Extending Middleware Protocols for Database Replication with Integrity Support

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

Instituto Tecnológico de Informática - Depto. de Ing. Matemática e Informática

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

Technical Report TR-ITI-ITE-08/14

Extending Middleware Protocols for Database Replication with Integrity Support

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

Instituto Tecnológico de Informática - Depto. de Ing. Matemática e Informática

Technical Report TR-ITI-ITE-08/14

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

Jul, 2008

Abstract

Current middleware database replication protocols take care of read-write conflict evaluation. If there are no such conflicts, protocols sanction transactions to commit. Other conflicts may arise due to integrity violation. So, if, in addition to the consistency of transactions and replicas, also the consistency of integrity constraints is to be supported, some more care must be taken. Some classes of replication protocols are able to seamlessly deal with the integrity support of the underlying DBMS, but others are not. In this paper, we analyze the support for integrity that can be provided in various classes of replication protocols. Also, we propose extensions for those that cannot directly manage certain kinds of constraints that are usually supported in DBMSs.

1 Introduction

Over the years, many database replication protocols have been proposed [3, 5, 11, 14]. 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 [3], since all protocols assumed a serializable isolation level.

Unfortunately, this result does not hold in a replicated database scenario if transactions trust that the enforcement of all constraints declared in the schema is delegated automatically to the DBMS. Concurrent transactions may start and execute in different nodes, and proceed as usual until they request commitment. At this point, 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 [9, 12, 21, 19], complications may arise, for instance if constraints are checked in deferred mode, i.e., at effective commit time, i.e., only after conflict validation by the replication protocol. Recall that deferred checking is unavoidable, sometimes, for maintaining transaction atomicity. Thus, integrity checking may diagnose constraint violations that remained unnoticed by the replication protocol. For instance, this may happen if integrity checking results in a failed read request of some item that otherwise would not be accessed, e.g., one referenced by a foreign key constraint. Thus, there may be transactions that are sanctioned to commit by the replication protocol but are aborted later, due to deferred integrity checking. Recall that to abort after commit is a cardinal sin against the principles of transaction processing.

Things may get even worse when isolation levels are relaxed. Recently, the *snapshot* isolation level [2] 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 [10], the replication protocols that support it [14, 16, 7, 19] 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.

Most modern database replication protocols use total order broadcast for propagating sentences or writesets/readsets of transactions to other replicas. We are going to analyze the integrity-checking support needed for different categories of replication protocols, according to the classification proposed in [26]. As far as the authors know, no such study of integrity support by replication protocols exists yet. If so, then our main contribution consists in inaugurating this subfield of research by the study at hand, and in identifying how some replication protocols should be adapted in order to correctly deal with integrity constraints.

Section 2 recapitulates the main database replication classes identified in [26]. Section 3 describes the problems that arise in constraint management in a replicated system. In Section 4, we propose solutions for the problems identified in Sect. 3. Later, Section 5 describes an analytical model that has been used for studying the costs of a correct constraint management in middleware protocols. Finally, in Section 6, we conclude with a long-term outlook.

2 Database Replication Protocols

Replication techniques were initially applied on distributed databases extending the distributed mechanisms already used in the latter; i.e., *2 Phase Commit* (2PC) for termination management and *Strict Two Phase Locking* (implemented by distributed locks) or *Multi-Version* management for concurrency control [3]. However, this approach prevented such first generation of replicated databases from achieving good performance. As a result, new proposals arose in the next decade. Thus, new techniques were proposed for concurrency control, trying to ensure that most of the requests were locally managed, as in the *Optimistic Two Phase Locking* technique described in [5] that only requests remote locks at commit time and whose performance and abort rates are better than any other of its contemporary approaches in most settings.

Despite this, termination management was also a problem, and some other papers [1, 22] proposed and proved that *atomic* (i.e., total-order) *broadcast* [6] can be an efficient replacement for the traditional 2PC protocol. Moreover, such two papers also maintained the suggestions given in [5] for improving concurrency control performance. As a result, a new family of database replication protocols was generated, based on first executing locally all kinds of transactions and broadcasting at commit time –and in total order– the updates made –if any– to all replicas, still following the ROWA principle presented in [3]. Note that the initial local transaction execution uses a quite optimistic concurrency control management, since evaluation of transaction conflicts is partially delayed until update-propagation time. On the other hand, at remote transaction's application time, local conflicting transactions are aborted in order to guarantee that such remote updates could be applied following the update total order imposed by the atomic broadcast mechanism. This guarantees an excellent performance and reduces the degree of optimism of the resulting concurrency control.

A comparison of these total-order based replication protocol families with other approaches typical in other replicated applications (that follow either the active or primary-copy replication techniques) was described in [26]. We follow such comparison in this section. Its protocol classes are:

- *Active replication* (AR). The client sends its transaction to a given replica R_d. The latter forwards all transaction operations to all database replicas by a single total-order broadcast. Each replica independently and deterministically processes the transaction. Total-order propagation ensures the same transaction scheduling in all replicas. So, all replicas should get the same results for each transaction. Once the transaction has been completed, R_d returns its results to the client.

