Managing Multiple Isolation Levels in Middleware Database Replication Protocols

Josep M. Bernabé-Gisbert, Raúl Salinas-Monteagudo, Luis Irún-Briz, Francesc D. Muñoz-Escoí

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

{jbgisber, rsalinas, lirun, fmunyoz}@iti.upv.es

Technical Report ITI-ITE-06/03

Managing Multiple Isolation Levels in Middleware Database Replication Protocols

Josep M. Bernabé-Gisbert, Raúl Salinas-Monteagudo, Luis Irún-Briz, Francesc D. Muñoz-Escoí

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

Technical Report ITI-ITE-06/03

e-mail: {jbgisber, rsalinas, lirun, fmunyoz}@iti.upv.es

Abstract

Many database replication protocols have been designed for guaranteeing a serialisable isolation level, since it is appropriate for almost all applications. However, it also requires a tight coordination among replicas and might generate high abortion rates with some workloads. So, other isolation levels have also been considered, such as snapshot isolation and cursor stability, but none of the previous works has proposed an overall support for more than one isolation level at the same time. This paper explores such a research line.

KEYWORDS: Middleware, Database Replication, Isolation Levels, High Availability

1 Introduction

Many data replication protocols have been published for years [5, 6, 11, 22], and they have always been centred on a single isolation level. Indeed, when multiple isolation levels have been presented [11], a separate protocol has been designed for each of them. There is no problem with this approach, since it makes possible a thorough description, discussion or justification for each protocol. However, applications may often require that their transactions were executed in different isolation levels, mainly for improving the access time of such transactions that tolerate reading non-strictly-consistent data. This necessity of managing multiple isolation levels is a main issue for database applications, and has been even included as part of several "*standard*" benchmark applications, such as the one defined in the TPC-C [20] specification. In such benchmark, its New-Order, Payment, Delivery and Order-Status transactions require the ANSI *serialisable* level, and the same set of transactions requires that other transactions accessing the same data (besides the Stock-Level one, that is also included in the benchmark) use the *repeatable read* level, whilst its Stock-Level transaction only demands the *read committed* level. Many applications follow similar patterns on their sets of transactions.

Any serious centralised *database management system* (DBMS, on the sequel) is able to manage without problem multiple isolation levels at a time (i.e. for several concurrent transactions), but a database replication middleware is faced with some inconveniences for providing such service. Mainly, there is no trivial way of coordinating different replication protocols, each one providing support for a single isolation level.

As a result, when such applications must be managed, there are only three options for dealing with them. The first one is to discard the modern database replication techniques, following a distributed locking approach for concurrency control. The rules for providing the most important isolation levels have already been specified for locking techniques [4], and are easily implementable in distributed systems with distributed locks. However, distributed locking has proven to show a poor performance when compared

with replication techniques based on total ordered write-set propagation [22]. The second approach consists in selecting a set of modern protocols with similar techniques and different isolation levels, defining from scratch the rules to be followed when different levels must be combined. This may be achieved when such protocols use similar solutions for the most important parameters that define a replication protocol [21]: server architecture, replica interaction, and transaction termination. The last option consists in supporting a single isolation level –the strictest one being needed–, thus requiring that all transactions were executed using such level. This leads to poor performance or higher abortion rates for those transactions that would have tolerated a more relaxed isolation level.

This paper describes a general scheme for designing replication protocols that support multiple isolation levels. Although there are multiple levels that could have been supported, this first solution only considers four basic alternatives that are quite similar to the ANSI standard levels, according to the generalised definitions proposed in [2].

The rest of the paper is structured as follows. Section 2 presents our system replication model. Section 3 outlines the supported isolation levels and how they have been implemented in previous replication protocols. Section 4 describes our solution, whilst section 5 compares it with other related work. Finally, section 6 concludes the paper.

2 System Model

We assume a partially synchronous distributed system –where clocks are not synchronised but the message transmission time is bounded– composed by N nodes where each node holds a replica of a given database; i.e., the database is fully replicated in all system nodes. These replicas might fail according to the *partial-amnesia crash* failure model proposed in [8], since all already committed transactions are able to recover, but on-going ones are lost when a node crashes. However, we do not focus on recovery issues in this paper.

