A Weak Voting Database Replication Protocol Providing Different Isolation Levels

March 13, 2007

J.R. Juárez[1] — J.E. Armendáriz[2] — J.R. González de Mendívil[1]
F.D. Muñoz-Escoí[2] — J.R. Garitagoitia[1]

(1) Departamento de Matemática e Informática
Universidad Pública de Navarra, Campus de Arrosadía s/n, 31006 Pamplona, Spain
{jr.juarez, mendivil, joserra}@unavarra.es

(2) Instituto Tecnológico de Informática
Universidad Politécnica de Valencia, Camino de Vera s/n 46022, Valencia, Spain
{armendariz, fmunyoz}@iti.upv.es

ABSTRACT. *Database replication protocols have been usually designed in order to support a single isolation level. This paper proposes a middleware replication protocol able to manage three different isolation levels over multi-version DBMSs that provides SI level: GSI, SI, and serializable. This ensures a better support for applications that demand different isolation levels for their transactions. Additionally, this protocol is also able to merge the coordination of the replicas for each isolation level, using a weak voting approach for all of them, whilst other recent protocols need a certifying technique for GSI, or a 2PC rule for serializable level.*

RÉSUMÉ. *Des protocols de réplication de bases de données ont été habituellement conçus pour supporter un seul protocole d'isolement. Cet article propose un protocole middleware de réplication qui peut contrôler trois niveaux différents d'isolement sur SGBD multi-version qui offrent le niveau SI : GSI, SI et serializable. Ceci assure un meilleur soutien des applications avec transactions qui demandent niveaux différents. En plus, ce protocole peut également fusionner la coordination des copies pour chaque niveau d'isolement, en utilisant un approche de vote faible pour tous, tandis que les autres protocoles récents ont besoin d'une technique d'attestation pour GSI, et de la règle de 2PC pour le niveau serializable.*

KEYWORDS: *Distributed data, replication protocols, isolation levels, middleware*

MOTS-CLÉS : *données distribuées, protocoles de réplication, niveaux d'isolement, middleware*

* Work supported by the Spanish Government under research grant TIN2006-14738-C02.

1. Introduction

Snapshot Isolation (SI) is a transaction isolation level introduced in [BER 95] and implemented, using multiversion concurrency control, in several commercial Database Management Systems (DBMS) as Oracle, PostgreSQL or Microsoft SQL among others. SI provides a weaker form of isolation than the serializable level [BER 87]. Under SI, a transaction reads data from a snapshot so that it sees all the updates done by transactions that committed before the transaction started its first operation. The resulting modifications due to its writes are installed when the transaction commits. However, a transaction will successfully commit if and only if there is not a concurrent transaction that has already committed before and some of its written items were also written by the transaction that wants to commit. Read-only transactions that are executed under SI level are neither delayed, blocked, nor aborted and they never cause update transactions to block or abort. This behavior is important for workloads dominated by read-only transactions, such as those resulting from dynamic content Web servers. These characteristics turn SI into an attractive isolation level because it provides sufficient data isolation while it maintains a good performance.

Many enterprise applications demand high availability and fault-tolerance since they have to provide continuous service to their users. For achieving such functionalities, the common solution consists in deploying multiple replicas of the information being used by such applications. One of the available database replication approaches is to deploy a middleware architecture [IRÚ 05, LIN 05, PAT 05] that creates an intermediate layer that features data consistency, being transparent to the final users, isolating the DBMS details from the replication management. This simplifies and provides a great flexibility to the development of replication protocols. Furthermore, middleware solutions can be maintained independently of the DBMS and may be used in heterogeneous systems. Recently, several database replication protocols based on the middleware approach have been proposed using DBMSs providing SI isolation level [MUÑ 06, LIN 05, ELN 05].The implementation of these protocols is based on a *certification* process to commit a transaction in the system. The basic mechanism behind these protocols follows the eager update everywhere replication strategy, which establishes that first a transaction is locally performed and then changes grouped in a writeset are propagated to the rest of the sites before committing. This strategy based on the use of the Read-One Write-All-Available (ROWAA) approach [GRA 96] allows these protocols to be able to scale quite well. In certification protocols, writesets are total order broadcast and at their delivery they are compared with the ones contained in a log. This log stores the writesets of already committed transactions in order. If the delivered writeset conflicts with other writeset included in the log, then the transaction being certified is aborted and otherwise it will commit. Thus, it is only needed to broadcast (using the total-order facility) [CHO 01] one message and keep a log, as part of the replication protocol.