- *Certification-based replication* (CBR). The transaction is initially executed in a single delegate replica. Once the client application requests its commitment, the transaction readset and writeset are collected and propagated to all replicas using total-order broadcast. Once such message is delivered, the transaction is certified against concurrent transactions whose messages where previously delivered. This validation stage is symmetrical since all replicas can hold the same historic list of previously delivered readsets and writesets. So, all of them can take the same decision on the transaction termination, i.e., no additional communication step is needed for that. Once certified, accepted

transactions are applied and committed, and their writesets and readsets will be temporarily held in order to certify other concurrent transactions whose data will be delivered later. Otherwise, i.e., when the transaction has been aborted in the certification stage, its readset and writeset are discarded.

- *Weak voting replication* (WVR). As in the previous class, each transaction is executed in a single delegate replica. When a transaction requests its commit, its readset and writeset are collected as in the previous class, but only its writeset is broadcast in total order to all replicas. Once the writeset is delivered, it is validated (but only in its own delegate replica) against the writesets of concurrent transactions that have been delivered before its own one. Note that read-write conflicts are detectable, since the delegate replica knows which has been the readset of its local transaction and also which writesets are from remote. If any conflict is found, the transaction is aborted, else it is committed. The decision about this is propagated using a *reliable broadcast* [6] to all replicas. The latter then behave according to the action prescribed in the broadcast message.

- *Primary copy replication* (PCR). All transactions must be executed by the same primary replica which may rely on its local concurrency and integrity control mechanisms to decide whether transactions can be committed or not. Once a transaction has been executed in the primary replica, its writeset will be propagated to other replicas (that behave as backups) and applied there. Finally, the transaction is committed in all replicas, and the client gets reported on this.

- *Lazy replication* (LR). This class is an extension of the previous one, where each transaction may select a different delegate replica and the client is informed after the delegate's commit. Thus, transactions are first completed in their selected delegate replicas and, once done, their updates are propagated to other replicas. This replication class is not practical, since conflicts are actually detected once transactions have been already committed, which demands reconciliation techniques for ensuring consistency. Therefore, we do not further discuss this kind of replication.

Classes AR and PCR do not pose any problem for integrity management. In AR, all replicas should behave in exactly the same way, thus local management is enough. Also in PCR, no coordination nor validation is required since only one replica is processing the transaction. So, in this regard, it is equivalent to a non-replicated architecture. Hence, both AR and PCR protocols are able to deal with integrity constraints in a seamless way, by relying on the support provided by the underlying DBMS.

Unfortunately, neither AR nor PCR protocols provide good performance, according to [26]. However, the authors of [13] propose some adaptations of the PCR class that partition the database and distribute the subsets according to predefined conflict classes among multiple primary copies (each one being responsible for each subset). These adaptations significantly improve the performance, while the reliance on the integrity support from the underlying DBMS is not impaired.

On the other hand, the two classes providing best performance (CBR and WVR) give rise to several problems with regard to integrity constraints. Let us discuss them in detail.

3 Integrity Problems with Deferred Checking

The study of this paper is limited to the SQL constructs that typically are offered and supported for specifying and maintaining semantic consistency. An analysis of integrity support for the SQL:1999 standard in relational DBMSs was given in [24]. Not too much has changed in the state of the art of integrity checking in SQL-based databases in the years after the appearance of the standard, except that various developments have rather diverged from than converged to it, and open source databases have come to be more on par with what has been offered by commercial vendors earlier, in terms of integrity support.

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 emphasize that 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.

In a centralized setting, integrity constraints have been traditionally managed using a deferred or delayed checking [15, 4, 17], instead of an immediate one. This can be explained both in terms of performance improvements (as reported in [15, 17]) and because such checking admits temporary inconsistencies that can be solved before transactions end [15], avoiding intermediate checkings that could have uselessly required time and resources.

Note that the semantics of *immediate checking* are unclear, as already explained in [17]. In order to preserve serializability, each transaction should use the appropriate long read and long write locks on all items it has read or updated. When another concurrent transaction immediately checks some constraint on any of its updates, which kind of isolation level should be used in such check? If serializable is used, none of the concurrent accesses will be observable –or they may block such checking–; if other levels are used, as either *read committed* or *snapshot*, as analyzed in [17], the serializable guarantees might be broken. As a result, deferred checking seems to be the safest approach.

At a glance, deferred checking should be used in middleware database replication protocols, but some problems arise in the CBR and WVR protocol families described above. Both protocol classes need at least two steps for transaction management. A first step devoted to transaction conflict checking and, if such check succeeds, a second step where remote updates are applied in non-delegate replicas and transactions are finally committed. If constraint checking is deferred till transaction commit time, some misbehavior could be generated: the protocol could have admitted a transaction as correct in such first transaction conflict check –reporting its success to the client application– and later the underlying DBMS might abort it when its commit is requested. So, deferred checking was the solution to some problems in a centralized system, but it is now the origin of other ones in a replicated scenario.