Each system node has a local DBMS that is used for locally managing transactions, and that provides the mechanisms needed for ensuring the standard ANSI isolation levels. On top of the DBMS a middleware is deployed in order to provide support for replication. This middleware also has access to a group communication service that should support atomic multicast [10] (or uniform atomic multicast if failures are considered). Our solutions might be also used in non middleware-based systems, but this requires at least a minimal modification of the DBMS core, and such extension depends on the DBMSes being considered. We do not describe such dependencies in this paper, so our discussion is better tailored for middleware solutions.

The replication model being used is *read one, write all available* (ROWAA, on the sequel), since in almost all replication protocols only the transaction write-sets are propagated. The comparison made in [22] also proves that this behaviour provides better performance than any other that requires read execution in all replicas.

3 Isolation Levels

Many current relational DBMSes support the standard ANSI isolation levels, as defined in [1]. However, the definitions given in such standard are not enough precise, as they were criticised in [4]. In that paper, its authors distinguished between strict interpretations of the *phenomena*[1] discussed in the standard, and loose interpretations, showing with some examples that with a strict interpretation some non-desired anomalies were possible in each isolation level. As a result, the standard specification must be understood using the loose interpretations outlined in [4] that generate stricter levels of isolation. Some traditional implementations based on locks already supported such loose interpretations, but others did not. Thus, some DBMSes using multi-version concurrency control (MVCC, for short) had followed the strict phenomena interpretations. Consequently, they only provided a *snapshot* isolation level (as defined in [4]) when they were asked for a *serialisable* one.

Unfortunately, both the loose phenomena interpretation and the lock-based concurrency control proscribed some transaction executions that were perfectly legal for the required isolation levels. Adya et al.

[1] The term *phenomenon* refers to consistency anomalies that should be avoided when transaction isolation is enforced.

[2] detected such problems and specified again the isolation levels. Their specifications are more precise than those presented in [4] and also implementable with optimistic concurrency control (and this is the most common in replicated systems, since transactions are generally allowed to proceed until they request their commit and get validated or certified).

So, in order to be complete, we provide on the sequel the phenomena definitions given in [2] that should be proscribed in some of the standard isolation levels. They are the following[2]:

G0 (**Write cycles**): A history H exhibits phenomenon G0 if DSG(H) contains a directed cycle consisting entirely of write-dependency edges.

In this definition, DSG(H) is a *direct serialisation graph* [2] based on direct conflicts between committed transactions. Additionally, a write dependency occurs when one transaction overwrites a version written by another transaction.

G1a (**Aborted reads**): A history H shows phenomenon G1a if it contains an aborted transaction T1 and a committed transaction T2 such that T2 has read some object modified by T1.

G1b (**Intermediate reads**): A history H shows phenomenon G1b if it contains a committed transaction T2 that has read a version of object x written by transaction T1 that was not T1's final modification of x.

G1c (**Circular information flow**): A history H exhibits phenomenon G1c if DSG(H) contains a directed cycle consisting entirely of dependency edges.

In this phenomenon definition, a dependency edge is either a write dependency (already defined in G0 description) or a read dependency. A read dependency arises when a transaction reads some items written by another transaction, or when the results of a transaction read (using a predicate) are modified by a write operation made by another transaction (including value changes, as well as element additions or removals in such results). The results in such predicate-based queries are all items accessed, plus their corresponding *truth degree* for the predicate, even if they do not match such predicate. Those items that match the predicate are added to the history as separate individual reads. So, the write operations that include or remove elements in a predicate read are those that inserted or deleted such items in or from their respective tables.

G2 (**Anti-dependency cycles**): A history H exhibits phenomenon G2 if DSG(H) contains a directed cycle with one or more anti-dependency edges.

Informally, an anti-dependency arises when a transaction overwrites a version observed by some other transaction.

When the anti-dependencies arise between transactions that do not use predicate-based reads, a **G2-item** phenomenon occurs. In the general case (i.e., with the **G2** phenomenon) both kinds of read operations are considered (predicate-based and item-based).