In [MEN 06], it is formally proved that the isolation level provided by these protocols is Generalized Snapshot Isolation (GSI), as stated in [ELN 05]. The GSI isolation level allows a transaction in the replicated system to read an older snapshot, although,

in contrast with SI, this snapshot may not be the latest snapshot of the database system. Besides, it is also formally proved in [MEN 06] that the SI level cannot be achieved in a replicated environment without blocking transactions at their beginning. This is necessary to guarantee that a transaction sees the latest installed version of the replicated items. Another important aspect to consider is that a dynamic rule (stated in [ELN 05]) permits transactions to be executed in a serializable way over GSI level. Thus, the certification mechanism that provides GSI level shows a better performance compared to SI, since it is not necessary to obtain the latest version of the system. However, transaction aborts may be greater with GSI in conflicting environments and data freshness may be compromised too. Moreover, to provide serializable executions following the mentioned dynamic rule, certification-based algorithms must propagate transaction readsets, what is actually prohibitive. To a lesser extent, the necessity of a garbage collector in these protocols implies some additional overhead, since they must keep track of their certification log to avoid its boundless growing.

In this paper, we propose a database replication protocol for a middleware architecture, called *Mid-Rep*. Most existing protocols are only able to provide a single isolation level, in particular, GSI when the database replicas provide SI level. However, our replication protocol offers a greater flexibility to applications, providing different isolation levels to transactions: GSI, SI and serializable (SER). *Mid-Rep* is a weak voting [WIE 05] replication protocol which follows the eager-update-everywhere strategy. Thus, transactions are locally executed and then changes are propagated, following a ROWAA approach. All changes performed in the database are grouped in a writeset and delivered to the rest of the sites using a total order broadcast delivery, as in the certification mechanism. After its successful application, a reliable delivery of a commit or abort message from its master site will decide whether the transaction must commit or however abort. Our protocol does not need the use of certification and hence there is no need of using a garbage collector. Moreover, it is not necessary to propagate the readsets to provide serial execution, as needed when using certification. Current certification protocols are only able to provide GSI level. However, our *Mid-Rep* protocol includes a mechanism that, although it blocks the beginning of transaction execution, allows to additionally provide SI level by sending a start message total ordered with regard to the transaction writesets. In this way, we achieve our aim to propose a single replication protocol to provide different isolation levels according to the application requirements.

The rest of this paper is organized as follows: The system model is introduced in Section 2. The formalization of our protocol and a brief outline of its correctness proof is presented in Section 3. In Section 4, we propose some enhancements for improving protocol performance. Finally, conclusions end the paper.

2. System Model and Definitions

For our proposal, we have taken the advantage from our previous works [MUÑ 06] and other middleware architectures providing database replication [LIN 05]. Thus, a

weak voting replication protocol, *Mid-Rep*, is proposed taking advantage of the capabilities provided by a middleware architecture called MADIS [MUÑ 06]. For the sake of the explanation of the replication protocol, an abstraction of the MADIS middleware architecture is presented in this Section. In the following, we highlight different aspects dealing with the design of the system and its operation.

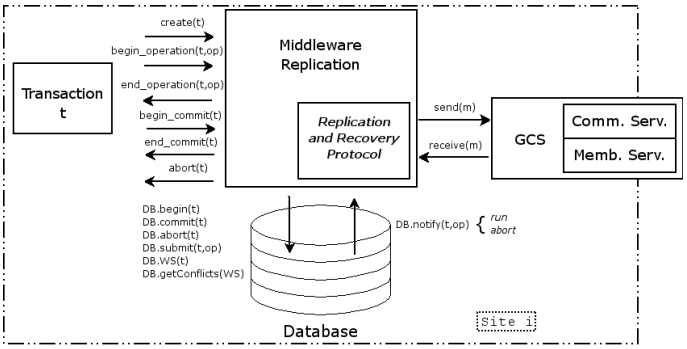

Figure 1. *Main components of the system*

The replicated system is composed of N sites communicating by message exchange. We assume a full replication system, i.e. each site includes an instance of a DBMS which contains a copy of the entire database. Users and applications submit transactions to the system. The middleware forwards them to the respectively nearest (local) site for their execution. The replication protocol in each replica coordinates the execution of the transactions among different sites to ensure the required isolation level for the transactions. The actions shown with arrows in Figure 1 describe how components interact with each other. Actions may easily be ported to the particular communication primitives and DBMS JDBC-like operations.