Let us present a first example with two concurrent transactions that is correctly managed with deferred checking in a centralized system but cannot be with immediate checking, illustrating what has been said before. Thus, let us assume a DB composed by two tables T1 and T2 and with two integrity constraints:

IC1: A primary key constraint in T1.

IC2: A foreign key constraint in T2, referring to the primary key of T1.

Imagine two concurrent transactions X1 and X2 that among other actions, perform the following operations:

X1: It inserts a tuple t2 in T2, referring to tuple t1 in T1.

X2: It inserts a tuple t3 in T1, with the same value in its primary key than tuple t1, and finally it removes tuple t1 from T1.

This kind of transaction could be needed for, e.g., removing a department (t1, in the example) in a given company and re-assigning all its employees (tuples in T2 referring to t1) to another one (t3, in this example).

Note that these transactions can proceed in a centralized system only with deferred checking –otherwise, constraint IC1 would have been violated, but if X2 executes its operations in the opposite order, constraint IC2 will be violated, so there is no way of completing X2 actions without any constraint violation using immediate checking–, and this admits both a serialization order X1, X2 and X2, X1. Additionally, no problem arises in a replicated system.

Let us imagine now a second example that shows which kind of problems might arise in a replicated system. In this case, a third transaction X3 is needed. It simply removes t1 from T1 and it is executed concurrently with transaction X1 from the previous example. This second example can be executed without problems in a centralized system if the serialization order X1, X3 is followed. However, execution of X3 will remove the effects of X1 if cascaded deletes are used. But these transactions generate the worst consequences if a replicated system is used. In such a case, and assuming deferred checking, such transactions have the following readsets and writesets:

- X1.rset = $\emptyset$
- X1.wset = $\{t2\}$

- X3.rset = $\emptyset$
- X3.wset = $\{$t1$\}$

As a result of this, no replication protocol will be able to detect any read-write nor write-write conflict between both transactions in the first step presented above (conflict evaluation made by the replication protocol). However, at commit time –and assuming that integrity checking is done at that moment– if X1's updates are delivered after X3's ones, X1 will get aborted due to its violation of IC2. Note that for the replication protocol both transactions have been considered as successful and the client has been reported on their commitment, but X1 is actually aborted when it tries to commit in the underlying DBMS. So, the current structure of middleware database replication protocols is not valid when integrity constraints exist in the DB schema.

4 How To Support Constraints

In order to adequately support constraints in the WVR and CBR protocols, some extensions are needed in both of them. We discuss such extensions in the subsections below. We start with the WVR class since its solution is easier.

Replication protocols are assumed to be implemented in a middleware layer (A solution for DBMS-core replication protocols could be trivially derived from the solutions presented here). The underlying 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.

The following symbols are used below for describing protocols: t is the transaction being processed; R is the set of alive replicas; r is the replica that executes the protocol; r_d is the delegate replica, i.e., the replica where the transaction is locally and directly served; c is the client process; DB is the local DBMS interface accessed by the replication protocol; *wset*(t) is the writeset of t, and *rset*(t) is the readset of t. Note that the $DB.abort(t)$ operation is used in non-delegate replicas of some protocols (concretely, in Figs. 2 to 3) 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.

4.1 Weak Voting Replication Protocols

A WVR protocol consists of the steps in the pseudo-code of Figure 1.a. It is structured by three event-driven blocks. In block I (lines 1-3), t is executed locally; its writeset is broadcast upon the event that t is requested to be committed. Blocks II (lines 4–9) and III (lines 10–13) describe what is executed in all replicas. In the former, action is taken upon arrival of the writeset, and t's status is broadcast after validation. In the latter, action is taken upon arrival of the status message, resulting in commit or abort.

Note that the *validate*( ) function evaluates whether there are read-write or write-write conflicts between t and any other local transaction that has not arrived yet to its corresponding step 6. This implies that such local conflicting transactions have not yet requested their commit, or will be delivered afterward in total order. In any case, when a conflict is found between a local and a remote transaction, the local transaction is immediately aborted. If such transaction's writeset has been already broadcast, its *validate*( ) call will finally return an $abort$ value.

Note also that the *DB.commit*( ) operation results are ignored in the WVR protocol, since such operation was assumed to be always successful. Our first extension consists in considering that *DB.commit*( ) requests can return an $abort$ value reporting a constraint violation, or a *commit* value meaning that the commit was successfully made.