These definitions match respectively the original P0, P1, P2 (equivalent to G2-item) and P3 (equivalent to G2) phenomena definitions of the ANSI standard. However, P1 was decomposed in three different G1 subcases in order to eliminate the problems detected in the loose interpretations proposed by [4]. Consider also that G1 implicitly includes the G0 phenomenon, so if a level proscribes G1 it also proscribes G0. With these phenomena definitions, Adya et al. specify some portable levels of isolation that we summarise in table 1. *Portable* refers here to the possibility of implementing all these levels with any concurrency control approach, and this characteristic is very convenient for replicated environments.

We use these portable isolation level specifications in the following sections for building a set of rules that might be used for defining general replication protocols able to manage multiple isolation levels.

[2] These definitions are in a summarised form. The interested reader should read [2] for complete and formal definitions.

Portable level	Disallowed phenomena	Equivalent ANSI level
PL-1	G0	READ UNCOMMITTED
PL-2	G1	READ COMMITTED
PL-2.99	G1, G2-item	REPEATABLE READ
PL-3	G1, G2	SERIALISABLE

Table 1: Portable ANSI isolation levels

4 A General Replication Protocol

There are many ways of writing a database replication protocol, since there are some parameters that define how such protocol should behave. Thus, in [21] three parameters of this kind were identified: server architecture, server interaction and transaction termination. Each one of these parameters can take two different values, generating eight different classes of protocols.

A general enough replication protocol that supports multiple isolation levels should be able to match any implementable protocol in all these classes. Unfortunately, there are big differences among such classes, and it would be quite difficult to provide a single principle easily adaptable for all classes.

For instance, the *server architecture* parameter distinguishes between protocols based on a primary server where all transactions should be forwarded, and protocols that allow the execution of transactions in any site (defined as *update everywhere* replication). Regarding concurrency control and isolation, the primary server approach does not imply any problem, since the execution of transactions is fully centralised and we may rely on the local concurrency control mechanisms in such primary copy; i.e., the protocols we are looking for are trivially implementable in this kind of replication since only one replica should take care of concurrency control, and its semantics can be directly driven by the underlying DBMS.

Besides this, other classes can be easily discarded due to other problems not related with isolation, but with other requirements such as performance. For instance, linear interaction (one of the alternatives for the server interaction parameter) implies extremely expensive overheads on communication among replicas, and complicates a lot the recovery subprotocols. So, it is commonly discarded in the general case.

As a result of this, only two of the original eight classes identified in [21] should be surveyed as general replication protocols in this paper: those based on an *update everywhere* server architecture, with *constant* server *interaction* and with either *voting* or *non-voting* transaction termination.

So, once identified the target protocol classes to be managed by our general solution, let us see which implementation choices we assume and how a general protocol can be defined, also proving how is it able to avoid each of the general phenomena described in [2].

4.1 Protocol Implementation Features

There have been multiple database replication protocols in the *update everywhere* server architecture with a *constant* server *interaction* [11, 19, 17, 9, 12, 23]. Many of them share the following characteristics, proving to be extremely adequate for replication purposes. Thus, we will take them as a basis for designing our general protocol:

- Since they belong to the update everywhere server architecture, transactions can be initiated in any replica. There is no special replica that centralises transaction management.
- As they also belong to the constant interaction class, only a constant number of messages are exchanged among replicas. In the common case, such messages are used for propagating the updates, and they are needed once the commit has been locally requested in the initiating replica. Although other solutions are possible, we will limit our discussion to protocols that propagate the transaction data at the end of each transaction; i.e., when the application has locally requested the commit.
- Write-set (and, in some cases, read-sets [23]) propagation is made using an *atomic multicast*; i.e., a multicast with message delivery in total order. This ensures that all replicas see the same sequence

of write-sets (and, if needed, read-sets); i.e., the same sequence of transactions.

- The underlying DBMS provides support for the isolation level being requested by the user transactions. Thus, local transactions can be managed by the underlying DBMS, and the middleware must ensure that the mix among remote and local transactions also follows the requested isolation levels.

Taking these features as a basis, the design of a database replication protocol is reduced to check for conflicts between local transactions and write-sets being delivered, or between those write-sets. Additionally, two schemes for such checking are possible, depending on the transaction termination alternative being chosen [21]: either a voting phase is needed in the transaction termination, or all replicas behave deterministically in the certification phase and all arrive to the same decision without needing any explicit coordination. But protocols based on voting can be divided in two different subclasses: those that are symmetrical, requiring a vote by every replica (for instance, in order to cope with unilateral abortions [16] or other sources of non-determinism), or others that rely on a delegate server[3], who imposes its decision to the rest of replicas (this approach is referred to as *weak voting replication* in [22]).