Communication System. Communication among sites is mainly based on the services provided by a Group Communication System (GCS) [CHO 01]. Basically, a GCS consists of a membership and a communication service [BAR 04]. The *membership service* monitors the set of participating sites and provides them with consistent notifications in case of failures, either real or suspected. Note that, although we consider the possibility of system failures, we are not going to detail in this work the recovery algorithm, for sake of space lack. The *communication service* supports different messaging services that provide several message delivery guarantees. A reliable broadcast primitive ($R\_broadcast$) ensures that messages are always delivered to correct processes despite failures. It also provides a total order broadcast delivery ($TO\_broadcast$) that guarantees all sites deliver messages in the very same order. Each site k has two input buffers for storing messages depending on their delivery guarantees: one for the reliable broadcast messages ($R\_channel_k$) and another for the total order broadcast messages ($TO\_channel_k$). Therefore, broadcasting a message will imply filling the corresponding buffer in all destinations.

Database. We assume a DBMS ensuring ACID transactions and complying with the SI level. The DBMS, as it is depicted in Figure 1 gives to the middleware some common actions. $DB.begin(t)$ begins a transaction t. $DB.submit(t, op)$, where op represents a set of SQL statements, submits an operation (denoted op) in the context of the given transaction t. After a SQL statement submission, the $DB.notify(t, op)$ informs about the successful completion of an operation (run); or, its rollback ($abort$) due to DBMS internals (e.g. deadlock resolution, enforcing SI level as the *first-updater-wins* rule or *first-committer-wins* rule determines, etc). As a remark, we also assume that after the successful completion of a submitted operation by a transaction, it can be committed at any time. In other words, a transaction may be unilaterally aborted by the DBMS only while it is performing a submitted operation. Finally, a transaction ends either by committing, $DB.commit(t)$, or rolling back, $DB.abort(t)$. We have added two additional functions that DBMSs do not provide by default, but may be built by standard database mechanisms [JIM 02, MUÑ 06]. The action $DB.WS(t)$ retrieves the transaction writeset, i.e. the set of pairs $\langle object\,identifier, value\rangle$ for the objects written by the transaction t. In a similar way, the function $DB.getConflicts(WS(t))$ provides the set of conflicting transactions between a writeset and current active ones.

Transaction Execution. Different transactions may be created in the replicated system. Each transaction t has a unique identifier that contains the information about the site which was firstly created in ($t.site$), called its *transaction master site*. This field is used to know whether it is a local or a remote transaction. Transactions created are locally executed at its master site and then interacts via the replication protocol with the other replicas when the application wishes to commit the transaction, following a ROWAA strategy. Thus, remote transactions containing the writeset of the transaction are executed in the rest of available sites of the system. A transaction also contains information about its isolation level ($t.mode$). Each transaction can select an isolation level (GSI, SI or SER), depending on its requirements, at the beginning of its execution. Our protocol is able to provide GSI level by default, given that transactions are atomically committed at all sites and their commit is totally ordered [MEN 06]. In order to obtain a serializable level, all the read operations for SER transactions are parsed to turn them into "`SELECT FOR UPDATE`" statements. This makes possible to see read-write conflicts as they were write-write, from the beginning of a transaction till its commit time. Thus, since this satisfies *dynamic serializability condition* (DSC) [ELN 05], a serializable level is achieved. It is important to remark that the proposed protocol only sends the actual writeset, without including the readset in the SER mode, to the rest of the sites. The price to pay for avoiding the readset propagation in the SER mode is to wait for the decision message, i.e. it needs a weak voting mechanism based on two message rounds: a total order message round with the writesets and another reliable message round with the final decision to commit or abort. This weak voting mechanism also avoids the use of a *garbage collector* since it is not necessary to keep a log with the writesets of transactions that committed previously.

A SI transaction isolation level is achieved by using *start* points in the transactions. These *start* points guarantee that, when a transaction begins its execution, it has seen all the changes applied in the system before that point. Thus, to obtain SI, the *Mid-Rep*

protocol broadcasts a *start* message (using a total order primitive) and the transaction will remain blocked until this message is delivered. Otherwise, in GSI or SER, the transaction starts straight away its reading and writing phase. The rest of the protocol has a similar operation for all the supported isolation levels and it is summarized in the following. During the commit phase, *Mid-Rep* total-order broadcasts the writeset to all available replicas. Upon delivery of this message at a replica it will block all write operations performed in the replica until the conflict detection and the writeset application are done. Moreover, no other writeset could be applied until the previous writeset application is finished. The writeset delivery at the master node will broadcast a $commit$ message (if it was not previously aborted) using a reliable broadcast service. The delivery of this message will commit the application of the writeset at the rest of replicas. In the following Section we describe in detail the protocol operation.