Some more extensions are needed, resulting in the protocol displayed in Figure 1.b. Note that the line numbers from the original version in Figure 1.a remain the same. The extensions simply consist of three added lines between the original 6^{th} and 7^{th} lines, and another one after line 10. Thus, lines 6a-6c check

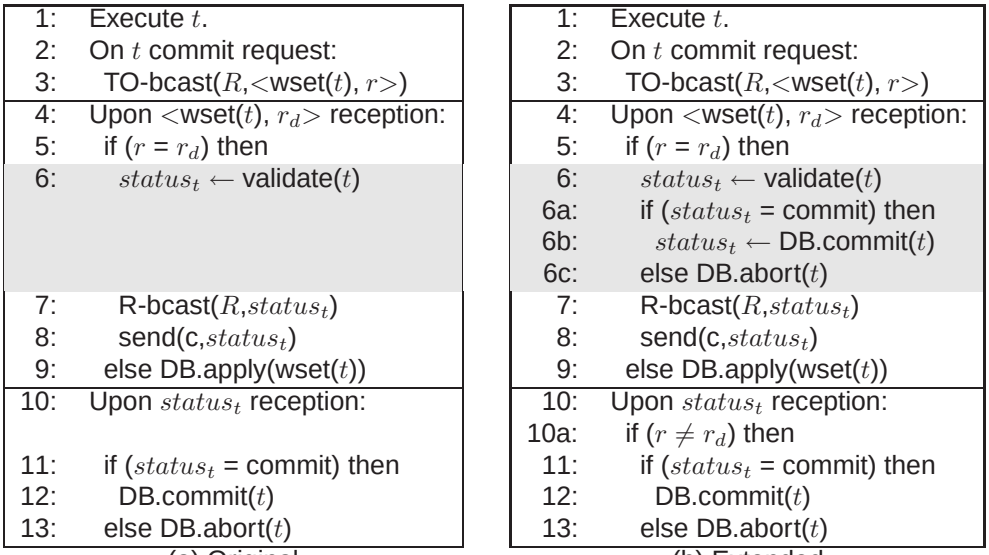

Figure 1: Weak voting protocols.

the result of the validation done by the replication protocol and, if it was successful, lead to the immediate transaction commit in the delegate replica, storing the result value in the $status_t$ variable. Note that in Fig. 1 (and in all subsequent figures), a sequence of rows shaded in gray corresponds to a block of protocol steps that should be executed in mutual exclusion. Note also that a transaction is accepted as aborted only if the abortion was caused by integrity constraint violations. Otherwise, e.g., abortions due to deadlocks or timeouts, the transaction will be indefinitely reattempted, applying its writeset repeatedly. This means that the transaction commitment is attempted before reliably broadcasting the protocol decision. Thus, in case of a commit-time constraint violation, the delegate replica will know about such event, and the replication protocol will be able to react by aborting the transaction in all replicas. This ensures consistency between the decisions taken by the underlying DBMS and the replication protocol. Additionally, line 10a is needed for ignoring in the delegate replica the reception of the message that communicates which has been the termination decision for t, since the actions needed for managing such message have already been executed in the delegate replica.

Although these extensions may seem to demand a minor effort, they do have some impact in transaction response time. In the original algorithm, control was returned to the client before the actual commit was requested to the DBMS, thus reducing the transaction completion time perceived by the user application. When integrity constraints are involved, such optimization is not correct.

Moreover, a potential recovery problem may arise. In case the delegate replica r_d fails (breaks down) for a to-be-committed transaction (t) between steps 6b and 7, the delegate replica had committed the transaction while all others are still waiting for its commit/abort message. This violates the uniformity principle that was required in [22] for preserving in modern replication protocols the same functionality than in the traditional 2PC one. Such uniformity is preserved in the original WVR protocol using uniform total-order broadcast for writeset propagation, and uniform reliable broadcast in the termination/voting final message. In our extensions, one of the non-delegate replicas must be selected as the new transaction delegate, for broadcasting that message. This complicates such transaction termination, demanding much more time. Additionally, this also complicates the recovery protocols since the state maintained in the crashed delegate replica for that transaction is uncertain (it may have applied its updates or not), violating also the assumptions of the *persistent logical synchrony* [18] model that was specifically tailored for replicated database recovery.

4.2 Certification-Based Replication Protocols

CBR protocols are quite similar to WVR ones. The main difference consists in using a symmetrical transaction evaluation stage (in CBR protocols), instead of one where the delegate replica decides, broadcasting later its decision (in WVR protocols). Thus, readset propagation is needed in CBR protocols, since only the delegate replica maintains such information when transactions request their commit, but all replicas need it when the evaluation stage is executed. So, the CBR basic protocol consists of the steps shown in Figure 2.

```
1:  Execute t.
2:  On t commit request:
3:    TO-bcast(R,<wset(t), rset(t), r>)
4:  Upon <wset(t), rset(t), r_d> reception:
5:    status_t ← certify(wset(t),rset(t))
6:    if (status_t = commit) then
7:      if (r_d ≠ r) then
8:        DB.apply(wset(t))
9:      DB.commit(t)
10:   else DB.abort(t)
11:   if (r_d = r) then
12:     send(c,status_t)
```

Figure 2: Certification-based protocol.