Between these three approaches for terminating transactions, we choose only the weak voting replication approach, since the other two have the following problems:

- **Non-voting termination**. In this case, if the *serialisable* isolation level has to be supported, read-sets must be propagated [16]. Although there are some techniques that allow read-set propagation with minimal costs [23], read-set collection can be a problem for long transactions.

- **Symmetrical voting termination**. The communication needs of this voting phase, plus those already paid for total order write-set delivery generate an overall communication cost similar to a 2PC. This scheme might be supported if a non-atomic multicast is used, such as in the protocols described in [3], but with the scheme outlined in this section its costs are too high to consider it appropriate.

Although these two approaches will not be the focus of this paper, the solution described in the following sections might be easily adapted to both of them. In all approaches a validation phase is needed, and the issues being considered in these validations are not too different among these approaches.

4.2 A General Scheme

Our general scheme for supporting multiple isolation levels is based on the following principles:

- If multiple isolation levels should be supported, a protocol for the strictest isolation level –among those to be supported– has to be selected.

- When a transaction is started, its intended isolation level should be requested to the underlying DBMS.

- When a transaction reaches the commit phase, and its write-set (and, in some cases, read-set) is propagated, its isolation level identifier has to be included into such propagation message.

- The validation step needed in the replication protocol for deciding whether a transaction must commit or abort has to consider the isolation levels of all the transactions being checked. The rules to check between transactions that have requested different isolation levels have to consider the phenomena to be proscribed by such isolation levels.

These principles are general enough to be applied to any transaction termination approach (i.e., weak voting, symmetrical voting, and non-voting cases). In this paper, such scheme will be applied to the weak voting replication approach. So, this kind of database replication must be considered as a case study for our general scheme.

A database replication protocol based on weak voting replication consists in the following steps [22]:

[3] The delegate server is the replica that has initiated the particular transaction.

1. When a delegate database server DS_d receives a transaction T from a client C, it executes the transaction but delays its write operations.

2. When client C requests the transaction commit, the transaction write-set is propagated to all replicas using atomic broadcast. Note that if a transaction has an empty write-set (i.e., it is a read-only transaction) no broadcast is needed and it immediately commits.

3. When such write-set message is delivered, the delegate server determines if conflicting transactions have been committed.

4. If so, transaction T must be aborted. Otherwise, it should be committed. Depending on the result of this validation, the replica DS_d uses a reliable broadcast to propagate this result.

5. Concurrently with these two last steps, the other replicas have received the same write-set and have locally applied it. Once they receive the validation result, they take the appropriate action (either to abort or to commit transaction T).

This protocol is able to provide a *serialisable* isolation level, but the key for this resides in its step number 3, where the write-set is validated and a result for each transaction is decided. Depending on the rules being used for determining "conflicting" transactions other isolation levels can be obtained.

For applying our general scheme, we only need to extend minimally this sample algorithm in order to:

a) Extend its step 1, requesting to the underlying DBMS the appropriate isolation level.

b) Extend its step 2, including the isolation level of such transaction as an additional field into the write-set message.

c) Adapt its step 3, using the appropriate conflict checking rules for each isolation level.

The last extension deserves further explanation and is thoroughly discussed on the sequel.

4.3 Avoiding General Phenomena

In this section, we will show how the general phenomena presented in section 3 can be proscribed using some concurrency control techniques and validation checks in the protocol outlined above. To begin with, let us start with the mechanisms needed for guaranteeing the isolation level PL-3, and later discussing how the other levels (PL-2.99, PL-2, and PL-1, respectively) can be ensured. In all these variants, read-set propagation is not needed since read accesses are only checked against write-sets (from either local or remote transactions) in the delegate replica where such transactions have been started.

4.3.1 Portable Level PL-3.

This portable level is almost identical to the ANSI *serialisable* level. It requires that both G2 and G1 phenomena were proscribed.

Using traditional locking techniques, this isolation level needs long read and write locks. In a replicated environment, these locks should be combined with the total order being guaranteed by the atomic broadcast.

