Weak Voting Database Replication Protocols Providing Different Isolation Levels

J.R. Juárez, J.E. Armendáriz, J.R. González de Mendívil, J.R. Garitagoitia, F.D. Muñoz

Instituto Tecnológico de Informática
Univ. Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia (Spain)

{jr.juarez,enrique.armendariz,mendivil,joserra}@unavarra.es, fmunyoz@iti.upv.es

Technical Report TR-ITI-ITE-07/16

Weak Voting Database Replication Protocols Providing Different Isolation Levels

J.R. Juárez, J.E. Armendáriz, J.R. González de Mendívil, J.R. Garitagoitia, F.D. Muñoz

Instituto Tecnológico de Informática
Univ. Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia (Spain)

Technical Report TR-ITI-ITE-07/16

e-mail: {jr.juarez,enrique.armendariz,mendivil,joserra}@unavarra.es,
fmunyoz@iti.upv.es

July 17, 2007

Abstract

Recently, several works have taken advantage of a database isolation notion suitable for replicated approaches, called Generalized Snapshot Isolation, that provides greater performance since read-only transactions are never blocked nor cause update transactions to block or abort. However, this concept has not been formally described for replicated environments where a logical copy of the system must be considered in order to study its behavior. In this work, we study formally the conditions that guarantee the one-copy equivalent Generalized Snapshot Isolation level in a database replicated systems using Snapshot Isolation replicas. Some recent protocols based on Snapshot Isolation replicas use a certifying technique, but this technique requires sending the readset in order to achieve a serializable level, what is prohibitive. Thus, we propose a basic replication protocol, and some possible enhancements for this protocol, based on a weak voting technique over replicas providing snapshot isolation level. This protocol and its enhanced versions are able to provide different isolation levels to transactions submitted by a database user only using one protocol. Each transaction submitted to the database can be executed ensuring a given isolation level, what provides a great flexibility to applications that may demand different isolation levels for their transactions.

1 Introduction

Database replication over a network such as the Internet is an effective way to cope with site failures. It increases the system availability and performance by storing copies of the same data at multiple sites and distributing clients among all available replicas. However, all these advantages do not come for free since data consistency is somehow sacrificed. It greatly depends on the replication policy followed in the system. Several correctness criteria [1, 2, 3, 4, 5] have been defined for replicated databases. The strongest, and firstly introduced, is the One-Copy-Serializable (1CS). It stands for the natural extension of a centralized serializable Database Management System (DBMS) scheduling of transactions to the replicated case. The 1C level definition states that the interleaved execution of transactions in a replicated case must be equivalent to their execution in a centralized serializable database.

Database replication techniques have been classified according to several parameters [6]. Regarding to who performs the updates, the *primary copy* [7] requires all updates to be performed on one copy and then changes are propagated to the rest of sites; whilst *update everywhere* [8] allows to perform updates at any copy but makes coordination more complex [9]. Considering the instant when a transaction update

propagation takes place, we can distinguish between *eager* [10] and *lazy* [11, 12] protocols. In eager replication schemes, updates are propagated inside the context of the transaction. On the other hand, lazy replication schemes propagate changes to the rest of available replicas after the commitment of the transaction. Data consistency is straightly forward by eager replication techniques although it requires extra messages. On the contrary, data copies may diverge on lazy schemes and, as there is no automatic way to reverse committed replica updates, a program or a person must reconcile conflicting transactions.

In a replicated database system, all replicas may contain a full copy of the database, i.e. full replication, or instead each data item may be stored in a different subset of the set of replicas, i.e. partial replication. As shown in [6], the most effective way to achieve database replication in a fully replicated architecture is the Read One Write All Available (ROWAA) algorithm, that combines eager techniques with the update-everywhere approach. Transactions are firstly executed at their delegate replicas and the interaction with the rest of replicas is started when they request the commit. At commit time, updates are grouped (denoted as the writeset of a transaction) and sent to the rest of available replicas.

The implementation of database replication systems has two main approaches. Originally, the DBMS-core was modified so as to include some communication support and means to deal with transactions coming from remote sites [13, 14, 15, 1, 10, 16, 17, 8, 18]. However, this solution is highly dependent on the DBMS core used and it is not portable among different DBMS vendors. The alternative approach is to deploy a middleware architecture that creates an intermediate layer that features data consistency, being transparent to the final users, isolating the DBMS details from the replication management [19, 20, 21, 22, 23, 3, 24, 25, 26, 27, 28]. 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.

Middleware replication is useful to integrate new replication functionalities (availability, fault-tolerance, etc.) for applications dealing with non-replicated database systems when it is not possible to modify their core. However, middleware solutions often lack scalability and exhibit a number of consistency and performance issues. The main reason is that in most cases the middleware has to handle the database as a black box, re-implementing many features provided by the DBMS, and hence, these approaches cannot take advantage of many optimizations implemented in the database kernel. Besides, the database schema has to be extended with standard database features, such as functions, triggers, stored procedures, etc. [23], in order to manage additional metadata that eases replication. This alternative introduces an overhead that penalizes performance but permits to get rid of DBMSs' dependencies. In this work we took the advantage from our previous works [29] and other middleware architectures providing database replication [3].

Database replication based on group communication systems has been proposed as an efficient and resilient solution for data replication. Protocols based on group communication typically rely on a broadcast primitive called *atomic* [30] or *total order* [31] broadcast. This primitive ensures that messages are delivered reliably and in the same order on all replicas. This approach ensures consistency and increases availability by relying on the communication properties assured by the atomic broadcast primitive. The update propagation using the total order facility avoids the drawbacks of the 2PC protocols [1]. On one hand, it makes distributed deadlock resolution easier. On the other hand, combined with eager update-everywhere protocols based on constant interation [32], i.e. a constant number of messages are exchanged between sites for a given transaction, it provides a better performance.

A comparison of database replication techniques based on total order broadcast is introduced in [33]. This work compares different techniques that offer the same consistency criterion (one-copy serializability) under the assumption that all DBMSs were based on a two phase locking (2PL) [1] implementation. From those presented there, the weak-voting [17] and certification-based [34] techniques, which are specifically built for database replication, present the best performance.

In certification protocols, writesets are total order broadcast to all the replicas and at their delivery they are compared with the ones contained in a log that stores the writesets of already committed transactions in order. If a delivered writeset conflicts with any 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) one message and keep a log, as part of the replication protocol. Nevertheless, to provide serializable executions 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.

These drawbacks are avoided with the weak-voting protocols, that do not use certification and hence there is no need of using a garbage collector. For the same reason, it is not necessary to propagate the readsets to provide serial execution, as needed when using certification. Certification based protocols only need a message so that each replica may decide on its own the commitment of a transaction. On the other hand, weak-voting replication protocols require an additional message from the delegate replica to decide about the outcome of a transaction. This implies a slight difference in performance regarding with the certification-based techniques. However, weak voting protocols present a lower abort rate than certification-based ones. In certification algorithms, transactions may stay too long in the log until removed from it when a transaction is known to have committed in all replicas. Therefore, even if transactions are executed sequentially, so that there should be no conflict, conflicts can appear since there may be conflicting writesets pending their removal from the log.

1.1 Motivation

However, it is important to mention that achieving serializability in replicated systems presents a major drawback since it requires DBMSs executing transactions under a strict serializable isolation mode, as 2PL ones do, what involves blocking read operations. Thus, read operations may become blocked by write operations. In the majority of web applications the number of write operations are overwhelmed by the set of read operations performed by a transaction. This makes more attractive the use of DBMSs providing Snapshot Isolation (SI) [35] where read operations never blocked rather than traditional serializable ones. In SI DBMSs, a transaction obtains at the beginning of its execution the latest snapshot of the database, reflecting the writes of all transactions that have committed before the transaction started. At commit time, the database checks that the updates preformed by the transaction do not intersect with the updates done by other transactions since the transaction started (i.e. since it obtained its snapshot); this is often denoted as *first-committer-wins* rule. If there is a non-zero intersection, the transaction will be rolled back; otherwise, it will commit.

More recently, it has been studied how to achieve database replication when DBMS replicas provide SI [36, 2, 3, 29, 18, 5, 37, 38]. From these solutions one can note that it is not straightforward to get the "latest" snapshot version in a distributed setting. In [2] it is extended the notion of conventional SI to Generalized SI (GSI) where transactions are not enforced to "see" the latest snapshot version but an older one. GSI maintains all the interesting properties of conventional SI at the price of a possible increase of the abortion rate if updates were performed in a very *old* snapshot. GSI is the natural extension of SI to a replicated setting where the read operations of a transaction never block in order to obtain One-Copy-GSI (1CGSI) schedulers. Non-blocking ROWAA protocols supporting 1CGSI will give as the snapshot of a transaction the one gotten at its delegate replica.

At commit time, it can be followed a certification process [2] pretty similar to the *first-committer-wins* rule, where a given (or all, for a distributed certification process) replica stores a log of certified transactions just to perform the *first-committer-wins* rule, i.e. the intersection between the writeset of the already delivered transaction and previously, but concurrent to the previous one, certified transactions must be non-empty. All these aspects have been thoroughly discussed in [4, 2] as well as together with the impossibility of deriving a One-Copy-SI (1CSI) without blocking the start of transactions, i.e. just to get the latest snapshot version at the start of the transaction.

