Integrity Constraint Management in Certification-Based Replication Protocols*

M. I. Ruiz-Fuertes, F. D. Muñoz-Escoí, H. Decker

Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia, Spain
{miruifue, fmunyoz, hendrik}@iti.upv.es

Abstract. Current database replication protocols take care of read-write and/or write-write conflict evaluation. If there are no such conflicts, protocols send transactions to the database and assume they will eventually commit. But commitment may fail due to integrity constraints violations. Also, new conflicts may arise as a result of the integrity checking process, as new objects may be accessed during this phase. 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. Certification-based replication protocols can be adapted to correctly deal with the integrity support of the underlying DBMS. In this paper, we analyse the support for integrity that can be provided in such replication protocol family. Also, we experimentally demonstrate the negative effects that an incorrect management of integrity constraints causes in a database replication distributed system.

1 Introduction

Many database replication protocols have been proposed [1,2,3,4,5] over the years. None of these proposals has assessed the support of semantic consistency as required by integrity constraints. This is in line with the well-known result that the serializability of integrity-preserving transactions yields integrity-preserving schedules [1], since all protocols assumed a serializable isolation level.

Unfortunately, this is not always true for replicated databases. Concurrent transactions may start and execute in different nodes, and proceed without problems until they request commitment. Then, the replication protocol is in the position to validate each transaction and request its commit, based on checking for read-write and write-write conflicts among concurrent transactions. However, at least in middleware-oriented protocols [6,7,8,9], 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. In this case, integrity

* This work has been partially supported by EU FEDER and the Spanish MEC under grants TIN2006-14738-C02-01 and BES-2007-17362.

checking may diagnose constraint violations on transactions already successfully validated by the protocol. Moreover, the integrity checking process may increase the readset/writeset of the transaction, remaining these new accesses unnoticed by the protocol.

On the other hand, if all checkings are made immediately after each update and the replication protocol guarantees a serializable isolation level, all the accesses made during integrity checking will be contained in the readset/writeset of the transaction, thus preventing other concurrent transactions to access the same items, even though no integrity constraint violation would have appeared with the commitment of both transactions. However, a deferred checking still presents the problems mentioned above.

Things may get even worse when isolation levels are relaxed. Recently, the *snapshot* isolation level [10] 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 [11], the replication protocols that support it [4,12,13,9] 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 [14,15], are facing the same problem as discussed above.

This way, if the replication protocol only guarantees a snapshot isolation level –or SI–, only write-write conflicts will be checked during the validation phase. In this scenario, integrity checks must be deferred in order to ensure a final integrity consistent state of the database. 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 being executed in different nodes: transaction A only updates column x, so its integrity checking process reads column y to verify integrity; on the other node, transaction B only updates column y, so its integrity checking phase reads column x. Suppose that both checkings are passed in their local nodes, but the update made by each transaction is such that, when combined, the integrity constraint is violated. Later, during validation, the protocol will not detect any write-write conflict between them and thus it will allow both transactions to commit, leading to an integrity inconsistent state in the database. For this reason, such constraint must be checked at commit time, in order to consider the updates made by previous remote transactions. Something similar will happen with foreign key constraints, for example, which are supported by all DBMSs supporting SI.

Thus, there may be transactions that are sanctioned to commit by the replication protocol but are aborted later, due to deferred integrity violation. 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 [5]. Among them, certification-based replication protocols provide good performance and have optimised algorithms, such as [12]. In this paper, we are going to analyse the integrity-checking support needed for certification-based replication protocols.

This work continues our previous technical report [16]. As far as the authors know, our work is the first in studying the integrity support by replication protocols. If so, then our main contribution consists in inaugurating this subfield of research and in identifying how some replication protocols should be adapted in order to correctly deal with integrity constraints. More specifically, this paper consists in a practical study of the negative effects of not properly 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, some of those incorrectly aborted transactions are prevented to cause the abortion of concurrent conflicting transactions, which are then incorrectly committed.