An example of database replication protocol that uses the weak voting replication approach ensuring a *serialisable* level is the SER protocol of [11]. This solution also uses a lock-based concurrency control, requesting long locks in the delegate server for both kinds of accesses (reads and writes), and requesting also write locks when the write-set is delivered in remote replicas. As a result of this, its validation procedure distinguishes the following actions:

1. The write-set application may get blocked in non-delegate replicas if the requested locks conflict with the locks already acquired by other transactions that have been previously delivered following the total order of the atomic broadcasts. Thus, such write-set application simply waits for the completion of such conflicting transactions, and no rollback is needed in this case.

2. Otherwise, if such lock request collides with some local read locks that belong to transactions whose write-sets have not been delivered, such local transactions are aborted.

Thus, in order to forbid phenomenon G2, we must ensure that no cycle with at least one anti-dependency edge might be created in any execution of this protocol. Recall that T1 has an anti-dependency on T2 if T1 overwrites an item (or the result of a predicate evaluation) read by T2. In this protocol cycles are prohibited, since the total order delivery ensures that all transactions are sequentially ordered and thus, it is impossible that the same transaction initiates and terminates a cycle of dependencies (it will be either the first or the last in such order, but not both since the local concurrency control in all replicas also prevents such kind of cycles among local transactions).

Supose that a node N_i is trying to apply T_i's write-set WS_i. Validation action 1 ensures that a WS_i is never applied before any previous conflicting delivered transactions because WS_i will be blocked until these transactions commit. Additionally, all not yet delivered local transactions with read locks on items accessed by T_i never commit before T_i because validation action 2 would abort them. Both actions combined ensure that the destination transaction for every dependency or anti-dependency edge commits after its source transaction. This implies a sequential committing order, and justifies the avoidance of phenomenon G2.

In a similar way, G1 is avoided since G1c is also proscribed due to the total order delivery, introducing a sequential order of transactions that prevents cycles from appearing in the DSG(H) of any history H. Moreover, the use of local long write locks avoids phenomena G1a and G1b. Thus, G1a (aborted reads) is avoided because due to the long write locks, it is impossible that a transaction T2 would have read an item previously written by a transaction T1 that finally had aborted. The same happens with G1b (intermediate reads).

If, instead of a lock-based concurrency control other local concurrency control approaches were used similar validation actions would be needed. For instance, with MVCC, the validation action 1 would have had the same behaviour, since write conflicts lead to blocking with such kind of concurrency control. On the other hand, the validation action 2 would have had a difficult management with this kind of concurrency control, since no locks are requested for reading. As a result, local read operations should be translated into SELECT FOR UPDATE statements in order to detect such read-write conflicts and a mechanism such as the one described in [15] would be needed for dealing with such kind of conflicts, leading to the abortion of these local transactions.

As it has been explained for lock-based concurrency control, this solution proscribes both G2 and G1 phenomena. Both G2 and G1c are prohibited by the total order being used for write-set delivery, whilst both G1a and G1b are trivially avoided by the underlying MVCC, since the versions being read by each transaction have been generated by transactions already committed (intermediate versions are always private for the transaction that has generated them, when a *serialisable* isolation level is requested in a MVCC system).

4.3.2 Portable Level PL-2.99.

This second portable level (PL-2.99) is almost equivalent to the ANSI *repeatable read* isolation level. For ensuring it, in lock-based concurrency control long locks are used for write and item-read operations, but only short locks when the read operations use a predicate. If we plan to use an underlying DBMS with this kind of concurrency control, we may use the same validation actions than we described for PL-3 –but considering that now predicate reads only need short locks and, as a result, will not get aborted by validation action 2–. Since transactions that need PL-2.99 have requested the *repeatable read* isolation level to the underlying DBMS, conflicts among PL-3 and PL-2.99 writing transactions will be correctly managed by such DBMS. In case of conflicts between remote write-sets and local reading transactions, the middleware will be able to detect such conflicts using the mechanisms outlined in [15]; i.e., reading one of the system-catalogue tables that records those transactions that have been blocked due to conflicts with other transactions.

As a result of this, no modification over the solution already described for PL-3 is needed for achieving PL-2.99 at the middleware level. Additionally, the justification of the proscription of the G2-item and G1 phenomena is identical to those already given above for PL-3.