Most protocols are only able to provide a single isolation level. However, we propose a database replication protocol for a middleware architecture that offers much more flexibility to applications, providing different isolation levels to transactions: GSI, CSI and serializable (SER). Generally, the different levels featured depend on: the transaction isolation level provided by the underlying DBMS; the ordering of commit operations at all nodes; and, the starting point of transactions [4]. We consider CSI replicas since most database vendors provides this isolation level by default. Our protocol does not need the use of certification, hence there is no need of using a garbage collector. For the same reason, it is not necessary to propagate the readsets to provide serial execution, as needed when using certification. This protocol is a weak voting replication protocol which is, up to our knowledge, the first protocol proposed in this way.

1.2 Contributions and Outline

The contributions of this paper are as follows:

- Formal study of the conditions that guarantee the one-copy equivalent Generalized Snapshot Isolation (1C-GSI) level in database replicated systems.
- Presentation of a formal system model for developing middleware-based replication protocols, using state transition systems in order to describe the operation of replication protocols;
- A brief analysis of the current state of the art in database replication protocols based on snapshot isolation replicas; and
- Introducing a weak voting replication protocol providing different isolation levels over CSI replicas, including some possible optimizations to increase its performance.

The remainder of this paper is organized as follows: Section 2 introduces some preliminaries and the concept of Generalized Snapshot Isolation. Section 3 introduces the conditions for the One Copy Generalized Snapshot Isolation for ROWAA protocols. Section 4 presents a discussion about recent database replication protocol proposals over Snapshot Isolation replicas. Section 5 presents the database system model and necessary definitions. In Section 6, we propose a weak-voting replication protocol providing different isolation levels and some enhancements from its basic operation. Finally, Section 7 presents the conclusions and future research directions.

2 Generalized Snapshot Isolation

ANSI SQL-92 [39, 40] defines several isolation levels in terms of different phenomena: dirty reads, non-repeatable reads and phantoms. These ANSI isolation levels are criticized in [35], since their definitions fail to characterize several popular isolation levels. Besides, an important multiversion isolation type, called *Snapshot Isolation* (SI), is also presented in [35]. In this isolation level, each transaction reads data from a *snapshot* of the committed data as of the time the transaction started. In SI, transactions hold the results of their own writes in local memory store [41], i.e transaction's writes will be reflected in its snapshot. So, if they access data they has written a second time, they will see its own output. When reading from a snapshot, a transaction sees all the updates done by transactions that committed before it started its first operation, whilst writes performed by other transactions that began after its start time, i.e. writes by concurrent transactions, are invisible to the transaction.

The results of a transaction writes are installed when the transaction commits. However, a transaction T_i will successfully commit if and only if there is not a concurrent transaction T_k that has already committed and some of the items written by T_k were also written by T_i. This technique, known as *first-committer-wins*, prevents lost updates [35, 2], since when a transaction commits, its changes become visible to all transactions that began after its commit time. Some database implementations do not follow strictly the first-committer-wins rule, and instead they use a similar one that is known as the *first-updater-wins* rule. The ultimate effect is the same in both of them, i.e. to abort one of the concurrent transactions updating the same data item. The main difference resides in when the validation is performed. Whilst the first-committer-wins rule is validated when the transaction wants to commit, the first-updater-wins rule is enforced by checks performed at the time of updating data items, allowing transactions to be rolled back earlier (not delaying its abortion until its commit time).

SI provides a weaker form of consistency than serializability, but it never requires read-only transactions to be blocked or aborted and they do not cause update transactions to be blocked or aborted, what is an important fact when working with intensive read applications. Moreover, it has been recently proved in [41] that under certain conditions on the workload transactions executing on a database with SI produce serializable histories.

Elnikety et al. define in [2] a new concept called *Generalized Snapshot Isolation* (GSI) level, that extends the SI level definition in a manner suitable for working in replicated environments. In the conventional notion of snapshot isolation, referred to in that paper as *Conventional Snapshot Isolation* (CSI),

each transaction must observe the $latest$ snapshot of the database. Unfortunately, working with the latest snapshot in a distributed setting is not trivial. It has been proved that CSI level cannot be obtained in replicated systems unless blocking protocols are used in order to work with the notion of $latest$ snapshot, what limits its application to distributed database systems. A prove of this impossibility result, initially mentioned in [2], is provided in [4].

However, in contrast to the CSI, the GSI level allows the use of $older$ snapshots of the database, facilitating its replicated implementation. A transaction may receive a snapshot that happened in the system before the time of its first operation (instead of its current snapshot as in CSI). To commit a transaction it is necessary, as in CSI, that no other update operation of recently committed transactions conflicts with its update operations. Thus, a transaction can observe an older snapshot of the database but the write operations of the transaction are still valid update operations for the database at commit time. Many of the desirable properties of CSI remain also in GSI, in particular, read-only transactions never became blocked and neither they cause update transaction to block or abort.

In this Section, we are going to formalize the GSI definition. The GSI level is defined independently of any replication considerations, just as CSI, considering a centralized database system. In order to consider the GSI notion in a replicated environment, it is necessary to work with one-copy equivalent executions. Thus, in Section 3 conditions that can be imposed on a ROWAA protocol in order to obtain One-Copy GSI (1C-GSI) are studied. This will facilitate later the study of the correctness of the replication protocols proposed in this work.

2.1 Preliminaries

From our point of view, histories generated by a given concurrency control providing snapshot-based isolation levels, such as GSI or CSI, may be interpreted as multiversion histories with time restrictions. In fact, isolation level definitions include the time sense implicity and hence it seems that working with transactions' operations and their times is more suitable than using partial orders and graphs. In the following, we define the concept of multiversion history for transactions using the theory provided in [1].

A database (DB) is a collection of data items, which may be concurrently accessed by transactions. A history represents an *overall partial ordering* of the different operations executed concurrently within the *context* of their corresponding transactions. A multiversion history extends the concept of a history by considering that the database items are versioned.

In order to formalize this definition, each transaction submitted to the system is denoted by T_i. A transaction is a sequence of read and write operations on database items ended by a commit or abort operation[1]. Each T_i's write operation on item X is denoted as $W_i(X)$ and a read operation on item X as $R_i(X)$. Finally, C_i and A_i denote the T_i's commit and abort operation respectively. We assume that a transaction does not read an item X after it has written it, and each item is read and written at most once. Avoiding redundant operations simplifies the presentation. The results for this kind of transactions are seamlessly extensible to more general models and thus the replication protocols presented in this work do not consider this restriction. In any case, redundant operations can be removed using local variables in the planification of the transaction [42].

Each version of a data item X contained in the database is denoted by X_i, where the subscript stands for the transaction identifier that installed that version in the DB. The *readset* and *writeset* (denoted by RS_i and WS_i respectively) express the sets of items read (written) by a transaction T_i. Thus, T_i is a *read-only* transaction if $WS_i = \emptyset$ and otherwise it is an *update* transaction.

We assume in our approach that aborted transactions are going to have no effect over generated histories. This is a reasonable assumption since usually a DBMS produces no anomalies when a transaction aborts. Therefore, in the properties studied in this Section we are only going to deal with committed transactions for simplicity's sake. Nevertheless, we will discuss this later in Section 2.4 in a more detailed way.

Let $T = \{T_1, ..., T_n\}$ be a set of *committed* transactions, where the operations of T_i are totally ordered by the order $\prec_{T_i}$. Since aborted transactions are not considered, the last operation of a transaction execution should be the commit operation. In order to process operations from a transaction $T_i \in T$, a multiversion

[1] Without losing rigor, sometimes a transaction denotes also the set of operations that contains.

scheduler must translate T_i's operations on data items into operations on specific versions of those data items. That is, there is a function h that maps each $W_i(X)$ into $W_i(X_i)$, each $R_i(X)$ into $R_i(X_j)$ for some $T_j \in T$ and each C_i just into C_i.

Definition 1. *A Complete Committed Multiversion (CCMV) history over a set of transactions T is a partial order $(H, \prec)$ such that:*

1. *there exists a mapping h such that $H = h(\bigcup_{T_i \in T} T_i)$*
2. *$\prec \supseteq \bigcup_{T_i \in T} \prec_{T_i}$.*
3. *if $R_i(X_j) \in H$, $i \neq j$, then $W_j(X_j) \in H$ and $C_j \prec R_i(X_j)$.*

In the previous Definition 1, condition (1) suggests that each operation submitted by a transaction is mapped into an appropriate multiversion operation. Condition (2) states that the CCMV history preserves all orderings stipulated by transactions. Condition (3) establishes that when a transaction reads a concrete version of a data item, it was written by a transaction that committed before the item was read.

Definition 1 is more specific than the one stated in [1], since the former only includes committed transactions and explicitly indicates that a new version may not be read until the transaction that installed the new version has committed. In the rest of this Section, we use the following conventions: (i) $T = \{T_1, ..., T_n\}$ is the set of committed transactions for every defined history; and (ii) any history is a CCMV history over T.

In general, two histories $(H, \prec)$ and $(H', \prec')$ are *view equivalent* [1] denoted $H \equiv H'$, if they contain the same operations, have the same *reads-from* relations, and produce the same final writes. The notion of view equivalence of CCMV histories reduces to the simple condition $H = H'$, if the following *reads-from* relation is used, *T_i reads X from T_j*, in history $(H, \prec)$, if and only if $R_i(X_j) \in H$.