3. Replication Protocol Description

The *Mid-Rep* protocol, presented in this paper, is modelled as a state transition system (introduced in Figures 2 and 3). It includes a set of state variables and actions, each one of them subscripted with the node identifier where they are considered. State variables include their domains and an initial value for each variable. The value of the state variables defines the system state. Each action in the state transition system has an enabling condition (precondition, *pre* in Figure 3), a logic predicate over the state variables. An action can be executed only if its precondition is enabled, i.e. if its predicate is evaluated to *true* on the current state. The effects of an action (*eff* in Figure 3) is a sequential program that atomically modifies the state variables; hence, new actions may become enabled while others become disabled respectively. Weak fairness is assumed for the execution of each action, i.e. if an action is continuously enabled then it will be eventually executed. Although the state transition system seems a static structure, it defines the algorithm's execution flow. This will be easy to understand after the explanation given in this Section. Without generalization loss, we assume a failure free environment throughout the protocol description. Figure 2 shows the protocol signature, which is the set of possible actions it may ever execute. It contains also the definition of the states variables of the transition system and their corresponding initial values. In this Figure, T represents the set of possible transactions, M the set of messages that can be exchanged and OP the set of operations that can be submitted to the database. Figure 3 describes the set of possible actions, detailing their preconditions and effects. We explain such algorithm on the sequel.

A transaction t can start its execution at any site k, which will be considered as its master site (it is a *local* transaction at this site) at any time, since $status_k(t) =$ idle is the initial value for a transaction state. It invokes the $create_k(t)$ action, where transaction is created in the local database replica and its status is set to $active$ to allow operations to be submitted. If a CSI level has been established for the transaction, a $start$ message will be total order broadcast to all the replicas. The transaction will be blocked ($status_k(t) =$ tostart), preventing new operations from being submitted

Signature:
$\{\forall\, k \in N, t \in T, m \in M, op \subseteq OP\colon \mathbf{create}_k(t), \mathbf{discard}_k(t,m), \mathbf{begin\_operation}_k(t,op),$
$\mathbf{end\_operation}_k(t,op), \mathbf{begin\_commit}_k(t), \mathbf{end\_commit}_k(t), \mathbf{receive\_commit}_k(t),$
$\mathbf{receive\_start}_k(t), \mathbf{receive\_WS}_k(t), \mathbf{local\_abort}_k(t), \mathbf{receive\_abort}_k(t)\}$
States:
$\forall\, k \in N, t \in T\colon status_k(t) \in \{idle, tostart, active, blocked, pre\_commit, await,$
$tocommit, toabort, committed, aborted\}$, initially $status_k(t) =$ idle
$\forall\, k \in N\colon TO\_channel_k \subseteq \{m\colon m = \langle start, t\rangle or m = \langle ws, t\rangle \forall t \in T\}$, initially $TO\_channel_k = \emptyset$
$\forall\, k \in N\colon R\_channel_k \subseteq \{m\colon m = \langle commit, t\rangle or m = \langle abort, t\rangle \forall t \in T\}$, initially $R\_channel_k = \emptyset$
$\forall\, k \in N\colon local_k$: **boolean**, initially $local_k =$ false
$\forall\, k \in N\colon ws\_run_k$: **boolean**, initially $ws\_run_k =$ false

Figure 2. *Signature and states for the state transition system of the* Mid-Rep *protocol*

to the local database until the reception of the start message, in $receive\_start_k(t)$, in order to guarantee that transaction is going to see the latest database snapshot.

The transaction creation action is followed by a sequence of pairs of the actions $begin\_operation_k(t, op)$ and $end\_operation_k(t, op)$. Each pair corresponds to a successful completion of a SQL statement. The invocation of a $begin\_operation$ submits the SQL statement to the database ($DB_k.submit(t, op)$) and sets its status to $blocked$. This is allowed provided that transaction is not blocked waiting for the $start$ message ($status_k(t) =$ active). Besides, we have to consider that a local transaction may conflict with a writeset application of a remote transaction once executed the $receive\_WS_k(t)$ action. Writeset modifications must be applied atomically in the database, without allowing other local or remote transactions to conflict with the modified values, to prevent consistency problems and also distributed and local deadlock situations. This may happen when a write operation is submitted to the database ($type(op) =$ WRITE). Also, when working in a serializable environment ($t.mode =$ SER), read operations must be considered as write operations in order to guarantee the isolation level. In both cases, an operation will be submitted only if there is no writeset being applied in the database ($ws\_run_k =$ false).

After the submission of an operation to the database, the transaction may be aborted by the DBMS replica ($local\_abort_k(t, op)$). This is only possible for local transactions. The causes of abortion are mainly related to the enforcement of the SI level or to a local deadlock. The $end\_operation$ action will be eventually invoked after the operation is successfully completed in the database and the local transaction may submit a new statement. Once the transaction is done, it requests its commitment by means of the $begin\_commit_k(t)$ action, as $status =$ active. In this action, the transaction writeset is collected from the database ($DB_k.WS(t)$). If the transaction is a read only transaction ($WS = \emptyset$) the transaction will commit immediately. Otherwise, the replication protocol broadcast a *writeset* message to all the replicas using the total order delivery and change the transaction status to *pre_commit*.