If MVCC is used, there is no way to allow phenomena G2 for predicate-based reads; i.e., allowing anti-dependency edges that overwrite predicate reads. Some DBMS based on MVCC are not able to provide an ANSI *repeatable read* isolation level: PostgreSQL [18] is an example. This kind of concurrency control ensures that each transaction gets item versions that correspond to the moment when such transaction was started. As a result of this, a write operation generates a new version for every updated item, but such version can not be accessed by concurrent transactions. So, the isolation achieved with this concurrency control technique for read accesses is more or less equivalent to using long read locks in a lock-based technique. Thus, level PL-2.99 is not achievable with this kind of concurrency control. On the other hand, this kind of concurrency control easily provides the *snapshot* isolation level [4] that shares some of the characteristics of this PL-2.99 level but that is not equivalent to it.

4.3.3 Portable Level PL-2.

In this portable level (more or less equivalent to the ANSI *read committed* level), phenomenon G2 is completely allowed, but G1 is still proscribed. So, anti-dependency edges may be present, being able to close dependency cycles among a given set of transactions. In an implementation based on locks this level only requires short locks for read accesses, and long locks for writes.

Regarding our sample protocol described for the PL-3 level, in this case validation action 2 can be completely removed since the application of a remote write-set would not abort any local transaction. As a consequence, no abortion is generated in such validation actions and this means that the reliable broadcast needed in step 4 of the sample protocol outlined in section 4.2 will not be needed by transactions that belong to the PL-2 and PL-1 isolation levels.

If the underlying DBMS supports this PL-2 level, both G1a and G1b phenomena are proscribed, since these phenomena are caused by read accesses and they can only be local in our sample general protocol. Phenomenon G1c should also be proscribed. To this end, no cycle of dependency edges should be allowed by our protocol. This is easily ensured, since validation action 1 ensures that write-dependencies can only be established in the order being imposed by the atomic delivery of write-sets, and this prevents the appearance of write-dependency cycles. Read-dependencies can be locally present in some replicas (in those where each transaction had its delegate server) but they would not be able to close any cycle. Otherwise, a single transaction would have read some information from a write-set that occurs after it in the write-set total order delivery, and this is impossible (since local concurrency control mechanisms prevent a transaction from reading something that has not yet been committed).

As stated above, phenomenon G2 should be allowed. So, a history like the following one should be permitted (it follows the notation proposed in [2]):

$$H\text{: } r_1(\text{Weight}{>}50{:}x_0,60;y_0,51)\ r_1(x_0,60)\ r_2(y_0,51)\ w_2(y_2,50)\ c_2\ r_1(y_2,50)\ c_1$$

In such sample, transaction T1 gets all items with a weight greater than 50 and two items are returned, x and y. Concurrently T2 updates y weight, setting it to value 50. Finally, when T1 gets y's data it recovers a 50 value that does not match the predicate being used. So, we have a T1 read-dependency on T2 and a T2 anti-dependency on T1. This situation is trivially allowed by our general protocol since read-only transactions are not broadcast to all replicas, and in this case T1 –a read-only transaction– has been allowed to commit by the local concurrency control on its delegate replica. Additionally, T2 is broadcast and committed without problems in all database replicas.

Note that the solution described for this portable level is independent on the underlying concurrency control mechanism being used.

In a general solution, with other validation techniques for its PL-3 level, the checks being made for write-set collisions should be maintained in this PL-2 level, but those checks associated to write-read conflicts can be eliminated if a local concurrency control able to provide PL-2 guarantees is present in each replica.

4.3.4 Portable Level PL-1.

Finally, portable level PL-1 (similar to *read uncommitted*) only proscribes phenomenon G0; i.e., write-dependency cycles. As already discussed in the previous case, write-dependency cycles are avoided if the

write-set delivery order is respected when such write-sets are applied on each replica. Again, as in the previous case, the specific validation action 2 is not needed. However, now the underlying DBMS needs to enforce locally only the PL-1 level and this only places some restrictions on write accesses but never on reads. Due to this type of read management, read-dependencies may arise, allowing thus the occurrence of phenomena G1a, G1b and G1c.

Note also that in many DBMSes (for instance, IBM DB2) the ANSI *read uncommitted* isolation level does not allow writes by default, although such behaviour can be reset. If the programmer accepts its default mode, transactions with this isolation level will not generate any trouble for our general protocol, since they will be locally committed and no write-set will need any kind of management for such transactions.