Note that readset collection and propagation can be costly if row-level granularity instead of table-level granularity is used by the protocol for managing readsets and writesets. So, in practice, certification-based protocols are rarely used for implementing serializable isolation. On the other hand, CBR serializable protocols are able to deal correctly with integrity constraints, since they rely on local *strict 2PL* for managing write-write conflicts, as shown in the last protocol presented in [1]. This ensures that the replication protocol certification step for a given transaction is not decoupled from such transaction termination. However, this also introduces another performance penalty on supporting the serializable isolation level with this kind of replication protocol.

```
1:  Execute t.
2:  On t commit request:
3:    TO-bcast(R,<wset(t), r>)
4:  Upon <wset(t), r_d> reception:
5:    status_t ← certify(wset(t), wslist)
6:    if (status_t = commit) then
7:      append(wslist,wset(t))
8:      if (r_d ≠ r) then
9:        DB.apply(wset(t))
10:     DB.commit(t)
11:   else DB.abort(t)
12:   if (r_d = r) then
13:     send(c,status_t)
```

(a) Original

```
1:  Execute t.
2:  On t commit request:
3:    TO-bcast(R,<wset(t), r>)
4:  Upon <wset(t), r_d> reception:
5:    status_t ← certify(wset(t), wslist)
6:    if (status_t = commit) then
7:      append(wslist,wset(t))
8:      if (r_d ≠ r) then
9:        DB.apply(wset(t))
10:     status_t ← DB.commit(t)
10a:    if (status_t = abort) then
10b:      remove(wslist,wset(t))
11:   else DB.abort(t)
12:   if (r_d = r) then
13:     send(c,status_t)
```

(b) Extended

Figure 3: SI certification-based protocols.

But CBR is the preferred protocol class when the *snapshot* isolation (abbr., SI) level [2] 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. Note also that several DBMS products (PostgreSQL, Oracle, Microsoft SQL Server,...) are currently able to support such isolation level, but some of them label it as *serializable* and do not enforce a true serializable level but their own flavor of the *SI* one. Our aim in supporting integrity checking in database replication protocols is to guarantee the same supporting level in a middleware protocol than it was present in its underlying DBMS. Regarding these DBMSs, they are actually providing an enhanced integrity management in their systems, equivalent to the one theoretically achievable when a *serializable* level is used. So, the resulting isolation level cannot be tagged as a pure SI one (as it was first defined in [2]) but as an extended SI level able to support integrity constraints in a serializable way. Our aim in this section is to describe how such extended SI level can be implemented in a database replication protocol based on the CBR approach. As a result of this, the extensions shown below in order to correctly manage integrity contraints in this kind of protocols should not be interpreted as a critique on already published SI certification-based protocols (e.g., [14, 16, 7, 19]), since they correctly support such *pure SI* level [2] and they did not aim to extend such level with constraint management as the underlying DBMSs do.

So, we focus on SI-oriented CBR protocols in the rest of this section. To this end, a general protocol of this kind is displayed in Figure 3.a. The symbol $wslist$ is needed for representing the list of successfully certified writesets in replica r. 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 [26].

Again, the extensions for managing integrity constraints in SI CBR protocols, as displayed in Figure 3.b, seem to be minor. Only a slight modification of the original line 10 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, as said before), then the writeset of such transaction should be removed from the $wslist$ variable, since it has not finally been accepted. This is done in lines 10a and 10b.

Again, these seemingly minor extensions may have a notable impact on system performance, this time even more than for extending WVR protocols. Typical SI CBR protocols [7, 16, 19, 8] use some optimizations in order to achieve good performance. One such optimization consists in minimizing 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. The related protocol section in Fig. 3.a only encompasses lines 5 to 7. 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 once line 10b in Fig. 3.b has been executed; 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.

A second optimization [8] consists 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, this optimization might compromise the recovery efforts in case of a system crash, since it is unclear which writesets were actually applied, and some additional actions would be needed to get such information. Also note that this optimization may not work due to reasons explained in the previous paragraph, since each writeset should be individually applied for finding out whether it violates some integrity constraint.

5 Performance Evaluation

In order to study the costs of a correct constraint management in middleware protocols, an analytical model has been used. This model identifies and measures each step followed by transactions in both the original and the extended replication protocols, as they were detailed before.

In the original WVR protocol (Fig. 1.a), the total duration of a transaction perceived by the client can be expressed with the following formula:

$$t_{origWVR} = t_{client} + t_{ws} + t_{TObcast} + [t_{val}] + t_{Rbcast} + t_{resp}$$

where t_{client} represents the interval the client spends accessing the database, t_{ws} is the time needed to collect the writeset of the transaction once the client asks for commitment, $t_{TObcast}$ is the duration of the total order broadcast used to send this writeset to all the system replicas, t_{val} represents the time used to validate the received writeset in the local replica, t_{Rbcast} is the time needed to reliably broadcast the outcome of the previous validation for all nodes to behave equally regarding this transaction termination, and t_{resp} is the time used by the protocol to inform the client about the outcome of this transaction. We represent between brackets the interval of mutual exclusion during which no new writesets can be validated by the protocol. Both t_{client} and t_{ws} depend on the amount of objects accessed by the transaction. Network load[1] will determine the duration for $t_{TObcast}$ and t_{Rbcast}. We can assume that t_{resp} is a constant in replica r for all transactions of the same client, although it can vary from client to client depending on the geographical distance between client and server. Finally, t_{val} depends on the amount of concurrent transactions to be checked against. Note that this formula is valid for both aborted and committed transactions, as the response to the client is sent before the actual termination of the transaction.

