware node are certified against a greater number of transactions, thus being more likely to get aborted by mistake. Notice that, for an IntS transaction to get erroneously committed, it can only present conflicts with other IntS transactions that were erroneously aborted. The graph shows that, as expected, the greater the percentage of IntV transactions, the greater number of abort errors made by the unaware node, whereas the number of commit errors never rises above 20.

The next graphs, in figure 3, show the average percentage of local transactions that got aborted in a node due to write conflicts with concurrent transactions previuosly delivered. Note that here we do not consider those transactions aborted by integrity violation. It can be seen that this abortion rate increases in IntUnaware nodes as we increase the percentage of IntV transactions, while it linearly decreases in both IntAware and IntAwareOpt nodes. Here it has to be remarked that this abortion rate was calculated in each node over its own local transactions. This way, although IntUnaware nodes incorrectly abort transactions from all nodes, transactions aborted by a node only contribute to the raise of the abortion rate of that node if they were local to it. Thus, in IntAware and IntAwareOpt nodes, as we increase the IntV percentage, more and more transactions are removed from the history list due to integrity violation, leading to a smaller probability for local IntS transactions to present conflicts with the remaining ones. Similar graphs are obtained with different timeout values for the block detection mechanism.

The last graphs, in figure 4, show the performance of each type of node, measured as the length in milliseconds of local transactions. As mentioned above, completion time for aborted transactions, figure 4b, in IntAware nodes is clearly greater than in IntUnaware ones. This led us to add the IntAwareOpt protocol



a) Block-detection timeout of 100ms.

b) Block-detection timeout of 500ms.

c) Block-detection timeout of 750ms.

Figure 3: Abortion rate due to actual certification conflicts.

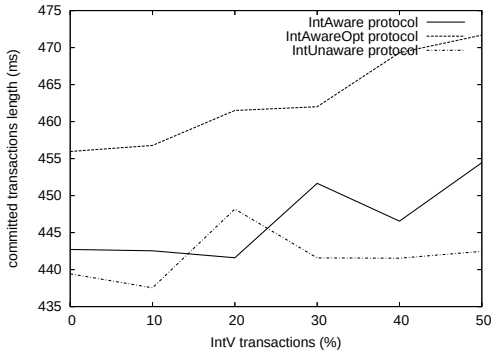
version, which achieves reductions in the completion time for aborted transactions from at least 4.42% –timeout of 500 ms and 10% of IntV transactions– up to 11.35% –timeout of 750 ms and 40% of IntV transactions–. As said before, as we increase the BlockDetector timeout, the length of aborted transactions becomes greater in those nodes where certification is delayed in exchange for correctness, i.e., in aware nodes. In the case of IntUnaware nodes, as they immediately (but possibly incorrectly) certify transactions, variations in the timeout have negligible effects.

With regard to committed transactions, figure 4a shows that IntAware nodes again perform worse than IntUnaware ones. Recall that the proper management of integrity constraints prevents the IntAware protocol from applying any optimisation proposed for certification-based replication protocols –actually, the IntAwareOpt version only achieves to half-apply one of them–. Even so, increments in completion time for committed transactions only reach up to 14.27%, but it has to be noticed that the IntUnaware protocol only applies the optimisation consisting in certifying newly delivered writesets concurrently with the application of previous ones. Thus, this difference would be bigger when comparing with an optimised version of the SI CBR protocol type. Graphs also show that the IntAwareOpt version performs worse than the IntAware one when considering committed transactions. The management of dependencies between transactions and the inclusion of a new thread to establish and control them, increases the completion time of committed transactions. Nevertheless, this increment never exceeds 5.8% of the IntAware completion time, whereas the improvement in time for aborted transactions reaches up to 11.35%.

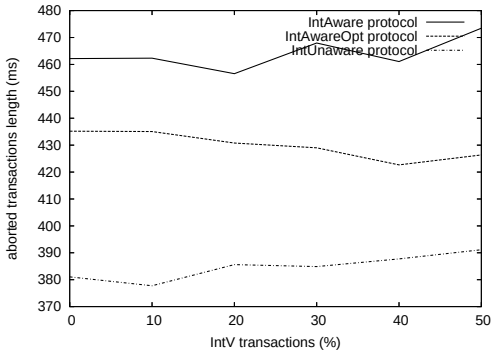
6 Related Work

This work is a continuation from our technical report [16]. As far as the authors know, these two papers are the first ones addressing the problem of integrity constraints management at protocol level of database replication systems.