Writeset message ($\langle ws, t\rangle$) reception at the master site of the transaction ($t.site = k$), where transaction should have $status_k(t) =$ pre_commit, leads to the execution of the $receive\_WS_k(t)$ action in that site. In order to enable this action, it is also necessary that there is no other writeset being applied in the database ($\neg ws\_run_k$)

Transitions:

$\mathbf{create}_k(t)$ // $t.site = k$ //
pre $\equiv status_k(t) =$ idle
eff $\equiv$ **if** $t.mode =$ SI **then**
$status_k(t) \leftarrow tostart$
$TO\_broadcast(\langle start, t\rangle)$
else
$status_k(t) \leftarrow$ active
$DB_k.begin(t)$

$\mathbf{begin\_operation}_k(t, op)$ // $t.site = k$ //
pre $\equiv status_k(t) =$ active $\wedge\ \neg(ws\_run_k \wedge$
$(type(op) =$ WRITE $\vee\ t.mode =$ SER)).
eff $\equiv status_k(t) \leftarrow$ blocked
$DB_k.submit(t, op)$

$\mathbf{end\_operation}_k(t, op)$ // $t.site = k$ //
pre $\equiv status_k(t) =$ blocked $\wedge$
$DB_k.notify(t, op) =$ run.
eff $\equiv status_k(t) \leftarrow$ active

$\mathbf{end\_operation}_k(t, t.ws)$ // $t.site \neq k$ //
pre $\equiv DB_k.notify(t, ws) =$ run.
eff $\equiv$ **if** $status_k(t) =$ blocked **then**
$status_k(t) \leftarrow$ await
else if $status_k(t) =$ tocommit **then**
$status_k(t) \leftarrow$ committed
$DB_k.commit(t)$
$ws\_run_k \leftarrow$ false
else if $status_k(t) =$ toabort **then**
$status_k(t) \leftarrow$ aborted
$DB_k.abort(t)$
$ws\_run_k \leftarrow$ false

$\mathbf{begin\_commit}_k(t)$ // $t.site = k$ //
pre $\equiv status_k(t) =$ active
eff $\equiv t.WS \leftarrow DB_k.WS(t)$
if $t.WS = \emptyset$ **then**
$status_k(t) \leftarrow$ committed
$DB_k.commit(t)$
else
$status_k(t) \leftarrow$ pre_commit
$TO\_broadcast(\langle ws, t\rangle)$

$\mathbf{end\_commit}_k(t)$ // $t.site = k$ //
pre $\equiv status_k(t) =$ pre_commit $\wedge$
$\langle commit, t\rangle$ first in $R\_channel_k$
eff $\equiv remove(m)$ from $R\_channel_k$
$status_k(t) \leftarrow$ committed
$DB_k.commit(t)$
$local_k \leftarrow$ false

$\mathbf{local\_abort}_k(t, op)$ // $t.site = k$ //
pre $\equiv status_k(t) =$ blocked $\wedge$
$DB_k.notify(t, op) =$ abort.
eff $\equiv status_k(t) \leftarrow$ aborted

$\mathbf{discard}_k(t, m)$
pre $\equiv status_k(t) =$ aborted $\wedge\ m \in$ any $channel_k$
eff $\equiv remove(m)$ from corresponding $channel_k$

$\mathbf{receive\_start}_k(t)$
pre $\equiv \wedge\langle start, t\rangle$ first in $TO\_channel_k$
$\wedge \neg ws\_run_k\ \wedge\ \neg local_k$
$status_k(t) =$ tostart.
eff $\equiv remove(m)$ from $TO\_channel_k$
if $t.site = k$ **then**
$DB_k.begin(t)$
$status_k(t) \leftarrow blocked$
$DB_k.submit(t, first\_op)$

$\mathbf{receive\_WS}_k(t)$ // $t.site = k$ //
pre $\equiv\langle ws, t\rangle$ first in $TO\_channel_k$
$\wedge \neg ws\_run_k\ \wedge\ \neg local_k$
$\wedge\ status_k(t) =$ pre_commit.
eff $\equiv remove(m)$ from $TO\_channel_k$
$status_k(t) \leftarrow$ tocommit
$local_k \leftarrow$ true
$R\_broadcast(\langle commit, t\rangle)$