The general solution to this isolation level is the same already stated for PL-2: to take care only for write-write conflicts when write-sets are delivered. The single difference with that level is that now some local concurrency control support specific for a PL-1 level is assumed. Note that not all MVCC DBMSes provide a so relaxed isolation level. For instance, PostgreSQL [18] is able to provide support for *read committed* (PL-2) and its *serialisable* (PL-3) (actually, *snapshot*) levels, but not for *read uncommitted* (PL-1) nor for *repeatable read* (PL-2.99). The same happens in Microsoft SQL Server 2005 [13] when it is configured for using optimistic concurrency control techniques.

4.3.5 Summary.

The study made in this section has shown that dealing with multiple isolation levels in a middleware-layer replication protocol is feasible. To this end, a good replication protocol for the strictest isolation level to be supported has to be chosen and local support for all the intended isolation levels should be present in the underlying DBMS. If all these requirements can be coped with, the concurrency control checks will be quite similar for all these isolation levels, but part of them are lost in the looser levels being supported (mainly, those related to read-write conflicts). Thus, when a write-set arrives, the receiving replica must only apply the checks associated to the isolation level of such incoming write-set. Note also that some non-standard isolation levels might require other techniques for avoiding their proscribed phenomena (e.g., in case of the *snapshot* or *cursor stability* levels), but such cases will be studied in further works to be completed in the near future.

In order to summarise this section, figure 1 shows the resulting database replication protocol supporting the standard isolation levels that we have generated taking the SER protocol of [11] as its basis.

1. When a transaction T_i starts, set its isolation level on the local DBMS.
2. When T_i is locally terminated:
   2.1. If it is read-only, it directly commits.
   2.2. Otherwise, get its write-set (WS_i) and its isolation level (IL_i) and broadcast them in total order to all replicas.
3. Upon (WS_i,IL_i) delivery:
   3.1. For each operation on WS_i:
      a) If there is a read-write conflict with a local transaction T_j with $IL_j >$ PL-2 that has not delivered its write-set, abort T_j.
      - If T_j has broadcast (WS_j,IL_j), broadcast abort(T_j).
      b) If there is any other conflict with another local transaction T_j, wait until T_j terminates (WS_j should have been delivered before).
   3.2. Apply WS_i locally.
   3.3. If T_i is a local transaction and $IL_i >$PL-2, broadcast commit(T_i).
   3.4. If $IL_i <$PL-2.99, commit T_i.
4. Upon commit(T_i) or abort(T_i) delivery, commit or abort T_i, depending on the received message.

Figure 1: A sample of general replication protocol.

5 Related Work

Most current database replication protocols aim to provide support for only the *serialisable* [22, 19, 17] or *snapshot* isolation levels [9, 12], since they are needed by a wide variety of applications. However, there have also been some works that have studied multiple levels of isolation, either providing protocols for each of them [11, 23] or by specifying new definitions of such levels [4, 2]. But none of these works has supported more than one isolation level in a single replication protocol. As a result, they have designed good solutions for a single level but they can not be merged easily into a single protocol, since such solutions are specifically tailored for their target level.

Despite this, there have been some attempts for providing support for multiple isolation levels in a single protocol. This was one of the aims in the GlobData project [19] and some initial solutions were provided in [14]. But the isolation levels defined in GlobData were not the ANSI standard ones, since GlobData was a system with an object-oriented interface able to provide an object-oriented replicated database using relational database replicas, and for those systems, there were considered another set of behaviours. So, such initial solution is not comparable to the one discussed in this paper.

A complementary solution is described in [7], where such specific protocols can be concurrently supported by a meta-protocol that directly manages all transactions asking to the installed protocols the results of the concurrency control checks needed for deciding if a transaction must be committed or rolled back. This meta-protocol allows the concurrent installation of multiple replication protocols highly optimised for their target applications, isolation levels and assumed system model. So, different applications with different requirements might use their tailored replication protocols using this kind of solution. Moreover, the support provided by this meta-protocol easily allows the dynamic exchange of the installed protocols; i.e., one application would be able to select at run-time the protocol most convenient for its objectives or for the current system environment (workload in each node, network traffic, failures that have occurred, etc.). However, the scheme being imposed by this meta-protocol requires a strong adaptation of the protocols to its required interfaces and this implies some re-writing effort.