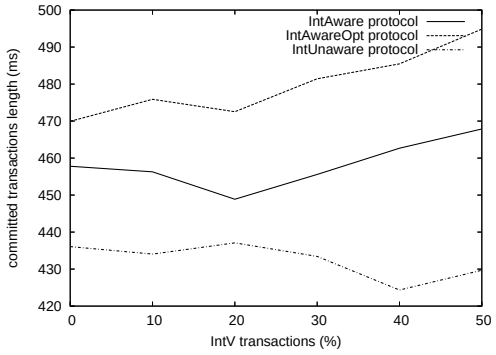
As seen in the previous section, the improper management of integrity-violating transactions leads



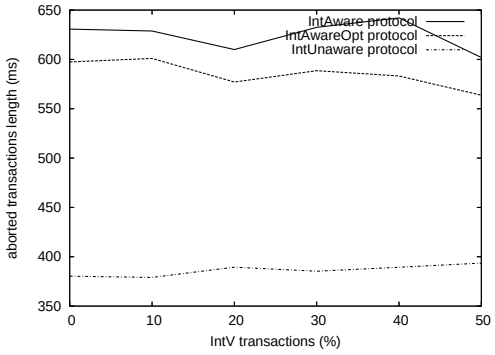
i) Block-detection timeout of 100ms.



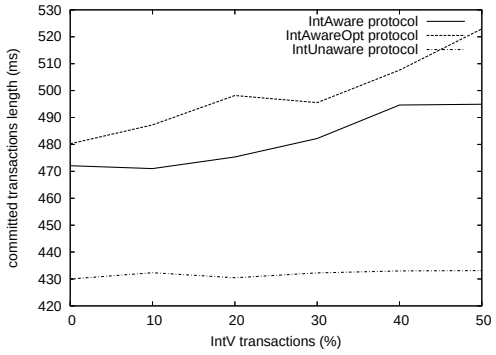
i) Block-detection timeout of 100ms.



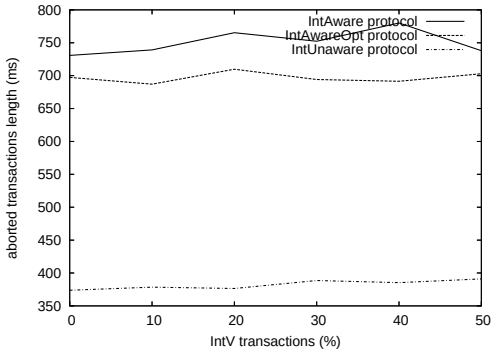
ii) Block-detection timeout of 500ms.



ii) Block-detection timeout of 500ms.



iii) Block-detection timeout of 750ms.



iii) Block-detection timeout of 750ms.

a) Committed transactions.

b) Transactions aborted due to conflicts.

Figure 4: Length of transactions.

to a higher abortion rate. This effect will be increased when exploiting an optimisation proposed for certification-based protocols [7], consisting in grouping multiple successfully certified writesets, applying all of them at once in the underlying DBMS. This reduces the number of DBMS and I/O requests, thus improving a lot the overall system performance. On the other hand, it will also generate the abortion of all transactions in a batch as soon as one of them raises an integrity constraint violation.

We have also remarked that indefinitely reattempting to commit an integrity-violating transaction –or batch of grouped transactions– is not only useless –as the database will always raise the integrity exception– but also prevents the protocol from proceeding normally, stopping the processing of all transactions in the system. This cannot be avoided even though some other optimisations are used, such as the holes technique presented in [14]. With such optimisation, several non-conflicting transactions can be sent to the database, in such a way that the commit order can be altered from one node to the others if a transaction commits before a previously delivered one. This way, when indefinitely reattempting one integrity-violating transaction, subsequent non-conflicting transactions can be sent to the database, not stopping the processing of the node. However, the holes optimisation cannot be applied when the next transaction presents conflicts with the ones already sent to the database, so the protocol will stop all processing eventually.

7 Conclusions

The literature on integrity checking in replicated database systems is extremely scant; solitary exceptions are few and peripheral, e.g., [22, 18]. None of the papers we have found deals with the problem of coordinating integrity checking with replication protocols. However, on the protocol level of replicated database architectures, many problems remain to be solved for implementing mechanisms that take care of controlling transaction consistency, replication consistency and integrity, i.e., semantic consistency. One of them is addressed in this paper.

Due to the physical distribution of database replicas over possibly wide areas, and to the communication between replicas needed to coordinate their actions, there is a latency between the point of time a transaction is requested to commit and the point of time it is effectively committed. For guaranteeing the ACID property of transactions [3], integrity can often not be checked soundly in immediate mode, but has to be delayed until all write actions of previously delivered transactions have been processed. For several classes of replication protocols, this poses a problem, because none of the known ones consider integrity constraints at all. Rather, they sanction transaction as ready to commit if no access conflict to shared data resources has been detected. That way, integrity checking may fail due to other writes from transactions delivered during the mentioned latency gap. Thus, the right moment of reacting suitably to integrity violations may be missed, so that committed transactions either are aborted behind schedule, or integrity remains persistently violated. Both of that is known to have potentially fatal consequences for consistency.