As pointed before, the snapshot-based isolation levels, such as CSI or GSI, include explicitly the time notion in their definitions and therefore in order to work with them it is suitable to use schedules, that contain simply the occurrence of the operations through time.

Definition 2. *Let $(H, \prec)$ be a history and $t\colon H \rightarrow \mathbb{R}^+$ a mapping such that it assigns to each operation $op \in H$ its real time occurrence $t(op) \in \mathbb{R}^+$. The schedule H_t of the history $(H, \prec)$ verifies:*

1. *if $op, op' \in H$ and $op \prec op'$ then $t(op) < t(op')$.*
2. *if $t(op) = t(op')$ and $op, op' \in H$ then $op = op'$.*

The mapping $t()$ totally orders all operations of $(H, \prec)$. Condition (1) states that the total order $<$ is compatible with the partial order $\prec$. Condition (2) establishes, for sake of simplicity, the assumption that different operations will have different times.

We are interested in operating with schedules, since it facilitates the work, but only with the ones that derive from CCMV histories over a concrete set of transactions T. One can note that an arbitrary time labeled sequence of versioned operations, e.g. $(R_i(X_j), t_1), (W_i(X_k), t_2)$ and so on, is not necessarily a schedule of a history. Thus, we need to put some restrictions to make sure that we work really with schedules corresponding to possible histories.

Property 1. *Let S_t be a time labeled sequence of versioned operations over a set of transactions T, S_t is a schedule of a history over T if and only if it verifies the following conditions:*

1. *there exists a mapping h such that $S = h(\bigcup_{i \in T_i} T_i)$.*
2. *if $op, op' \in T_i$ and $op \prec_{T_i} op'$ then $t(op) < t(op')$ in S_t.*
3. *if $R_i(X_j) \in S$ and $i \neq j$ then $W_j(X_j) \in S$ and $t(C_j) < t(R_i(X_j))$.*
4. *if $t(op) = t(op')$ and $op, op' \in S$ then $op = op'$.*

The proof of this fact can be inferred trivially. In the following, we use an aditional convention: (iii) A schedule H_t is a schedule of a history $(H, \prec)$.

Note that every schedule H_t may be represented by writing the operations in the total order ($<$) induced by $t()$. We define the *"commit time"* (c_i) and *"begin time"* (b_i) for each transaction $T_i \in T$ in a schedule H_t as $c_i = t(C_i)$ and $b_i = t(\textit{first operation of } T_i)$, holding $b_i < c_i$ by definition of $t()$ and $\prec_{T_i}$. We are going to use these values when working with schedules in order to represent the time sequence sense, apart from the operation that involves each value, since it facilitates the comprehension of some aspects explained in this work.

In the following, we formalize the concept of snapshot of the database. Intuitively it comprises the latest version of each data item. Let us consider the following transactions T_1, T_2 and T_3:

$$T_1 = \{R_1(X)\, W_1(X)\, C_1\}$$
$$T_2 = \{R_2(Z)\, R_2(X)\, W_2(Y)\, C_2\}$$
$$T_3 = \{R_3(Y)\, W_3(X)\, C_3\}$$

A sample of a posible schedule of these transactions might be the following one:

$$b_1\, R_1(X_0)\, W_1(X_1)\, c_1\, b_2\, R_2(Z_0)\, b_3\, R_3(Y_0)\, W_3(X_3)\, c_3\, R_2(X_1)\, W_2(Y_2) c_2.$$

As this example shows, each transaction is able to include in its snapshot (and read from it) the latest committed version of each existing item at the time such transaction was started. Thus T_2 has read version 1 of item X since T_1 has generated such version and it has already committed when T_2 started. But it only reads version 0 of item Z since no update of such item is seen by T_2. This is true despite transactions T_2 and T_3 are concurrent and T_3 updates X before T_2 reads such item, because the snapshot taken for T_2 is previous to the commit of T_3. This provides the basis for defining what a snapshot is. For that purpose, we need to define first the set of installed versions of a data item X in a schedule H_t, as the set $Ver(X, H) = \{X_j \colon W_j(X_j) \in H\} \cup \{X_0\}$, being X_0 its initial version.

Definition 3. *The snapshot of the database DB at time $\tau \in \mathbb{R}^+$ for a schedule H_t is defined as:*

$$Snapshot(DB, H_t, \tau) = \bigcup_{X \in DB} latestVer(X, H_t, \tau)$$

where the latest version of each item $X \in DB$ at time τ is the set:

$$latestVer(X, H_t, \tau) = \{X_p \in Ver(X, H) \colon (\nexists\, X_k \in Ver(X, H) \colon c_p < c_k \leq \tau)\}$$

From the previous definition, it is easy to show that a snapshot is modified each time an update transaction commits. If $\tau = c_m$ and $X_m \in Ver(X, H)$, then $latestVer(X, H_t, c_m) = \{X_m\}$.

In order to formalize some schedule-related concepts, we utilize a slight variation of the predicate *impacts*, presented in [2]. Consider two transactions $T_i, T_j \in T$:

- T_j impacts T_i on write at time $\tau \in \mathbb{R}^+$ in a schedule H_t, denoted T_j *w_impacts* T_i *at* τ, if the following predicate holds: $WS_j \cap WS_i \neq \emptyset \wedge \tau < c_j < c_i$.
- T_j impacts T_i on read at time $\tau \in \mathbb{R}^+$ in a schedule H_t, denoted T_j *r_impacts* T_i *at* τ, if the following predicate holds: $WS_j \cap RS_i \neq \emptyset \wedge \tau < c_j < c_i$.

From now on, when talking simply about *impacts*, we will be referring to impacts on write, and we will denoted it similarly as $T_j\, impacts\, T_i$.

2.2 Generalized Snapshot Isolation Definition

A hypothetical concurrency control algorithm could have stored some past snapshots. A transaction may receive a snapshot that happened in the system before the time of its first operation. The algorithm may commit the transaction if no other transaction impacts with it from that past snapshot. Thus, a transaction can observe an older snapshot of the DB but the write operations of the transaction are still valid update operations for the DB at commit time. These previous ideas define the concept of GSI.

Definition 4. *A schedule H_t is a GSI-schedule if and only if for each $T_i \in T$ there exists a value $s_i \in \mathbb{R}^+$ such that $s_i \leq b_i$ and:*

1. *if $R_i(X_j) \in H$ then $X_j \in Snapshot(DB, H_t, s_i)$; and*
2. *for each $T_j \in T$: $\neg$(T_j impacts T_i at s_i).*

Condition (1) states that every item read by a transaction belongs to the same (possible past) snapshot. Condition (2) also establishes that the time intervals $[s_i, c_i]$ and $[s_j, c_j]$ do not overlap for any pair of write/write conflicting transactions T_i and T_j.

Considering the transactions T_1, T_2, T_3 and T_4 described below,

$$T_1 = \{R_1(X)\, W_1(X)\, C_1\}, \qquad T_2 = \{R_2(Y)\, W_2(Y)\, C_2\},$$
$$T_3 = \{R_3(Z)\, W_3(Z)\, W_3(X)\, C_3\}, \quad T_4 = \{R_4(Z)\, R_4(X)\, C_4\}$$

the following schedule is an example of a GSI-schedule:

$$b_1\, R_1(X_0)\, b_2\, R_2(Y_0)\, W_1(X_1)\, c_1\, b_3\, R_3(Z_0)\, W_3(Z_3)\, W_3(X_3)\, c_3\, W_2(Y_2)\, c_2\, b_4\, R_4(Z_0)\, R_4(X_1)\, c_4.$$

In this schedule, transaction T_2 can be executed concurrently to T_1 and T_3 since it does not impact with them, but T_1 and T_3 cannot be executed concurrently since $WS_1 \cap WS_2 \neq 0$. Note that transaction T_4 reads X_1 (version of X established after the commit of T_1), despite that transaction T_3, which established a new version X_3, commits previously to the read operation of X in T_4. This is perfectly correct for a GSI-schedule, taking the time point of the snapshot used by T_4 (i.e. s_4) previous to the commit of T_3, as it is shown in the following schedule:

$$b_1\, R_1(X_0)\, b_2\, R_2(Y_0)\, W_1(X_1)\, c_1\, b_3\, R_3(Z_0)\, W_3(Z_3)\, \mathbf{s_4}\, W_3(X_3)\, c_3\, W_2(Y_2)\, c_2\, b_4\, R_4(Z_0)\, R_4(X_1)\, c_4.$$

Note also that T_4 reads Z_0, although the snapshot s_4 is taken after a write operation $W_3(Z_3)$ of transaction T_3. This is possible because, as pointed in Definition 1, versions of data items are always established after the transaction commitment, in our case c_3.

The intuition under this schedule in a distributed system is that the message containing the modifications of T_3 (the write operations on X and Z) would have not yet arrived to the site at the time transaction T_4 began. This may be the reason for T_4 to see this past version of items X and Z. Precisely, the fact that GSI captures these delays into schedules makes its usage on distributed environments attractive.