$\mathbf{receive\_WS}_k(t)$ // $t.site \neq k$ //
pre $\equiv\langle ws, t\rangle$ first in $TO\_channel_k$
$\wedge \neg ws\_run_k\ \wedge\ \neg local_k.$
eff $\equiv remove(m)$ from $TO\_channel_k$
if $status_k(t) =$ toabort **then**
$status_k(t) \leftarrow$ aborted
else
$A \leftarrow getConflicts(t.ws)$
for each t' **in** A
$status_k(t') \leftarrow$ aborted
$DB_k.abort(t')$
if $status_k(t') =$ pre_commit **then**
$R\_broadcast(\langle abort, t\rangle)$
$DB_k.begin(t)$
$status_k(t) \leftarrow$ blocked
$DB_k.submit(t, t.ws)$
$ws\_run_k \leftarrow$ true

$\mathbf{receive\_commit}_k(t)$ // $t.site \neq k$ //
pre $\equiv\langle commit, t\rangle$ first in $R\_channel_k.$
eff $\equiv remove(m)$ from $R\_channel_k$
if $status_k(t) =$ await **then**
$status_k(t) \leftarrow$ committed
$DB_k.commit(t)$
$ws\_run_k \leftarrow$ false
else $status_k(t) \leftarrow$ tocommit

$\mathbf{receive\_abort}_k(t)$ // $t.site \neq k$ //
pre $\equiv\langle abort, t\rangle$ first in $R\_channel_k.$
eff $\equiv remove(m)$ from $R\_channel_k$
if $status_k(t) =$ await **then**
$status_k(t) \leftarrow$ aborted
$DB_k.abort(t)$
$ws\_run_k \leftarrow$ false
else $status_k(t) =$ toabort

Figure 3. *Transitions for the state transition system of the* Mid-Rep *protocol*

and there is no other local transaction waiting to commit ($\neg local_k$) as well. This action will broadcast a *commit* message with a reliable service ($R\_broadcast$) and sets the transaction status to $tocommit$ in order to emphasize that this transaction is about to commit. Beside this, the variable $local_k$ is set to *true* in order to point that there is a transaction waiting for its commit message to finally commit into the local database. The main aim of this *commit* message is related to recovery issues, but are not explained in this paper for sake of brevity. The reception of this message at the transaction master site will finally commit the transaction in the local database replica and will set the variable $local_k$ to $true$, allowing other transactions to commit.

In the other sites ($t.site \neq k$), the reception of a writeset message ($\langle ws, t \rangle$) will create a remote transaction in that site if the $receive\_WS_k(t)$ action becomes enabled. In order to guarantee the global atomicity of a transaction, it is a must that a remote transaction, not yet submitted to execution, never aborts a remote transaction already submitted to the database or a local transaction waiting to its commit message. For that reason, the $receive\_WS_k(t)$ action requires that no other writeset is being applied in the database ($\neg ws\_run_k$) and also that no local transaction is waiting ($\neg local_k$) for commit. Unless the transaction corresponding to the writeset had been aborted previously by its master site, the $receive\_WS_k(t)$ action aborts all the local transactions conflicting with the received writeset ($DB_k.getConflicts(t.ws)$). This is necessary to prevent remote transactions from becoming blocked by a conflicting local transaction. Afterward, it applies the writeset in the database ($DB_k.submit(t, t.ws)$) and sets the variable ws_run_k to $true$ until writeset application ends (either with the commitment or the abortion of the remote transaction). It is important to note that aborting all local conflicting transactions before the execution of a remote transaction has several consequences. If one of the conflicting local transactions is in the *pre_commit* state, it is necessary to broadcast an *abort* message (it will enable the $receive\_abort_k(t)$ in the rest of sites) to abort its remote transactions.

Once the writeset is successfully applied, the $end\_operation_k(t, t.ws)$ for that site ($t.site \neq k$) becomes enabled. If *commit* or *abort* message has been received through the corresponding action ($receive\_commit_k(t)$ or $receive\_abort_k(t)$), transaction status will have been modified and it will be waiting for commit ($tocommit$) or abort ($toabort$) respectively. Thus, remote transaction will simply commit or abort locally in that replica and in both cases the writeset application process will have finished ($ws\_run_k \leftarrow false$) and other writesets may be applied into the database. Otherwise, the transaction has to wait for the master site decision and thus it changes it status to *await*. Hence, once successfully applied the writeset, the reception of the commit ($receive\_commit$) or abort message ($receive\_abort$) will commit or abort the remote transaction. Note that reliable broadcast latency is lower than total order one and that applying a writeset takes some time. Hence, a reliable message with the final decision of commit or abort may be delivered before the reception of the writeset message, which is broadcast in total order as our protocol states, or before its application. This implies that if the final decision arrives before the writeset has been applied, it will be necessary to indicate that it has to commit or abort straight after the application of the writeset by changing its status to the corresponding one (*tocommit* or *toabort*).