The rest of this paper is structured as follows. In section 2 is detailed the system model assumed in this work. Section 3 defines integrity constraints and section 4 explains in detail the certification-based replication and the extensions that have to be made to properly manage integrity consistency 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 analises 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 –where clocks are not synchronized but the message transmission time is bounded– composed by N nodes where each one holds a replica of a given database; i.e., the database is fully replicated in all system nodes. Each system node has a local DBMS that is used for locally managing transactions. On top of the DBMS a middleware –called MADIS– is deployed in order to provide support for replication. More information about our MADIS middleware can be found in [7,9]. This middleware also has access to a group communication service (abbreviated as GCS, on the sequel). A GCS provides a communication and a membership service supporting virtual synchrony [17]. The communication service features a total order multicast for message exchange among nodes through reliable channels. The GCS groups messages delivered in views [17]. The uniform reliable multicast facility [18] 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 [19] as our GCS.

3 Integrity Constraints

Integrity constraints define what is a consistent database state, by requiring that certain conditions be invariant across updates. Consistency as defined by integrity constraints is sometimes called *semantic consistency*, in order to emphasise that the constraints express properties that pertain to the domain captured by the database and the application programs that use 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.

Although integrity constraints can be classified under different criteria –they can be declarative or procedural, static or dynamic, column, table or inter-table constraints–, in this paper we only consider the effects of not correctly coordinating the results of such checks with the actions taken by the replication protocol, regardless of the integrity constraint type. The only relevant distinction is the one introduced by SQL-based systems for practical purposes: the *checking mode*. It specifies when a constraint is to be checked: either *immediate* (i.e., directly after each update action) or *deferred*, i.e., delayed until the transaction containing the update action requests checking. This means that the checking mode is controlled by the transaction.

4 Certification-Based Database Replication Protocols

Certification-based database replication protocols –or CBR– use a total order broadcast mechanism [17] for update propagation and replica coordination. As defined in [5], certification-based replication 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 historic 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 a historic 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 T is, for instance, locally involved in a deadlock, being aborted by the DBMS, 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 [10] 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 [15] 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. To this end, 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 used in non-delegate replicas without having previously applied transaction t's updates. If so happens, this means that t's writeset should 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. A writeset should be added to that list in step 7 of this new protocol, once it has been accepted for commitment. Thus, the list might grow indefinitely. To avoid such problem, the list can be pruned following the suggestions given in [5]. 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).

Unfortunately, certification-based replication, despite of providing good performance, 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 previous sequence. However, as we have seen, that may lead to constraint violation and abortion behind schedule. In this case, reattempts to commit the transaction clearly would be in vain. Also note that in step 5 the transaction was assumed successful and other transactions whose data were delivered after T might have already been aborted due to the assumed T's commit. So, certification-based pro-

tocols need to be modified in order to correctly deal with deferrable constraints. We discuss such extensions 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.

a) SI CBR protocol.	b) Extended SI CBR protocol.
1: Execute t.	1: Execute t.
2: On t commit request:	2: On t commit request:
3: TO-bcast(R, <wset(t), r_i>)	3: TO-bcast(R, <wset(t), r_i>)
4: Upon <wset(t), r_d> reception:	4: Upon <wset(t), r_d> 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.

Fig. 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 failed due to integrity constraints (but not if failure is due to other causes, since deadlock-caused abortions should be indefinitely reattempted), then the writeset of such transaction should be removed from the $wslist_d$ variable, since it has not finally been accepted. This is done in lines 11a and 11b.

Again, these seemingly minor extensions may have a notable impact on system performance. Typical SI CBR protocols [13,12,9,20] 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., [12,9]), an auxiliary list is used for storing the writesets to be committed (the related protocol section in fig. 1b 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; 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 the practical analysis, we have implemented two SI CBR protocols: a) IntSimple 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; and b) IntAware protocol, which corresponds to the pseudocode shown in Figure 1b. In short, the integrity management error made by the IntSimple protocol is to keep in the historic list those transactions that were aborted due to integrity violations.

To accomplish the analysis, we use PostgreSQL [21] as the underlying DBMS, and a database with two tables, both of them with 10000 rows and two columns, one of them declared as primary key. The remaining row of table2 is a foreign key that references table1 –the integrity constraint related is set deferred–. The second row of table1 is just an integer field that is updated by transactions.