Remark 1. *As observed in the example, we can conclude that if there exists a transaction $T_i \in T$ such that conditions (1) and (2) from the Definition 4 are only verified for a value $s_i < b_i$ then there is an item $X \in RS_i$ for which $latestVer(X, H_t, s_i) \neq latestVer(X, H_t, b_i)$. That is, the transaction T_i has not seen the latest version of X at the begin time b_i, since there was a transaction T_k with $W_k(X_k) \in H$ such that $s_i < c_k < b_i$.*

2.3 Conventional Snapshot Isolation and Serializability

In CSI reading from a snapshot means that a transaction sees all the updates performed by transactions that committed before the transaction started its first operation. If condition (1) and (2) of the Definition 4 holds for $s_i = b_i$ for a transaction T_i then it means that transaction T_i sees the latest version of the items accessed by the transaction and then we can affirm that the isolation level of such transaction is CSI.

When considering a schedule of a history H_t, if for all $T_i \in T$ the level of each transaction is CSI then the schedule H_t is a *CSI-schedule*.

Let consider the previously proposed schedule:

$$b_1\, R_1(X_0)\, b_2\, R_2(Y_0)\, W_1(X_1)\, c_1\, b_3\, R_3(Z_0)\, W_3(Z_3)\, \mathbf{s_4}\, W_3(X_3)\, c_3\, W_2(Y_2)\, c_2\, b_4\, R_4(Z_0)\, R_4(X_1)\, c_4.$$

This schedule is not a possible CSI-schedule since, although transactions T_1, T_2 and T_3 fulfill the CSI level definition, condition (1) does not hold for transaction T_4. Transaction T_4 is reading old versions (X_1, Z_0) of the items X and Z, that does not match the latest version corresponding to the snapshot at its beginning, i.e. when $s_4 = b_4$.

We can think about a possible example of a CSI-schedule just ensuring that each transaction sees at its beginning the last version of the items it uses by changing the time when the snapshot for transaction T_4 (i.e. s_4) is taken:

$$b_1\, R_1(X_0)\, b_2\, R_2(Y_0)\, W_1(X_1)\, c_1\, b_3\, R_3(Z_0)\, W_3(Z_3)\, W_3(X_3)\, c_3\, \mathbf{s_4}\, W_2(Y_2)\, c_2\, b_4\, R_4(\mathbf{Z_3})\, R_4(\mathbf{X_3})\, c_4.$$

This ensures that transaction T_4 really reads the latest versions of X and Z available at its beginning, which are the ones established after the commitment of transaction T_3. Note that we can relax the condition $s_i = b_i$ and s_i may be previous to b_i if there exists no write operation that modifies some of the items read by T_i between this two events.

Serializable level provides the highest transaction isolation. This level describes a serial transaction execution, as if transactions had been executed one after another serially. In SER, as in CSI, a transaction sees only data committed before its beginning. However, a serial execution requires that transactions cannot modify items read by another concurrent transaction. Thus, if a transaction T_i verifies condition (1) and (2) of the Definition 4 and besides the following condition:

3. *for each* $T_j \in T : \neg(T_j\, r\_impacts\, T_i\, at\, s_i)$

then we can assure that the isolation level of such transaction is serializable (SER). This fact has been already formally proved in [2]. Note that the serializable level achieved through the previous condition is far more restrictive than the one provided by the general definition of a strict serializable history [35].

When considering a schedule of a history H_t, if for all $T_i \in T$ the level of each transaction is SER then H_t is a *SER-schedule*.

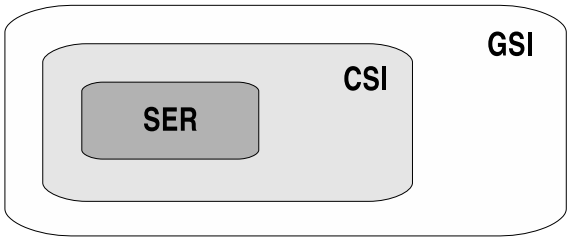

Figure 1: Relationship between database isolation levels considered in this paper

It is easy to note that the different isolation levels presented in this work are related. The relationship between these database isolation levels is clearly shown in the Figure 1. As can be inferred trivially, the CSI level is just a particular case of the GSI level definition, i.e. for $s_i = b_i$, when transactions see the latest version of the database. Therefore, any CSI-schedule is actually a GSI-schedule.

At first sight, we could think that it is possible to reach a serializable level coming from either CSI or GSI. This means that it would be possible to have two different definitions of the serializable level, provided that transactions do not modify items read by another concurrent transaction: a CSI-SER level where transactions see the latest database snapshot or a GSI-SER level where they may see older snapshots. However, the latter is not possible. If we have a GSI-schedule, then there exists at least a $T_k \in T$ such that $WS_k \cap RS_i \neq \emptyset$ and $s_i < c_k < b_i$ by the Remark 1. Therefore, $T_k\, r\_impacts\, T_i\, at\, s_i$ and hence a non-CSI schedule cannot be a SER-schedule what implies that SER-schedules are strictly contained in the set of CSI-schedules.

It is important to note that the presented isolation definitions are given for each transaction regarding other transactions. This implies a great flexibility since there can exist different sub-schedules in a global schedule, each one of which may contain operations that fulfill a given isolation level in their respective sub-schedules. Achieving this flexibility in replicated systems means allowing different transactions to be executed concurrently with different isolation levels.

2.4 Abortion Causes in Centralized Database Implementations

In order to study the behavior of a replicated database system, we have to understand well how centralized databases work. We have only considered up to now committed transactions in order to define the different

isolation levels. However, it is necessary to know the possible abortion causes of a centralized DBMS. They have a great influence over replication protocol designs, since replication protocols must consider all these causes in order to avoid anomalies when aborts arise.

Most commercial databases typically provide CSI level by default, e.g. Oracle [43] and PostgreSQL [44], but only a few databases, such as Microsoft SQL Server [45], provide actually the *real* theoretical definition of a serializable level [35] (not the ANSI one that is more relaxed). The main abortion reason is related to the fact of ensuring the transaction's isolation level. As explained above in this Section, there exist two main approaches for determining how to resolve isolation conflicts: the *first-committer-wins* and the *first-updater-wins* rules. In both cases, the ultimate effect is the same, i.e. to abort one of the concurrent transactions updating the same data item. Nevertheless, the first-updater-wins is more advantageous than the first-committer-wins since conflict checks are performed each time a transaction performs an update operation (or also a read operation in serializable). As Figure 2 shows, the first-updater-wins allows to detect conflicts sooner and also avoids performing possible unnecessary operations that will have to be rolled back when transactions try to commit. Note how, in the first-committer-wins approach, if one of the conflicting transactions commits, the other transaction keeps performing operations even knowing that it will abort when it tries to commit. On the contrary, if the first-updater transaction had finally aborted, the other transaction would have been blocked for a time unnecessarily. Commonly, commercial databases work by default with the first-updater-wins approach.

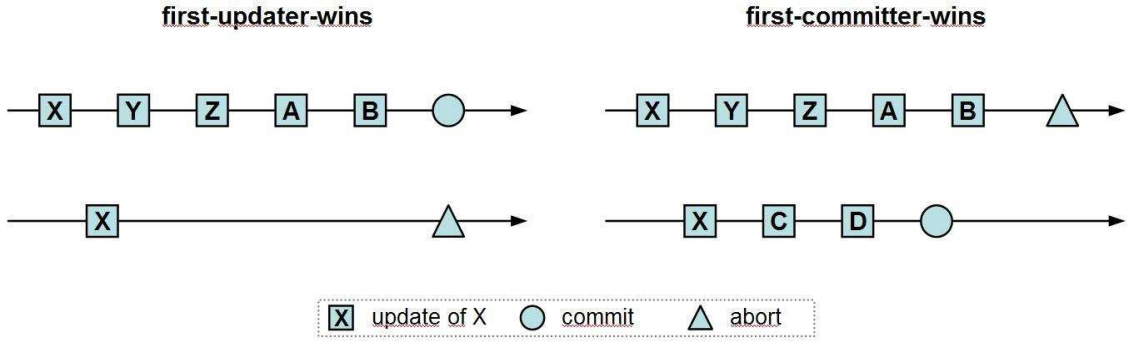

Figure 2: First-updater-wins and first-commiter-wins approaches

The CSI level does not avoid deadlocks. Most databases are based on locking implementation to provide CSI so deadlocks may arise between two transactions when both hold and are waiting for respective locks. To detect and resolve deadlocks, a concurrency control lock service should provide a deadlock detector that aborts one of the deadlocked transactions. Alternative schemes are possible for either avoiding deadlocks (the call to acquire a lock checks whether by waiting for the lock the transaction would become deadlocked), or associating timeouts with transactions or locks and aborting the transaction when the timeout expires. Replication protocols must consider how deadlocks may be resolved in order to work in consequence, e.g. a deadlock between two transactions may be resolved by aborting one or even both of them. Most commercial databases, including Oracle, PostgreSQL and SQL Server, provide mechanisms for deadlock detection that usually resolve them by rolling back one of the transactions involved in the deadlock, thus releasing one set of data locked by that statement.

Another critical feature of any database is data integrity. Transactions should leave items in a consistent state following given rules, e.g. the sum of two items must not be greater than a value. Therefore, there may be integrity rules that must be checked beyond isolation issues. This may lead to conflicts between transactions, with which databases deal similarly to isolation conflicts. Thus, integrity constraints may be checked at write time (*first-updater-wins*) or checking may be delayed until commit time (*first-committer-wins*).