Proof of Correctness

In the following lines, we are going to outline the correctness proof of our algorithm. As proved in [MEN 06], a ROWA replication protocol generates One Copy Schedules of the transactions that verify the GSI level, providing that the following conditions are fulfilled: (a) *(SI replicas)* each database replica provides SI isolation level; (b) *(atomicity)* each transaction submitted to the system will be either committed at all sites, or will be aborted at all sites if it is aborted in one site; and (c) *(total order of committed transactions)* transactions that have been committed follows the same total order at all sites.

The proposed replication protocol is based on the use of database replicas that provide SI isolation level, therefore the *SI replicas* condition is fulfilled. The *atomicity* is also fulfilled by the following reasoning: let us consider that a remote transaction t, with a writeset $t.ws$, aborts somewhere. Given that the protocol does not allow processing new writeset messages (ws_run_k = true), that the conflicting local transactions have been previously aborted ($\forall t' \in DB_k.getConflicts(t.ws) : status_k(t') =$ aborted) and that blocks between writeset $t.ws$ application and local operations are prevented (precondition of $begin\_operation()$), then the only way a transaction has aborted is through the reception of the $\langle abort, t\rangle$ message from the transaction t master site. Therefore, the transaction master site had to first total order broadcast the $t.ws$ to all the sites and afterward the $\langle abort, t\rangle$ message. By the reliable property of $TO\_broadcast$ and $R\_broadcast$, the $status_k(t)$ will finally become $toabort$ or $aborted$. The same argument is valid for a transaction t that commits through the corresponding actions. Necessarily, in the transaction master site its status is to_commit or $committed$ and for the rest of the sites is either to_commit or $committed$. Given that each writeset is delivered in total order and that only the master site is allowed to broadcast the commit or abort decision, the actions that reach the finalization of the transactions will be enabled and the final decision reached. Finally, the *total order of committed transactions* condition is also verified given that the writesets are delivered in total order at all the sites and they are applied in sequence after having reached the final decision of committing or aborting and hence all committed transactions have committed in the same order. Therefore, our protocol generates One Copy Schedules of the transactions that verify GSI. Given that the SER and SI mode are restrictions to GSI, is can be proved that the restrictions are verified for the said transaction types.

4. Performance Optimizations

In essence, the protocol *Mid-Rep* presented in this paper is pessimistic. On one hand, writesets received from a remote site are applied one after another in each database replica. On the other hand, this protocol avoids that the remote writesets become blocked by local transactions, disabling for that purpose potential conflicting local transactions' access to the database. The main objective of the proposed protocol is simply to show that it is possible to achieve the three isolation levels considered (GSI, SI and SER) with the very same protocol. However, due to its pessimistic na-

ture, the expected performance is quite poor. Nevertheless, several optimizations can be taken into account in order to improve significantly its performance. Basically, it is necessary to increase first the concurrency between writeset applications and local transactions, and also among the writeset applications themselves.

Our protocol includes a deadlock prevention schema in order to avoid that transactions become blocked in the local database replicas. An initial improvement to be considered is the replacement of this deadlock prevention mechanism with a detection mechanism as the one stated in [MUÑ 06] that has been successfully applied in several works with satisfying results.This mechanism is based on a block detection mechanism that uses the concurrency control support of the underlying DBMS. Thereby, the middleware is enabled to provide a row-level control (as opposed to the usual coarse-grained table control), while all transactions (even those associated to remote writesets) are subject to the underlying concurrency control support. The block detection mechanism looks periodically for blocked transactions in the DBMS metadata (e.g., in the pg_locks view of the PostgreSQL system catalogue). It returns a set of pairs consisting of the identifiers of the blocked and blocking transactions and the replication protocol will decide which one must abort. This detection mechanism provides several advantages that increase the protocol performance. On one hand, this mechanism allows remote writesets to be directly submitted to the database replicas, without the necessity of checking conflicts with existing transactions. This reduces the protocol overhead, since unnecessary calls to database primitives are avoided when there is no conflicting local transaction. Beside this, this mechanism of deadlock detection also allows local transactions to be concurrently executed with writesets applications. This implies a higher degree of concurrency and therefore a better performance. If a transaction associated to a remote writeset is aborted, then it will be necessary simply to reattempt to apply the writeset in the database until succeed.