We have presented an experimental study of the negative effects of not correctly managing integrity constraints. This has been accomplished by comparing the behavior of two protocols. One of them reflected the traditional behavior of protocols which do not care about integrity maintenance, based on the uncautious assumption that all transactions are programmed in such a way that they will preserve integrity. As opposed to that, the other protocol studied in our analysis properly handles semantic consistency as declared by integrity constraints. Only a simple modification of the traditional protocol is enough to reach the correctness of the second protocol. We have showed that an improper processing of integrity-violating transactions entails a history list that does not correspond to the transactions actually applied in the database, which leads to errors in the certification phase of subsequent transactions. This causes not only incorrect abortions but also incorrect commits, as explained in this paper.

Moreover, and resulting from the errors mentioned above, incorrect nodes present higher conflict-related abortion rates –computed in each node as the percentage of local transactions that were aborted by the local protocol due to write conflicts with concurrent transactions–.

Finally, results also show that the proposed integrity-aware protocol introduces higher delays due to the larger extension of the mutually-exclusive zone needed to safely access the history list. These delays arrive up to a 14.27% of the completion time, and an optimised version of the integrity-aware protocol achieves to reduce the completion time only for aborted transactions, showing that more researching has to be done in order to obtain efficient constraint-aware replication protocols.

References

[1] 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.

[2] Josep M. Bernabé-Gisbert, Raúl Salinas-Monteagudo, Luis Irún-Briz, and Francesc D. Muñoz-Escoí. Managing multiple isolation levels in middleware database replication protocols. *Lecture Notes in Computer Science*, 4330:511–523, December 2006.

[3] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, MA, USA, 1987.

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

[5] Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[6] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database replication using generalized snapshot isolation. In *SRDS*, pages 73–84, Orlando, FL, USA, October 2005.

[7] Sameh Elnikety, Steven G. Dropsho, and Fernando Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *EuroSys*, pages 117–130, Leuven, Belgium, April 2006.

[8] Alan Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[9] J. Gray. Notes on database operating systems. In R. Bayer, R. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*. Springer-Verlag, 1979.

[10] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *SIGMOD Conference*, pages 173–182, 1996.

[11] 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.

[12] Luis Irún-Briz, Hendrik Decker, Rubén de Juan-Marín, Francisco Castro-Company, José Enrique Armendáriz-Iñigo, and Francesc D. Muñoz-Escoí. MADIS: A slim middleware for database replication. *Lecture Notes in Computer Science*, 3648:349–359, August 2005.

[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 *SIGMOD Conference*, pages 419–430, 2005.

[15] Davide Martinenghi, Henning Christiansen, and Hendrik Decker. Integrity checking and maintenance in relational and deductive databases and beyond. In Zongmin Ma, editor, *Intelligent Databases: Technologies and Applications*, pages 238–285. Idea Group Publishing, 2006.

[16] Francesc D. Muñoz-Escoí, Hendrik Decker, José Enrique Armendáriz-Íñigo, and José Ramón González de Mendívil. On supporting integrity constraints in relational database replication protocols. Technical Report ITI-ITE-08/05, Instituto Tecnológico de Informática, Valencia, Spain, March 2008.

[17] Francesc D. Muñoz-Escoí, Jerónimo Pla-Civera, María Idoia Ruiz-Fuertes, Luis Irún-Briz, Hendrik Decker, José Enrique Armendáriz-Iñigo, and José Ramón González de Mendívil. Managing transaction conflicts in middleware-based database replication architectures. In *Symposium on Reliable Distributed Systems*, pages 401–410, 2006.

[18] Michael Okun and Amnon Barak. Atomic writes for data integrity and consistency in shared storage devices for clusters. *Future Generation Comp. Syst.*, 20(4):539–547, 2004.

[19] PostgreSQL. Web site. Accessible in URL: http://www.postgresql.org, 2008.

[20] Raúl Salinas-Monteagudo, Josep M. Bernabé-Gisbert, Francesc D. Muñoz-Escoí, J. Enrique Armendáriz-Íñigo, and J. R. González de Mendívil. SIRC: A multiple isolation level protocol for middleware-based data replication. In *Intnl. Symp. on Comp. Inform. Sciences*, Ankara, Turkey, November 2007.

[21] Spread. The Spread communication toolkit. Accessible in URL: http://www.spread.org, 2008.

[22] Luís Veiga and Paulo Ferreira. RepWeb: Replicated web with referential integrity. In *SAC*, pages 1206–1211, Melbourne, FL, USA, March 2003.

[23] Matthias Wiesmann and André Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Trans. on Knowledge and Data Engineering*, 17(4):551–566, April 2005.