Finally, note that a transaction may also abort due to database system failures (e.g., a processor failure), or just because a programmer chose to execute an abort call.

A replication protocol must consider all these kinds of abortion causes. It needs to know what happens

when an abortion occurs in order to proceed consequently to avoid anomalies and guarantee the correct operation of the system. In general, abortions caused by system failures are not considered since it is quite difficult to handle them due to its Byzantine nature. These kinds of situations would lead to disconnect the replica from the system and a recovery protocol should cover its reconnection when it gets recovered. We consider also that users do not abort explicitly transactions, although there is no problem with this. In our developed protocols, we do not distinguish between different causes of abortion since all the abort situations can be treated in a similar way.

In this work, we are going to consider that a DBMS works as most commercial databases do. That is, the first-updater-wins technique rules the resolution of isolation conflicts and deadlock situations are resolved by aborting one of the transactions involved in it.

Considering that most commercial databases support the CSI level, it seems interesting to study the behavior of a replicated database composed of DBMS replicas providing CSI. Thus, in the following Section we see how to study the isolation level of a replicated database working with CSI replicas referred to a corresponding equivalent centralized database.

3 One Copy Generalized Snapshot Isolation

Increasing system availability and also performance are the main reasons for using data replication [1]. In order to maintain the data freshness and consistency, a replicated database system requires a database replication protocol running at all sites. The traditional correctness criterion for replicated protocols is the 1CS [35]. A replicated database history is 1CS if it is equivalent to a serial execution in a centralized database. Many replication protocols verify such correctness criterion when the database management system at each site implements the strict two phase locking (2PL). However, it is not clear which isolation level is achieved when each database replica provides the CSI level, as most of commercial ones do.

In this Section, we study the conditions that a replication protocol has to verify to obtain one-copy GSI schedules. We set such conditions for ROWA (Read One-Write All) protocols since we consider no failures. This is not especially realistic , but this allows to simplify the presentation and nevertheless obtained conclusions can be directly extrapolated to a ROWAA approach.

One can note that the GSI concept is particularly interesting in replicated databases using ROWA protocols and databases with CSI. A ROWA replication protocol executes each transaction initially in a delegate replica, propagating later its updates to the rest of available replicas. This means that transaction write-sets cannot be immediately applied in all replicas at a time and therefore the snapshot being used by a transaction might be 'previous' to the one that would have been assigned to it.

The conditions a replication protocol has to verify in order to obtain one-copy GSI schedules using CSI replicas (proved in [4]) are: (i) Each submitted transaction to the system either commits or aborts at all sites (*atomicity*); (ii) All update transactions are committed in the same total order at every site (*total order of committed transactions*). Total order ensures that all replicas see the same sequence of transactions, being thus able to provide the same snapshots to transactions, independently of their starting replica. Without such order, those transactions without write/write conflicts might be applied in different orders in different replicas. So, transactions would be able to read different versions in different replicas. Atomicity guarantees that all replicas take the same actions regarding each transaction, so their states should be consistent, once each transaction has been terminated.

In the following, we first formalize the concept of the one-copy schedule for ROWA replication protocols and then we expound the main result of the one-copy GSI isolation level in replicated environments.

3.1 ROWA Replication Protocols

We consider a failure free distributed system that consists of m sites, being $I_m = \{1..m\}$ the set of site identifiers. Sites communicate among them by message passing. We make no assumptions about the time it takes for sites to execute and for messages to be transmitted. Each site k runs an independent instance of the database management system and maintains a copy of the database DB, that is, we consider a fully-replicated system. We will assume that each database copy, denoted DB^k with $k \in I_m$, is managed by a DBMS that provides the CSI level. We use the transaction model of Section 2. Let $T = \{T_i : i \in I_n\}$ be

the set of transactions submitted to the system, where $I_n = \{1..n\}$ is the set of transaction identifiers. We can also consider $T^k = \{T_i^k : i \in I_n\}$, i.e. the set of transactions submitted at each site $k \in I_m$ for the set T. In general, the ROWA approach establishes that some of these transactions are local at k while others are remote ones.

Formally, the ROWA strategy for replication defines for each transaction $T_i \in T$ the set of transactions $\{T_i^k : k \in I_m\}$ in which there is only one, denoted $T_i^{site(i)}$, verifying $RS_i^{site(i)} = RS_i$ and $WS_i^{site(i)} = WS_i$ ordered by $\prec_{T_i}^{site(i)}$. The rest of the transactions, T_i^k with $k \neq site(i)$, must have the same write operations as $T_i^{site(i)}$ and in the same order, i.e. $RS_i^k = \emptyset$ and $WS_i^k = WS_i$ with operations $T_i^k \subseteq T_i^{site(i)}$ and order $\prec_{T_i}^k \subseteq \prec_{T_i}^{site(i)}$. $T_i^{site(i)}$ determines the local transaction of T_i, i.e., the transaction executed at its delegate replica or master site, whilst $T_i^k, k \neq site(i)$, is a remote transaction of T_i, i.e., the updates of the transaction executed at a remote site. An update transaction reads at one site and writes at every site, while a read-only transaction only exists at its local site. In the rest of the paper, we consider the general case of update transactions with non-empty sets.

Note that we consider the general definition of the ROWA approach and hence our discussion is independent from any specific implementation of such strategy, i.e. it does not matter the techniques used to achieve such behavior.

In a ROWA replication protocol, as stated before, updates applied in a replica by a given transaction are also applied in the rest of replicas. Since only committed transactions are relevant for our discussion, the histories being generated at each site should be histories over T^k, as defined above. This implies that each transaction submitted to the system either commits at all replicas or in none of them, making possible to maintain the concept of full replication. This leads to the following assumption.

Assumption 1 (Atomicity)**.** *$(H^k, \prec^k)$ is a CCMV history over T^k for all sites $k \in I_m$.*

In the considered distributed system there is not a common clock or a similar synchronization mechanism. However, we can use a real time mapping $t : \bigcup_{k \in I_m}(H^k) \leftarrow \mathbb{R}^+$ that totally orders all operations of the system. This mapping is compatible with each partial order $\prec^k$ defined for H^k for each site $k \in I_m$. In the following, we consider that each DB^k provides CSI-schedules under the previous time mapping.

Assumption 2 (CSI Replicas)**.** *H_t^k is a CSI-schedule of the history $(H^k, \prec^k)$ for all sites $k \in I_m$.*

In order to study the level of isolation implemented by a ROWA protocol is necessary to define the one copy schedule (*1C-schedule*) obtained from the schedules at each site. A 1C-schedule of a replicated database describes its behavior as if it was a centralized system working over a logical copy of the full database. Thus, its isolation level is referred to that of the corresponding centralized database.

Let S_t be the complete schedule [2] of the distributed system over a set of transactions $T_i^k : k \in I_m$ and $i \in I_n$. That is, $S = \bigcup_k H^k$ and $S_t|^k = H_t^k$ being $S_t|^k$ the subschedule of S_t including only operations of site k. The ROWA approach guarantees that $t(op_i^{site(i)}) < t(op_i^k)$ with $k \neq site(i)$ when op_i is an update operation. This condition allows to avoid considering inconsistencies, e.g. $c_i^k < t(W_i(X_i)^{site(i)})$. However, note that $c_i^k < c_i^{site(i)}$, i.e. a remote transaction T_i^k may commit before $T_i^{site(i)}$.

In the next definitions, properties and theorems we use the following notation: for each transaction T_i, $i \in I_n$, $C_i^{min(i)}$ denotes the commit operation of the transaction T_i at site $min(i) \in I_m$ such that $c_i^{min(i)} = \min_{k \in I_m}\{c_i^k\}$ under the considered mapping $t()$. In the following, we proceed to define formally how a 1C-schedule is built from a given complete schedule of the distributed system.

Definition 5. *Let $T = \{T_i : i \in I_n\}$ be the set of committed transactions in a fully replicated database system with a ROWA strategy that verifies Assumption 1 and Assumption 2. Let S_t be the complete schedule of the system.*

The 1C-schedule, $H_{t'} = (H, t' : H \rightarrow \mathbb{R}^+)$, is built from S_t as follows:

For each $i \in I_n$ and $k \in I_m$

1. Remove from S operations such that:

$W_i(X_i)^k$, with $k \neq site(i)$, or

[2] S_t is a time labeled sequence (through a mapping $t : S \rightarrow \mathbb{R}^+$) of a set S of versioned operations

C_i^k, *with* $k \neq min(i)$

2. *H is obtained with the rest of operations in S after step 1, applying the renaming:*
   $W_i(X_i) = W_i(X_i)^{site(i)}$
   $R_i(X_j) = R_i(X_j)^{site(i)}$, *and*
   $C_i = C_i^{min(i)}$
3. *Finally, $t'()$ is obtained from $t()$ as follows:*
   $t'(W_i(X_i)) = t(W_i(X_i)^{site(i)})$
   $t'(R_i(X_j)) = t(R_i(X_j)^{site(i)})$, *and*
   $t'(C_i) = t(C_i^{min(i)})$

Condition (1) establishes that a 1C-schedule is built from the local operations of each transaction and that the commit time of a transaction is defined by the first commit time of the transaction at any site of the system. Condition (2) defines the logical operations that are considered to work over the logical copy of the replicated database. This condition ensures that a transaction is considered logically as committed as soon as it has been committed in any replica. As condition (3) states, $t'()$ receives its values from $t()$, preserving the original time order. Thus, by condition (3) we can write H_t instead of $H_{t'}$. In the 1C-schedule H_t, for each transaction T_i, is trivially verified $b_i < c_i$ because the ROWA strategy guarantees that for all $k \neq site(i)$, $b_i^{site(i)} < b_i^k$.