Most of the applications present a low conflict rate. In this case, it is not necessary to apply writesets in sequence since we can schedule several transactions in the database concurrently. At the delivery of a remote writeset, protocol will check if there is any writeset scheduled in the database that conflicts with the incoming one. If no conflict is detected, replication protocol will permit scheduling the writeset application in the database. Otherwise, it must wait on the commitment of the writesets scheduled in the database. Note that scheduled transactions must commit in the same order of the total order delivery, so as to guarantee that isolation levels are fulfilled.

Apart from increasing system concurrency, we can reduce the time a transaction must wait in order to obtain the latest snapshot in the SI level. We can consider an optimistic approach for SI transactions, in which it is not necessary to wait for the reception of the start message in order to be able to submit operations to database. Different writesets may be applied in the database replicas until the reception of the start message for a given transaction. Only if none of the applied writesets is going to conflict when transaction tries to commit, SI level will be achieved. Thus, depending on the number of writesets applied during that time, there will be a certain probability of having achieved the SI level, what can be enough for some applications.

5. Conclusions

This paper has presented a single middleware database replication protocol able to support different degrees of isolation (SI, GSI and serializable) on top of DBMSs supporting SI. This provides a great flexibility in the application development process. Its main advantage is that it does not need a certification process but a weak voting one. This fact represents a novelty over SI replicas, since it usually reduces the abortion rate and avoids the drawbacks certification presents, such as keeping track of its log. Since the proposed protocol is rather pessimistic, we have also pointed out some optimizations for increasing its performance.

6. References

[BAR 04] BARTOLI A., "Implementing a Replicated Service with Group Communication.", *Journal of Systems Architecture*, vol. 50, num. 8, 2004, p. 493-519, Elsevier.

[BER 87] BERNSTEIN P. A., HADZILACOS V., GOODMAN N., *Concurrency Control and Recovery in Database Systems*, Addison Wesley, 1987.

[BER 95] BERENSON H., BERNSTEIN P. A., GRAY J., MELTON J., O'NEIL E. J., O'NEIL P. E., "A Critique of ANSI SQL Isolation Levels.", *SIGMOD Conference*, 1995, p. 1-10.

[CHO 01] CHOCKLER G., KEIDAR I., VITENBERG R., "Group communication specifications: a comprehensive study.", *ACM Comput. Surv.*, vol. 33, num. 4, 2001, p. 427-469.

[ELN 05] ELNIKETY S., PEDONE F., ZWAENOPOEL W., "Database Replication Using Generalized Snapshot Isolation.", *SRDS*, IEEE-CS, 2005.

[GRA 96] GRAY J., HELLAND P., O'NEIL P. E., SHASHA D., "The Dangers of Replication and a Solution.", JAGADISH H. V., MUMICK I. S., Eds., *SIGMOD Conference*, ACM Press, 1996, p. 173-182.

[IRÚ 05] IRÚN L., DECKER H., DE JUAN R., CASTRO F., ARMENDÁRIZ J. E., MUÑOZ F. D., "MADIS: A Slim Middleware for Database Replication.", *Euro-Par*, vol. 3648 of *LNCS*, Springer, 2005, p. 349-359.

[JIM 02] JIMÉNEZ-PERIS R., PATIÑO-MARTÍNEZ M., KEMME B., ALONSO G., "Improving the Scalability of Fault-Tolerant Database Clusters.", *ICDCS*, 2002, p. 477-484.

[LIN 05] LIN Y., KEMME B., PATIÑO-MARTÍNEZ M., JIMÉNEZ-PERIS R., "Middleware based Data Replication providing Snapshot Isolation.", *SIGMOD Conference*, 2005.

[MEN 06] DE MENDÍVIL J. R. G., ARMENDÁRIZ J. E., GARITAGOITIA J. R., IRÚN L., MUÑOZ F. D., "Non-blocking ROWA Protocols Implement GSI Using SI Replicas", report num. ITI-ITE-06/04, 2006, ITI.

[MUÑ 06] MUÑOZ F. D., PLA J., RUIZ M. I., IRÚN L., DECKER H., ARMENDÁRIZ J. E., DE MENDÍVIL J. R. G., "Managing Transaction Conflicts in Middleware-Based Database Replication Architectures", *SRDS*, IEEE-CS, 2006, p. 401-410.

[PAT 05] PATIÑO-MARTÍNEZ M., JIMÉNEZ-PERIS R., KEMME B., ALONSO G., "MIDDLE-R: Consistent database replication at the middleware level.", *ACM Trans. Comput. Syst.*, vol. 23, num. 4, 2005, p. 375-423.

[WIE 05] WIESMANN M., SCHIPER A., "Comparison of Database Replication Techniques Based on Total Order Broadcast.", *IEEE TKDE.*, vol. 17, num. 4, 2005, p. 551-566.