When we extend the WVR protocol to correctly manage integrity constraints, the resulting formula is the following:

$$t_{extWVR} = t_{client} + t_{ws} + t_{TObcast} + [t_{val} + t_{term}] + t_{Rbcast} + t_{resp}$$

where t_{term} is the time needed for terminating the transaction, either in abortion or with commitment, depending on the validation outcome. This new component lengthens the total time perceived by the client in two ways: directly, by adding to the formula, and indirectly, by extending the duration of the mutual exclusion zone, making subsequent transactions to wait more before being validated.

Similarly, times for transactions increase in the SI CBR protocol when extended to provide a proper constraint management. In the original SI CBR, the duration of a transaction that results aborted due to conflicts is the following:

$$t_{origCBR_{ab}} = t_{client} + t_{ws} + t_{TObcast} + [t_{cert}] + t_{abort} + t_{resp}$$

where t_{cert} is the time spent during certification, and t_{abort} is the time needed to abort the transaction. On the other hand, a committed transaction presents a total time of:

$$t_{origCBR_{co}} = t_{client} + t_{ws} + t_{TObcast} + [t_{cert} + t_{hist}] + t_{commit} + t_{resp}$$

where t_{hist} is the time required to insert the writeset into the historic list, and t_{commit} is the time used to commit the transaction in the database.

When extending SI CBR protocols to manage contraints, formulas for conflict-aborted transactions, committed ones and those that finally abort due to integrity violations are:

$$t_{extCBR_{ab}} = t_{client} + t_{ws} + t_{TObcast} + [t_{cert}] + t_{abort} + t_{resp}$$
$$t_{extCBR_{co}} = t_{client} + t_{ws} + t_{TObcast} + [t_{cert} + t_{hist} + t_{commit}] + t_{resp}$$
$$t_{extCBR_{in}} = t_{client} + t_{ws} + t_{TObcast} + [t_{cert} + t_{hist} + t_{commit} + t_{hist}] + t_{resp}$$

where $t_{extCBR_{in}}$ is the time for integrity-aborted transactions, for which a second t_{hist} is included in order to represent the time needed to remove the writeset from the historic list.

All these formulas can be simplified making some assumptions. The two initial components of each formula, as said before, depend on the transaction size –the number of accessed objects– and can be considered independent of the protocol version and added in one component t_{local}. Let's also suppose that the network load is such that the costs in the communications can be considered a constant t_{comm}. Finally, we can consider that t_{resp} is negligible or, at least, a constant value only affecting the time of one transaction: it is outside the mutual exclusion zone and is not a broadcast message, so it does not increase the network load (we can add this constant to the t_{local} time). With all this, formulas can be simplified to the following ones:

[1] Note, however, that in most GCSs such broadcasts are asynchronous, demanding a negligible time.

$$
\begin{aligned}
t_{origWVR} &= t_{local} + t_{comm} + [t_{val}] \\
t_{extWVR} &= t_{local} + t_{comm} + [t_{val} + t_{term}] \\
t_{origCBR_{ab}} &= t_{local} + t_{comm} + [t_{cert}] + t_{abort} \\
t_{origCBR_{co}} &= t_{local} + t_{comm} + [t_{cert} + t_{hist}] + t_{commit} \\
t_{extCBR_{ab}} &= t_{local} + t_{comm} + [t_{cert}] + t_{abort} \\
t_{extCBR_{co}} &= t_{local} + t_{comm} + [t_{cert} + t_{hist} + t_{commit}] \\
t_{extCBR_{in}} &= t_{local} + t_{comm} + [t_{cert} + t_{hist} + t_{commit} + t_{hist}]
\end{aligned}
$$

The goal of this analysis is to measure the increase in time for transactions executed by protocols that correctly manage integrity constraints. To this end, the main component to study is that corresponding to the mutual exclusion zone, as the bigger the exclusion zone, the greater the time transactions must wait to be processed, i.e. to enter this mutual exclusion zone. This wait can be represented by a queue in which transactions are inserted when received from the broadcast. The mutual exclusion zone will act as the server in a queueing system modelling the processing of transactions.

We can consider that the arrival of new clients, i.e. new transactions, to the queueing system follows a Poisson distribution with arrival rate of λ. A Poisson distribution is a discrete probability distribution that models the number of events occurring within a given time interval. In our case, it models transactions arriving to the mutual exclusion zone in such a way that the intervals between subsequent arrivals follow an exponential distribution and the average arrival rate is λ. This queueing system has only one server, the mutual exclusion zone, whose service time can be assumed to be exponentially distributed with rate μ. With this M/M/1 queue model, some values can be estimated, like the mean queue length L_q (i.e. the average number of transactions waiting to enter the mutual exclusion zone) and the mean waiting time W_q (i.e. the time transactions must wait to enter that zone). Formulas are:

$$L_q = \frac{(\frac{\lambda}{\mu})^2}{1 - \frac{\lambda}{\mu}} \qquad W_q = \frac{\frac{\lambda}{\mu}}{\mu - \lambda}$$