Conditions of the Definition 5 make possible that all the conditions explained in the Property 1 are verified by H_t. Therefore, it can be proved that any 1C-schedule H_t derives from a schedule of a history $(H, \prec)$ for the set of transactions T.

Although Assumptions 1 and 2 are included in Definition 5, they do not guarantee that the obtained 1C-schedule is a CSI-schedule. This is best illustrated in the following example, where it is also shown how the 1C-schedule may be built from each site CSI-schedules. In this example two sites and the next set of transactions are considered:

$$T_1 = \{R_1(Y), W_1(X), C_1\}, \quad T_2 = \{R_2(Z), W_2(X), C_2\},$$
$$T_3 = \{R_3(X), W_3(Z), C_3\}, \quad T_4 = \{R_4(X), R_4(Z), W_4(Y), C_4\}$$

Figure 3 illustrates the mapping described in Definition 5 for building a 1C-schedule from the CSI-schedules seen in the different nodes I_m. T_2 and T_3 are locally executed at site 1 ($RS_2 \neq \emptyset$ and $RS_3 \neq \emptyset$) whilst T_1 and T_4 are executed at site 2 respectively. The writesets are afterwards applied at the remote sites. Schedules obtained at both sites are CSI-schedules, i.e. transactions read the latest version of the committed data at each site. The 1C-schedule is obtained from Definition 5. For example, the commit of T_1 occurs for the 1C-schedule in the minimum of the interval between C_1^1 and C_1^2 and so on for the remaining transactions.

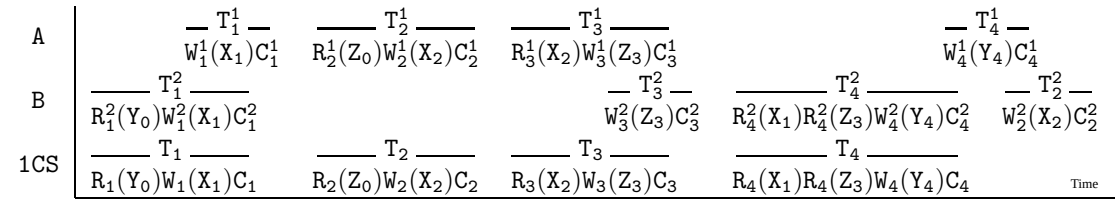

Figure 3: Replicated one-copy execution not providing CSI nor GSI.

In the 1C-schedule of Figure 3, T_4 reads X_1 and Z_3 but the X_2 version exists between both (since X_2 was installed at site 1). T_1 and T_2, satisfying that $WS_1 \bigcap WS_2 \neq \emptyset$, are executed at both sites in the same order. As T_1 and T_2 are not executed in the same order with regard to T_3, the obtained 1C-schedule is neither CSI nor GSI[3].

[3]Under the Assumptions (1) and (2), the obtained 1C-schedule seems to verify the conditions of the *read committed* isolation level definition, although this has not formally proved. Nevertheless, it is clear that the 1C-schedule provides an isolation level weaker than the original provided by each database replica.

Thus, we need that transactions whose writesets intersect ($WS_i \cap WS_j \neq \emptyset$) be executed in the same order, so as to observe at least the condition (2) of the Definition 4. However, this is not enough to guarantee that the 1C-schedule be equivalent to a GSI-schedule and another kind of restrictions must be considered.

3.2 Main Result

As we will explain later in Section 4, most developed replication protocols under CSI replicas are based on multicasting transaction writesets in total order, and on guaranteeing the same commit order in all replicas. Actually, the main issue in these protocols is to maintain this total order of commits. As a result, since all replicas generate CSI-schedules and their local snapshots have received the same sequence of updates, transactions starting at any site are able to read a particular snapshot, that perhaps is not the latest one, but that is consistent with those of other replicas.

Assumption 3 (Total order of committed transactions)**.** *For each pair T_i, $T_j \in T$, a unique order relation $c_i^k < c_j^k$ holds for all CSI-schedules H_t^k with $k \in I_m$.*

The CSI-schedules H_t^k have the same total order of committed transactions. Without loss of generalization, we consider the following total order in the rest of this section: $c_1^k < c_2^k < ... < c_n^k$ for every $k \in I_m$.

In the next property we are going to verify that, thanks to the total order, versions of items read by a transaction belong to the same snapshot in a given time interval. This interval is determined for each transaction T_i by two commit times, denoted c_{i_0} and c_{i_1}. The former corresponds to the commit time of a transaction T_{i_0} such that T_i reads an item from T_{i_0} for the last time and from then it performs no other read operation. The latter corresponds to the commit time of a transaction T_{i_1}, so that it is the first transaction that verifies $WS_{i_1} \cap RS_i \neq \emptyset$ and modifies the snapshot of the transaction T_i. In case that T_{i_1} does not exist, the correctness interval for T_i will extend from c_{i_0} to b_i.

Property 2. *Let H_t be a 1C-schedule verifying Assumption 3. For each $T_i \in T$ if $R_i(X_j) \in H$ then $X_j \in Snapshot(DB, H_t, \tau)$ and $\tau \in \mathbb{R}^+$ satisfies $c_{i_0} \leq \tau < c_{i_1} \leq b_i$.*

Proof. Let $T_{i_0}^{site(i)}$ be a transaction such that $WS_{i_0} \cap RS_i \neq \emptyset$ and $c_{i_0}^{site(i)}$ defines the last time in $H_t^{site(i)}$ from which transaction $T_i^{site(i)}$ no longer *reads from* $T_{i_0}^{site(i)}$ a version of a data item. By Assumption 2: $\forall Y \in WS_{i_0} \cap RS_i$: $\{Y_{i_0}^{site(i)}\} = latestVer(Y^{site(i)}, H_t^{site(i)}, b_i^{site(i)})$. By Assumption 1 and Definition 5: $T_{i_0} \in T$ and $c_{i_0} < b_i$.
Let $X \in RS_i$ be an item read by T_i such that $X \notin WS_{i_0} \cap RS_i$ and $\{X_j^{site(i)}\} = latestVer(\, X^{site(i)}, H_t^{site(i)}, b_i^{site(i)})$. We prove that $\nexists T_r \in T$: $X_r \in Ver(X, H) \wedge c_j < c_r < c_{i_0}$. Note that if this property is false, then the version X_r will be more up-to-date than X_j in H_t when T_i *reads from* T_{i_0}. The 1C-schedule H_t will not be a GSI-schedule. By contradiction, if there exists T_r and $c_j < c_r < c_{i_0}$ then by Assumption 3 and 1: $c_j^{site(i)} < c_r^{site(i)} < c_{i_0}^{site(i)}$. Thus, $X_j^{site(i)}$ is not the latest version in $H_t^{site(i)}$ at $b_i^{site(i)}$.
It is important to note that $c_{i_0}^{site(i)}$ defines the moment where $T_i^{site(i)}$ reads the latest version for $H_t^{site(i)}$. Hence, c_{i_0} will define for H_t the time instant of T_i's snapshot. If there exists a transaction T_{i_1} with $WS_{i_1} \cap RS_i \neq \emptyset$, then T_i will not see the versions installed by T_{i_1}. Thus, $b_i^{site(i)} < c_{i_1}^{site(i)}$. However, it may happen in H_t that $c_{i_1} < b_i$. In fact, this is the main reason to be H_t a GSI-schedule.
In conclusion, for all $X \in RS_i$, $X_j \in Snapshot(DB, H_t, c_{i_0})$ holds. This is valid for every $\tau, c_{i_0} \leq \tau$, until the first transaction $T_{i_1} \in T$ such that $WS_{i_1} \cap RS_i \neq \emptyset$ or until $b_i (b_i = b_i^{site(i)})$ if there not exists such a transaction. Therefore, $c_{i_0} \leq \tau < c_{i_1} \leq b_i$ holds.

□

The aim of the next theorem is to prove that the 1C-schedules generated by any ROWA protocol that verifies Assumption 3 are actually GSI-schedules; i.e., they comply with all conditions stated in Definition 4. Whilst proving that a transaction always reads from the same snapshot in a particular time interval is easy (Condition 1), it is not trivial to prove that there has not been any transaction that has impacted in that interval (Condition 2). However, due to the total commit order an induction proof is possible, showing that the obtained 1C-schedule verifies all conditions in order to be a GSI-schedule.