Two types of transactions have been used in the experiments: a) transactions that do not violate any integrity constraint, called IntS –integrity safe– transactions; and b) transactions that cause an integrity violation as they update the foreign key to a value not contained in the referenced primary key. These last transactions are called IntV –integrity violating– and are introduced in the system in a varying percentage –0, 10, 20, 30, 40 and 50%– in order to accomplish the study. 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 historic list. Thus, all the graphs presented have this percentage in their x axe.

Both protocols have been tested using our MADIS middleware with 4 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 7,500 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, two replica nodes work with the IntAware protocol and the other two use the IntSimple protocol. Although the final database state will differ from IntAware nodes to IntSimple ones –as expected–, no interference appears between the two types of protocol, as a certification-based protocol takes its decisions independently from the rest of nodes. So it is possible to run these two protocols together in the same cluster. This way, and assigning to each transaction in the system a global identifier –determined by the total order broadcast–, we can determine the outcome for each one of the executed transactions –a total of 120,000 in each execution of the test– in each type of protocol.

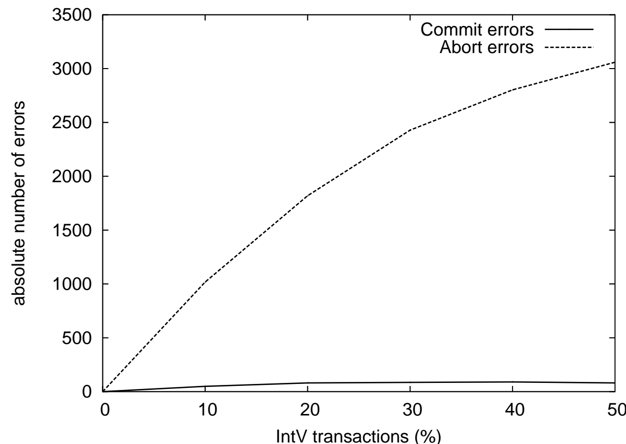

Fig. 2. Errors of the IntSimple protocol.

With this cluster configuration, it is easy to detect the incorrect decisions made by the IntSimple protocol. Suppose that an IntV transaction T_v is delivered in the system, presenting no conflicts with concurrent transactions. IntAware nodes try to commit it and discard it when the database notifies the integrity violation, removing T_v from the historic list. IntSimple nodes, after the certification step, include T_v in both the historic 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 historic list. Problems arise when subsequent IntS transactions present write conflicts only with those IntV transactions not applied but maintained in the historic of IntSimple nodes. The outcome for these transactions will differ from IntAware to IntSimple nodes: they will be incorrectly aborted in IntSimple nodes, while committed in IntAware ones. This is what we have called abort errors. But these abortions are not the only problem caused by the improper

management of integrity. Notice that a transaction T_a, incorrectly aborted in IntSimple nodes, is committed in IntAware ones. This way, it appears in the historic of IntAware nodes but not in the historic of IntSimple ones. 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 IntAware nodes but committed in IntSimple nodes. This is called a commit error. Both abort and commit errors were computed in the tests and are shown in Figure 2. Notice that only IntS transactions can suffer these errors in their outcome, as IntV transactions always end in abortion.

Notice also that transactions incorrectly included in the historic –IntV or IntS erroneously committed– and transactions incorrectly missed from it –IntS erroneously aborted– affect subsequent certifications. This way, it is possible that IntSimple nodes certify incoming transactions in a wrong way, i.e. with different certification result that IntAware 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, detected errors consist in abort errors, i.e. aborting transactions that present conflicts with others incorrectly included in the historic list. Commit errors are less usual as transactions in IntSimple nodes are certified against a greater number of transactions, thus being more likely to get aborted by mistake. Notice that, for an IntS transaction T_s to get erroneously committed, it can only present conflicts with other IntS transactions that were erroneously aborted.