With these formulas, and giving some standard values for each step duration, we can analyze the performance loss due to the longer mutual exclusion zone that appears when correctly managing constraints. Note, however, that the mutual exclusion zone is not the only step in the processing of transactions and that the number of concurrent operations in the underlying database is usually limited for perfomance reasons. So, in the original protocols, the output rate of the mutual exclusion zone is not the output rate of the whole system, as transactions must be effectively terminated in the database after they leave the mutual exclusion zone. As a result of this, in the original protocols we have assumed that there exists a second server –the DBMS involved in the transaction termination– whose service time (i.e., its μ) is equal to t_{commit} and whose arrival rate (λ) is equal to the output rate of the mutual exclusion zone. Note that for the original protocols we have considered as their queue lengths and queue waiting times the accumulated values from both queues: the one needed to access the mutual exclusion zone and the second one needed to terminate the transaction in the underlying DBMS.

A set of values representing the typical behavior of our MADIS middleware for the steps identified above is: $t_{val} = t_{cert} = 3\,ms$, $t_{term} = t_{commit} = t_{abort} = 25\,ms$ and $t_{hist} = 1\,ms$. With these values, the mean service time can be evaluated depending on the protocol version and the outcome of transactions. For WVR, the original version takes 3 ms whilst the correct one lasts 28 ms. This leads to rates of $\mu_{origWVR} = 333.33\,tr/sec$ (if we only consider its mutual exclusion zone, but later its second "server" –DBMS termination, persisting the updates– provides an effective $\mu_{origWVR} = 40\,tr/sec$) and $\mu_{extWVR} = 35.71\,tr/sec$. For the CBR case, the service time depends on the outcome of the transaction. In order to obtain an average service time, some values can be assigned to the abort rate due to conflicts and also to the abort rate due to integrity violations. With values of 3% and 2%, respectively, weigthed average service time is 3.97 ms for the incorrect version and 28.24 ms for the correct one. These values lead to rates of $\mu_{origCBR} = 251.89\,tr/sec$ (again, limited to $\mu_{origCBR} = 40\,tr/sec$ for the same reasons explained in the WVR case) and $\mu_{extCBR} = 35.41\,tr/sec$.

In order to maintain the system under its saturation point, arrival rates λ cannot be greater than service rates μ. Varying the value for the arrival rate from 5 to 30 tr/sec, plots in Figures 4 and 5 are obtained. Figure 4 shows the mean number of transactions waiting in the queue in order to enter the mutual exclusion zone. Values for both original protocols are quite low, as their service rate is always higher than the arrival

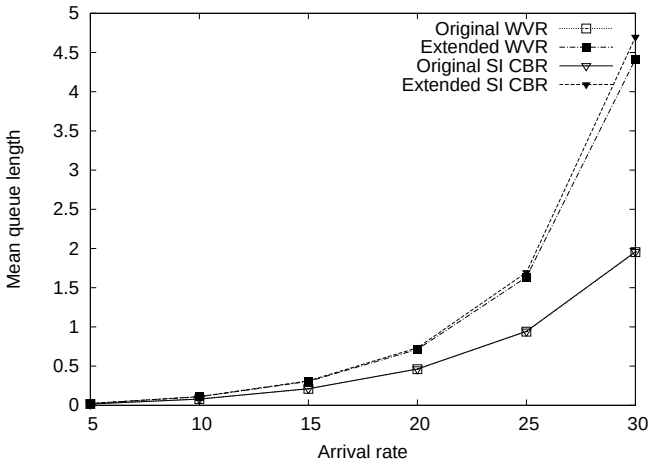

Figure 4: Values for L_q depending on λ

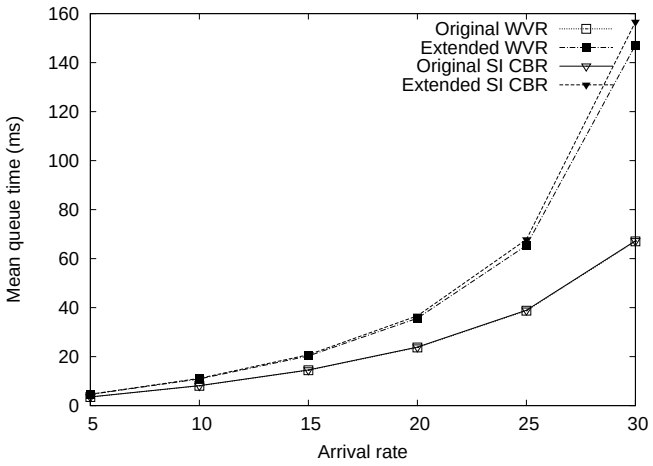

Figure 5: Values for W_q depending on λ

rate. On the other hand, both extended protocols present a notable increment in their graphs as the arrival rate becomes closer to the service rate. Figure 5 shows the mean time transactions must wait in order to access the mutual exclusion zone, measured in milliseconds. As expected, such waiting times are proportional to the queue lengths shown in Fig. 4. Experimental measures of transaction completion times were obtained in [23] for both the original and the extended protocol. Times for the extended version presented increments similar to the differences between mean queue times predicted in the previous graph. This supports the idea of considering the greater mutual exclusion zone as the main disadvantage of the proposed extended version. This proves that a correct integrity management demands much longer transaction completion times, and that some additional research is needed in order to find compatible optimizations able to reduce such times.