Theorem 1. *Under Assumption 3, the 1C-schedule H_t is a GSI-schedule.*

Proof. H_t is a GSI-schedule if it verifies Definition 4. Under Assumption 3 and 2, the CSI-schedules H_t^k have the same total order of committed transactions: $c_1^k < c_2^k < ... < c_n^k$ for every $k \in I_m$. H_t also verifies such an order $c_1 < c_2 < ... < c_n$ because by Definition 5 $min_{k \in I_m}\{c_i^k\} < min_{k \in I_m}\{c_j^k\}$ with $1 \leq i < j \leq n$. The rest of the proof is made by induction over such a total order. First, we define the subsets of transactions for each $i \in I_n$: $T(i) = \{T_1, T_2, ..., T_i\} \subseteq T$ and for each $k \in I_m$: $T^k(i) = \{T_1^k, T_2^k, ..., T_i^k\} \subseteq T^k$. Using these subsets we define $H_t^k(i)$ and $H_t(i)$. They are exactly equal to H_t^k and H_t respectively, except that they only include the operations in $T^k(i)$ or $T(i)$. Thus, it is clear that $H_t^k(n) = H_t^k$ and $H_t(n) = H_t$.
Induction Base. $H_t(1)$ is a GSI-schedule. There is only one committed transaction in $T(1)$. Therefore, Definition 4 is trivially verified for $H_t(1)$.
Induction Hypothesis. $H_t(j)$ is a GSI-schedule $1 \leq j \leq i-1$.
Induction Step. We will prove that $H_t(i)$ is a GSI-schedule, $i \in I_n$. Note that $T(i) = T(i-1) \cup \{T_i\}$. As $H_t(i-1)$ is a GSI-schedule, by Hypothesis, for any pair $T_j, T_j' \in T(i-1)$: $\neg(T_j$ impacts T_j' at $s_j')$. As $c_j < c_i$ for $1 \leq j \leq i-1$, by the considered total order, $\neg(T_i$ impacts T_j at $s_j)$ in $H_t(i)$. If $R_j(X_r) \in H(i-1)$ and $X_r \in Snapshot(DB, H_t(i-1), s_j)$ for $1 \leq j \leq i-1$ then $R_j(X_r) \in H(i)$. $X_r \neq X_i$ because $c_j^{site(j)} < c_i^{site(j)}$ and $H_t^{site(i)}$ is a CSI-schedule and hence $X_r \in Snapshot(DB, H_t(i), s_j)$.
Therefore, in order to prove that $H_t(i)$ is a GSI-schedule, we only need to prove for $T_i \in T$ that there exists a value $s_i \leq b_i$ such that:
(a) if $R_i(X_r) \in H(i), X_r \in Snapshot(DB, H_t(i), s_i)$ and
(b) for each $T_j \in T(i) : \neg(T_j$ impacts T_i at $s_i)$.
The begin time $b_i^{site(i)}$ and the commit time c_{i_0} of the transaction $T_{i_0} \in T(i)$ from which T_i reads for the last time, allow us to define the sets:
$T_1(i) = \{T_j \in T : b_i^{site(i)} < c_j^{site(i)} < c_i^{site(i)}\}$
$T_2(i) = \{T_j \in T : c_{i_0}^{site(i)} < c_j^{site(i)} < b_i^{site(i)}\}$
By Assumption 2, $\forall T_j \in T_1(i) : WS_j \cap WS_i = \emptyset$, i.e. $H_t^{site(i)}$ is a CSI-schedule, and by definition of $T_{i_0} \in T$ and again Assumption 2, $\forall T_j \in T_2(i) : WS_j \cap RS_i = \emptyset$. Let $T_{i_2} \in T_2(i)$ be the last transaction such that in the total order it verifies $WS_{i_2} \cap WS_i \neq \emptyset$. Note that in the 1C-schedule, obtained from Definition 5, a commit time $c_j^{site(i)}$ for a transaction in $T_1(i)$ may change its relation with respect to b_i, but maintaining the order relation with respect the other commit times. Let $T_{i_1} \in T_1(i)$ be the first transaction such that $c_{i_1} < b_i$ in H_t and $WS_{i_1} \cap RS_i \neq \emptyset$ Thus, $c_{i_0} < c_{i_2} < c_{i_1} < b_i$ holds in H_t.
For any value $s_i \in (c_{i_2}, c_{i_1})$, (a) holds for Property 2 and (b) holds by the way $T_{i_2} \in T$ has been defined. For each T_j such that $c_{i_2} < c_j < b_i$, if $T_j \in T_2(i)$ then $WS_j \cap WS_i = \emptyset$. If not, T_{i_2} is not the last transaction verifying such a condition; and if $T_j \in T_1(i)$ then $WS_j \cap WS_i = \emptyset$. Thus, these transactions do not impact with T_i. The rest of transactions do not either impact with T_i because their commit times are sooner than s_i.
To conclude, if there does not exist T_{i_1} then $s_i = b_i$ and therefore (a) and (b) holds. In case that it does not exist T_{i_2} then $s_i \in (c_{i_0}, c_{i_1})$ and again (a) and (b) holds.

□

This proof has not been given before in any ROWA-based CSI replication protocol ensuring total order for the commit operations of all transactions in the system replicas. This theorem formally justifies such protocols correctness and establishes that their resulting isolation level is GSI. Additionally, it is worth noting that Assumption 3 is a sufficient condition, but not necessary, for obtaining GSI. Despite this, replication protocols that comply with such an assumption are easily implementable. In order to conclude this Section, we establish the correctness criterion for replicated protocols based on total order guarantees and we study different implementations presented recently in the literature in the following Section.

3.3 Correctness Criteria for Database Replication Protocols

Database replication protocols are a particular case of distributed algorithms. In traditional distributed algorithms, correctness criteria of distributed algorithms are formulated basing on a specific interface that models the requirements for which the algorithm was developed. In order to prove algorithm's correctness, safety and liveness properties are defined on this interface.

However, this is useless when working with database replication protocols. In this case, transactions compose the system interface. Thus, properties for their correctness proves need to be defined over the

behavior of transactions. This properties must ensure that the replicated database system works as if it were a unique data management system providing the required isolation level.

For the kind of protocols proposed in this paper and the assumptions made about their operation in the previous sections to study formally their behavior, the following correctness criteria may be useful in order to facilitate their correctness proofs.

In order to ensure the system consistency, all transactions must commit in the same order at all available replica sites. Beside this safety property, a data replication protocol must guarantee other liveness properties such as the atomicity of a transaction:

- If a transaction commits at a site, it will finally commit at all sites.
- If a transaction aborts at a site, it will finally abort at all the sites where such transaction had started or it will be discarded at all the sites where the transaction had not yet started.

Another interesting liveness property to consider is that if infinitely often transactions are submitted to the system, infinitely often transactions will commit. This ensures fairness when submitting transactions so that a transaction will never be aborted infinitely, if it wants to commit its changes.

Finally, it is also necessary to ensure that there can exist no distributed deadlock situations.

4 Replication Protocols over CSI replicas

Several database replication protocols have been proposed in the latter years. Most of them consider that database replicas provide strict serializable isolation level. This, however, is not the usual case since most database vendors provide lower degrees of isolation levels such as CSI. Thus, current research is focused on developing replication protocols over CSI replicas.

Although they provide a good performance, protocols based on database core modifications are not very flexible. In order to adapt some protocols that worked with serializable database replicas [16] to work with CSI ones [46], it is necessary to reimplement all the core modifications performed previously. As a result, most of database replication protocols are based on middleware architectures that simplify the development and maintenance of replication protocols since the database internals remain inaccessible.

Elnikety et al. [47, 2] formalized some useful isolation issues for replicated environments. They pointed out in those works that CSI is impossible to achieve in a replicated setting without blocking transactions (including read-only ones) on their start until writesets of all prior transactions are received and applied. As a result of this, they introduced the GSI concept for the first time, relaxing the required freshness for the snapshot being taken when a transaction starts in its local replica. They also established two sufficient conditions (one statically checked an one dynamic) that guarantee serializable execution for transactions running under GSI.

In [47, 2], two implementation strategies based on certification for GSI replication protocols are introduced: the first uses centralized certification and the second uses distributed certification. They compare analytically the performance of the GSI level to CSI when using a centralized certification approach. The model shows that the response time in GSI is generally lower than in CSI, but the abort rate in GSI may be higher than in CSI since it compromises the data freshness. In their following works [48, 49], they study some aspects about the integrity of transactions when applying remote writesets and propose also some enhancements, such as compacting writesets or a memory aware load balancer that distributes queries so that updates can be executed in-memory, to improve performance and scalability of database replication protocols.

Considering the former premises, Lin et al. proposed in [3] a middleware replication protocol based on a distributed certification scheme that provided the CSI level as a centralized database system providing that the underlying database replicas provided CSI. In this work, they consider a gray approach [50] where the middleware does not handle the database as a black box. It is a must not to reimplement features provided by the underlying DBMS since it performs them much more efficiently. Replication protocols could take advantage of these features and hence, the replication code can be separated from the regular database operations. For example, the transaction writeset retrieval can be optimized easily using database features.

Besides, they propose in their work some optimizations for the replication middleware. Their middleware includes a mechanism that is able to apply concurrently several writesets that do not intersect providing a better system performance. However, it is necessary to block the transaction start in order to not lose the CSI level. There exists also another issue to be considered in their protocol specification. Consider that a certified writeset starts a local transaction in one replica. It may become blocked by another existing local transaction in that replica. The writeset application will stay blocked until the local transaction tries to commit. Then the local transaction will realize that it has to abort and it finally aborts. However, transactions associated with certified writesets may become blocked by local transactions (that should finally abort) during a long time. Furthermore, it is possible that these transactions be aborted and in consequence the writeset application must be reexecuted to keep the system consistency.