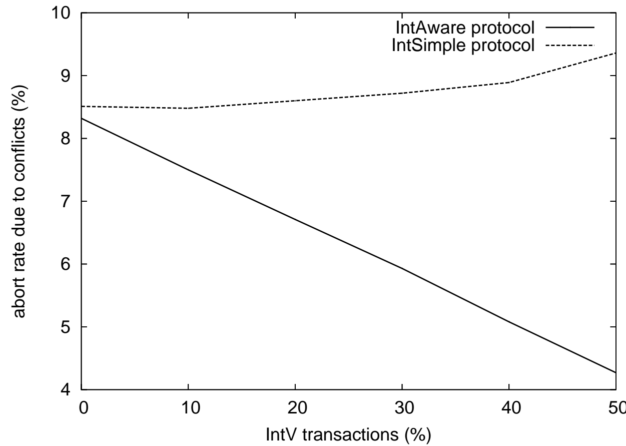

Fig. 3. Abortion rate due to conflicts.

The next graph, Figure 3 shows 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 IntSimple nodes as we increase the percentage of IntV transactions, while it linearly decreases in IntAware nodes. Here it has to be remarked that this abortion rate was calculated in each node over its own local transactions. This way, although IntSimple 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 nodes, as we increase the IntV percentage, more and more transactions are removed from the historic due to integrity violation, leading to a smaller probability for local IntS transactions to present conflicts with the remaining ones.

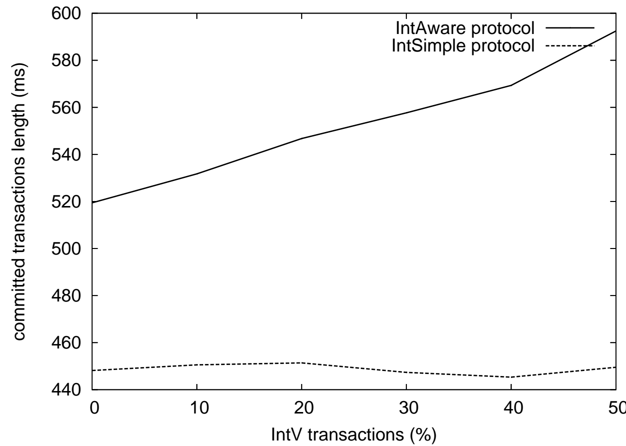

Fig. 4. Length of committed transactions.

The last graph, Figure 4, shows the performance of each type of node, measured as the length in milliseconds of local committed transactions. As seen before, the IntAware protocol is slower than the IntSimple one, as integrity constraint management has prevented it to make any optimisation. Moreover, as the percentage of IntV transactions grows, committed transactions increase their completion time. This can be explained considering that a local transaction T_b is blocked by the underlying DBMS when it tries to update an item that has been previously updated by a concurrent transaction T_c. This block ends with the abortion of T_b if T_c finally commits. On the other hand, if T_c aborts, T_b is allowed to proceed, having its duration increased in the elapsed time. Thus, when the number of IntV transactions –which finally abort– increases, the probability of proceeding after a block also increases, leading to higher completion times for committed transactions.

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 to a higher abortion rate. This effect will be increased when exploiting an optimisation proposed for certification-based protocols [20], 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 hole technique presented in [12]. 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., [22,23]. 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 [1], 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, only one of them properly handling the semantic consistency, demonstrating the consequences suffered. We have showed that an improper processing of integrity-violating transactions entails an historic 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 historic list. These delays arrive up to a 25% of the completion time, showing that more researching has to be done in order to obtain efficient constraint-aware replication protocols.