6 Conclusions

The literature on integrity checking in replicated database systems is extremely scant; solitary exceptions are few and peripheral, e.g., [25, 20]. 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 control-

ling 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 a transaction 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 transactions as ready to commit if no access conflict to shared data resources has been detected. That way, integrity may be lost in 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 first have described and then analyzed this problem in detail, for several well-known classes of replication protocols that are based on total order communication mechanisms. For some, we have seen that, surprisingly, support for integrity checking can be integrated seamlessly. For others, careful modifications are needed to make them work well also when integrity is checked by the DBMS at hand. We have proposed such extensions for each critical class. For CBR and WVR protocols (which have the best performance properties according to [26]), their extensions for integrity control prevent them from using most of the optimizations that are responsible for their good reputation. An analytical study is also included, showing the average additional time transactions must wait when managed by the extended protocol versions. These delays are due to the greater mutual exclusion zone, which forces transactions to wait before accessing it. This way, as the arrival rate of transactions increases, also the queue length and, therefore, the waiting time become larger. So, this opens a new line of research in the field of database replication, that could lead to efficient constraint-aware protocols, if new protocol optimizations are designed for overcoming the current limitations, as identified in this paper.

So far, we have only considered those integrity constructs that are actually supported by currently available DBMSs. However, our goal in the long run is to offer a transparent integration of support for transaction consistency, replication consistency and semantic consistency, where the latter is expressed by integrity constraints that are as general as possible. Up to now, the onus of ensuring integrity in replicated databases has been on the designers and users of applications. In the long run, this should give way to specifications of integrity constraints that can be supported automatically, just the way they are supported already in centralized, non-distributed database systems.

Acknowledgments

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

References

[1] Divyakant Agrawal, Gustavo Alonso, Amr El Abbadi, and Ioana Stanoi. Exploiting atomic broadcast in replicated databases. In *3rd International Euro-Par Conference*, pages 496–503, Passau, Germany, August 1997.

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

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

[4] Stephanie J. Cammarata, Prasadram Ramachandra, and Darrell Shane. Extending a relational database with deferred referential integrity checking and intelligent joins. In *SIGMOD Conference*, pages 88–97, Portland, Oregon, May 1989.

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

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

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

[8] 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.

[9] Javier Esparza-Peidro, Antonio Calero-Monteagudo, Jordi Bataller, Francesc D. Muñoz-Escoí, Hendrik Decker, and José M. Bernabéu-Aubán. COPLA - a middleware for distributed databases. In *APLAS*, pages 102–113, Shanghai, China, December 2002.

[10] 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.

[11] 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.

[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] Ricardo Jiménez-Peris, Marta Patiño-Martínez, Bettina Kemme, and Gustavo Alonso. Improving the scalability of fault-tolerant database clusters. In *ICDCS*, pages 477–484, 2002.

[14] 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.

[15] Gilles M. E. Lafue. Semantic integrity dependencies and delayed integrity checking. In *Eigth International Conference on Very Large Data Bases*, pages 292–299, Mexico City, Mexico, September 1982.

[16] 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.

[17] François Llirbat, Eric Simon, and Dimitri Tombroff. Using versions in update transactions: Application to integrity checking. In *23rd International Conference on Very Large Data Bases*, pages 96–105, Athens, Greece, August 1997.

[18] Francesc D. Muñoz-Escoí, Rubén de Juan-Marín, J. Enrique Armendáriz-Íñigo, and J. R. González de Mendívil. Persistent logical synchrony. In *7th International Symposium on Network Computing and Applications*, Toronto, Canada, July 2008.

[19] 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.

[20] 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.

[21] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. MIDDLE-R: Consistent database replication at the middleware level. *ACM Trans. Comput. Syst.*, 23(4):375–423, 2005.

[22] Fernando Pedone, Rachid Guerraoui, and André Schiper. Exploiting atomic broadcast in replicated databases. In *4th International Euro-Par Conference*, pages 513–520, Southampton, UK, September 1998.

[23] María Idoia Ruiz-Fuertes, Francesc D. Muñoz-Escoí, Hendrik Decker, José Enrique Armendáriz-Íñigo, and José Ramón González de Mendívil. Integrity Dangers in Certification-Based Replication Protocols. Technical Report ITI-ITE-08/13, Instituto Tecnológico de Informática, Valencia, Spain, May 2008.

[24] Can Türker and Michael Gertz. Semantic integrity support in SQL:1999 and commercial (object-)relational database management systems. *The VLDB Journal*, 10:241–269, June 2001.

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

[26] 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.