Other papers deserve special attention although their objectives are not exactly the same as ours. In [23] two new protocols are described. The first one is an evolution of the original *Database State Machine* (DBSM) approach [16] that provides *snapshot* isolation, while the other uses some rules that are similar to those of the original DBSM but it is able to guarantee serialisability without transferring read-sets. The latter manages conflict classes (or logical sets) and introduces dummy writes that are able to simulate the read-sets. Unfortunately, these two protocols are not based on the same principles (the first one uses an *update everywhere* server architecture with non-voting termination, whilst the second one uses a primary copy server architecture) and, as a result of this, they will be difficult to merge in a single protocol supporting both isolation levels. On the other hand, it analyses two of the most used isolation levels.

In [11], two different protocols were provided for supporting the same isolation levels. Moreover, in [11] other non-standard isolation levels (*cursor stability*, for instance) were also supported by other protocols. However, although all of them share similar architectures (update everywhere server architecture and constant interaction), nothing is said about merging all levels in a single protocol.

6 Conclusions

This paper has presented a general scheme for designing middleware database replication protocols supporting multiple isolation levels. It is based on progressive simplifications of the validation rules used in the strictest isolation level being supported, and on local (to each replica) support for each isolation level in the underlying DBMS.

Such scheme provides a uniform management for all isolation levels, needing minimal extensions to the original database protocol (the one initially designed for the strictest level). This support for multiple isolation levels is specially fruitful for those applications that manage multiple kinds of transactions, since they get an improved performance for those transactions able to run with the loosest isolation levels. Without the described general protocol, an application of this kind would have used a single replication protocol supporting the strictest needed isolation level, and this would have penalised their performance, or would increase the abortion rate of the most relaxed transactions.

References

[1] ANSI X3.135-1992, American National Standard for Information Systems - database language - SQL, November 1992.

[2] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Proc. of IEEE Intnl. Conf. on Data Engineering*, pages 67–78, San Diego, CA, USA, March 2000.

[3] J. E. Armendáriz-Íñigo. *Design and Implementation of Database Replication Protocols in the MADIS Architecture*. PhD thesis, Depto. de Matemática e Informática, Univ. Pública de Navarra, Pamplona, Spain, February 2006.

[4] 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.

[5] P. A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. on Database Sys.*, 9(4):596–615, Dec. 1984.

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

[7] F. Castro-Company and F. D. Muñoz-Escoí. A exchanging algorithm for database replication protocols. Technical report, Instituto Tecnológico de Informática, Valencia, Spain, 2006.

[8] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

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

[10] 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.

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

[12] 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.

[13] Microsoft Corp. Choosing row versioning-based isolation levels, June 2006. Microsoft Developer Network. Accessible in URL: http://msdn2.microsoft.com/en-us/library/ms188277.aspx.

[14] F. D. Muñoz, L. Irún, P. Galdámez, J. Bernabéu, J. Bataller, and M. C. Bañuls. Consistency protocols in GlobData. In *Proc. of X Jornadas de Concurrencia*, pages 165–178, Jaca (Huesca), Spain, June 2002.

[15] F. D. Muñoz-Escoí, J. Pla-Civera, M. I. Ruiz-Fuertes, L. Irún-Briz, J. E. Armendáriz-Íñigo, and J. R. González de Mendívil. Dealing with writeset conflicts in database replication middlewares. In *Proc. of XIV Jornadas de Concurrencia y Sistemas Distribuidos*, San Sebastián, Spain, June 2006.

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

[17] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.

[18] PostgreSQL Global Development Group. PostgreSQL 8.1: Concurrency control, June 2006. Accessible in URL: http://www.postgresql.org/docs/8.1/static/mvcc.html.

[19] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. The GlobData fault-tolerant replicated distributed object database. In *EurAsia-ICT*, pages 426–433, 2002.

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

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

[22] 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.

[23] V. Zuikeviciute and F. Pedone. Revisiting the database state machine approach. In *Proc. of VLDB Workshop on Design, Implementation and Deployment of Database Replication*, Trondheim, Norway, September 2005.