References

1. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison Wesley, Reading, MA, USA (1987)
2. Carey, M.J., Livny, M.: Conflict detection tradeoffs for replicated data. ACM Trans. on Database Systems **16**(4) (1991) 703–746
3. Gray, J., Helland, P., O'Neil, P.E., Shasha, D.: The dangers of replication and a solution. In: SIGMOD Conference. (1996) 173–182
4. Kemme, B., Alonso, G.: A new approach to developing and implementing eager database replication protocols. ACM Transactions on Database Systems **25**(3) (September 2000) 333–379
5. Wiesmann, M., Schiper, A.: Comparison of database replication techniques based on total order broadcast. IEEE Trans. on Knowledge and Data Engineering **17**(4) (April 2005) 551–566
6. Esparza-Peidro, J., Calero-Monteagudo, A., Bataller, J., Muñoz-Escoí, F.D., Decker, H., Bernabéu-Aubán, J.M.: COPLA - a middleware for distributed databases. In: APLAS, Shanghai, China (December 2002) 102–113
7. Irún-Briz, L., Decker, H., de Juan-Marín, R., Castro-Company, F., Armendáriz-Iñigo, J.E., Muñoz-Escoí, F.D.: MADIS: A slim middleware for database replication. Lecture Notes in Computer Science **3648** (August 2005) 349–359

8. Patiño-Martínez, M., Jiménez-Peris, R., Kemme, B., Alonso, G.: MIDDLE-R: Consistent database replication at the middleware level. ACM Trans. Comput. Syst. **23**(4) (2005) 375–423
9. Muñoz-Escoí, F.D., Pla-Civera, J., Ruiz-Fuertes, M.I., Irún-Briz, L., Decker, H., Armendáriz-Iñigo, J.E., González de Mendívil, J.R.: Managing transaction conflicts in middleware-based database replication architectures. In: Symposium on Reliable Distributed Systems. (2006) 401–410
10. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. In: Proc. of the ACM SIGMOD International Conference on Management of Data, San José, CA, USA (May 1995) 1–10
11. Fekete, A., Liarokapis, D., O'Neil, E.J., O'Neil, P.E., Shasha, D.: Making snapshot isolation serializable. ACM Trans. Database Syst. **30**(2) (2005) 492–528
12. Lin, Y., Kemme, B., Patiño-Martínez, M., Jiménez-Peris, R.: Middleware based data replication providing snapshot isolation. In: SIGMOD Conference. (2005) 419–430
13. Elnikety, S., Zwaenepoel, W., Pedone, F.: Database replication using generalized snapshot isolation. In: SRDS, Orlando, FL, USA (October 2005) 73–84
14. Bernabé-Gisbert, J.M., Salinas-Monteagudo, R., Irún-Briz, L., Muñoz-Escoí, F.D.: Managing multiple isolation levels in middleware database replication protocols. Lecture Notes in Computer Science **4330** (December 2006) 511–523
15. Salinas-Monteagudo, R., Bernabé-Gisbert, J.M., Muñoz-Escoí, F.D., Armendáriz-Íñigo, J.E., González de Mendívil, J.R.: SIRC: A multiple isolation level protocol for middleware-based data replication. In: Intnl. Symp. on Comp. Inform. Sciences, Ankara, Turkey (November 2007)
16. Muñoz-Escoí, F.D., Decker, H., Armendáriz-Íñigo, J.E., González de Mendívil, J.R.: On supporting integrity constraints in relational database replication protocols. Technical Report ITI-ITE-08/05, Instituto Tecnológico de Informática, Valencia, Spain (March 2008)
17. Chockler, G., Keidar, I., Vitenberg, R.: Group communication specifications: A comprehensive study. ACM Comput. Surv. **33**(4) (2001) 427–469
18. Hadzilacos, V., Toueg, S.: Fault-tolerant broadcasts and related problems. In Mullender, S., ed.: Distributed Systems. 2nd edn. ACM Press (1993) 97–145
19. Spread: The Spread communication toolkit. Accessible in URL: http://www.spread.org (2008)
20. Elnikety, S., Dropsho, S.G., Pedone, F.: Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In: EuroSys, Leuven, Belgium (April 2006) 117–130
21. PostgreSQL: Web site. Accessible in URL: http://www.postgresql.org (2008)
22. Veiga, L., Ferreira, P.: RepWeb: Replicated web with referential integrity. In: SAC, Melbourne, FL, USA (March 2003) 1206–1211
23. Okun, M., Barak, A.: Atomic writes for data integrity and consistency in shared storage devices for clusters. Future Generation Comp. Syst. **20**(4) (2004) 539–547