In order to avoid this, Muñoz et al. propose in [29] to include, following the gray approach, a block detector in each database replica. The block detector is able to detect the blocking situation before the commit time. When it is detected, the replication protocol is notified to operate as necessary. This allows to abort the local transaction earlier and therefore the middleware provides a higher performance since writesets can be applied sooner. Its cost is quite low and it also allows the protocols themselves to become simpler. They have implemented and tested this mechanism with snapshot-based replication protocols and the obtained results prove that the performance of this approach is better than a programmed check at the middleware layer.

The impossibility result presented in [2] is formally proved by González de Mendívil et al. in [4], together with the formal conditions required to obtain GSI in a replicated system. In such work, it is formally proved that replication protocols exclusively based on propagating transaction writesets cannot achieve the strict one-copy equivalent CSI level unless they block the beginning of transactions until they get the latest system snapshot. Thus, they propose a mechanism for replication protocols based on the total order broadcast of a *start* message at the beginning of a transaction in order to guarantee the CSI level in a replicated setting.

Due to this limitation, instead of using a strict one-copy CSI level, we have proposed in a recent work [51] to select the *outdatedness* of the snapshot being taken when a transaction starts. This makes possible to select which kind of snapshot isolation compliance is needed by each transaction, ranging from a default 1C-GSI to a 1C-CSI. Our proposed approach is far more optimistic since transactions do not get blocked even for the CSI case. It is only necessary to restart them when conflicts arise and normally they are detected soon and only a few operation may be rolled back.

Most protocol proposals provide excellent performance in LAN environments by using useful multicast primitives. However, little research has been done regarding whether these solutions can also be applied to WANs. In [52], a detailed WAN based performance analysis of data replication protocols is presented together with some optimizations proposed to circumvent the limitations of the replication protocols in WANs. There exist also other works that show how to maintain the snapshot isolation in lazy replicated systems. For example, Daudjee and Salem describe in [53] some algorithms that take advantage of the local CSI concurrency controls to maintain global snapshot isolation in lazy replicated systems.

Replication protocols presented in the literature are mostly based on the certification approach. However, no relevant works about using the weak-voting alternative to obtain snapshot isolation in replicated systems exist. We have studied this possibility in some recent works [38, 54]. Thus, we have proposed some weak-voting protocols based on a middleware architecture that are able to provide several isolation levels. These preliminar works set the starting point for this work. In these works, as in this paper, we present the replication algorithms based on a formal model since this simplifies not only the proof of the correction criteria of the protocols, but also their future implementation.

5 System Formal Model

For our proposal, we have taken the advantage from our previous works [29] and other middleware architectures providing database replication [3]. Thus, several replication protocols are proposed in this work taking advantage of the capabilities provided by a middleware architecture called MADIS [29]. For the sake of the explanation of the replication protocols, 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.

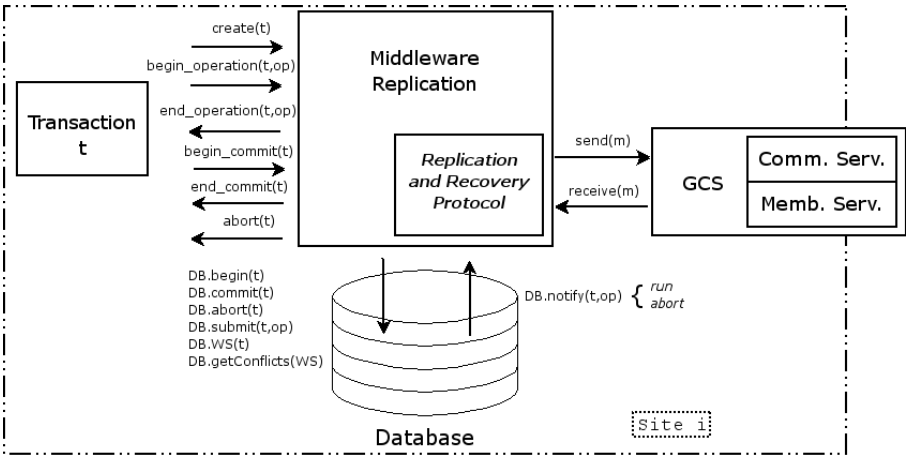

Figure 4: Main components of the system

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 respective 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 4 describe how components interact with each other. Actions may easily be ported to the particular communication primitives and DBMS JDBC-like operations.

5.1 Communication System

Communication among sites is mainly based on the services provided by a Group Communication System (GCS) [31]. Basically, a GCS consists of a membership and a communication service [55]. 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, according to its delivery guarantees.

5.2 Database

We assume a DBMS ensuring ACID transactions and complying with the SI level. The DBMS, as it is depicted in Figure 4 gives to the middleware some common actions. $DB.begin(t)$ begins a transaction t[4]. $DB.submit(t, op)$, 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 CSI level as the *first-updater-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

[4]In the following, transactions are denoted by the lowercase letter t.

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 [23, 29]. The database action $DB.WS(t)$ retrieves the transaction writeset, 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.

5.3 Transactions

Different transactions may be created in the replicated system. Each transaction t has also 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 interact 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 original transaction ($t.WS$) 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. In general, the protocols presented in this work are able to provide GSI level by default, given that transactions are atomically committed at all sites and their commit is totally ordered [4]. In order to obtain higher isolation levels, such as serializable or CSI, it is only necessary to set some constraints on the normal operation of the protocols.

5.4 Protocols

The protocols presented in this work, are modeled as state transition systems. Each state transition system 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 Figures), 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 Figures) 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 explanations given for each protocol operation. Without generalization loss, we assume a failure free environment throughout the protocols description.

6 Weak Voting Protocols

This technique, described firstly in [17], uses a weak voting phase for committing transactions, i.e. the transaction master site takes the decision to commit or abort. Weak voting [33] replication protocols usually follow the eager-update-everywhere strategy: transactions are locally executed and then changes are propagated, following a ROWAA approach. All changes performed by a transaction in the database are grouped in a writeset and delivered to the rest of the sites using a total order broadcast delivery. After its application, a reliable delivery of a commit or abort message from its master site will decide whether the transaction must commit or however abort. This message does not need to be total ordered but must be reliable to preserve the system consistency. The voting is said to be weak as only the master site can decide on the outcome of the transaction. Other servers cannot influence this decision and must abide by the master site decision. Figure 5 illustrates this technique basic operation.

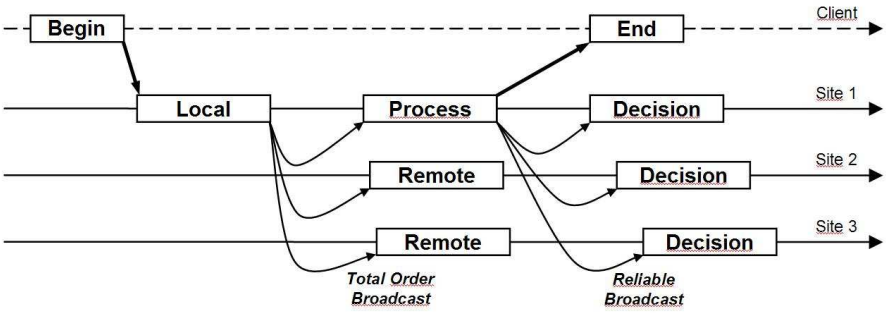

Figure 5: Weak voting replication scheme. When the system receives a transaction from a client, it will be submitted to an available site which executes transaction locally. When commit time is reached, the transaction writeset is broadcast to all servers using a total order mechanism. Upon delivering the message that contains the writeset of transaction at its master site, it can determine if conflicting transactions have been committed and consequently abort or commit the transaction. Thus, the master site sends a new broadcast containing the outcome of the transaction (commit or abort).

6.1 Basic Sequential Weak Voting Protocol

In this Section we present a basic weak voting protocol that commits transaction sequentially in the database replicas. This protocol avoids in advance possible conflicts that may happen between writesets of transactions coming from remote sites and local transaction operations. Thus, when a writeset is being applied, local transactions are not permitted to submit any possible write operation to the database. Besides, before applying writesets from remote transactions, it is necessary to abort all existent conflicting transactions in the local database. In this way, we are ensuring that the writeset application is going to be performed correctly, since we have remove any possible conflictive transaction from the database and besides we are allowing no other local transaction to submit any conflictive operation. However, writesets from remote transactions may conflict between them. In order to avoid this, we do neither permit a writeset to be submitted to the local database if another transaction that submitted another writeset has not still finished, either committing or aborting. In the following Section, we describe in detail the protocol operation.

6.1.1 Protocol Description

The protocol presented in this Section is modeled as a state transition system, as explained in Section 5.4. Figure 8 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 and describes the set of possible actions, detailing their preconditions and effects. Most actions are only applicable to either the master site of the transaction ($t.site = k$) or the rest of remote sites of the replicated system ($t.site \neq k$). In the Figure 8, N stands for the number of sites, 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. We explain such algorithm on the sequel.

A transaction t may be created at any site k of the replicated database system, which will be considered as its master site and where the transaction is considered as a *local* transaction. It can start its execution 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.

The transaction creation action may be followed by a sequence of pairs of $begin\_operation_k(t, op)$ and $end\_operation_k(t, op)$ actions. Each pair corresponds to a successful completion of a set of SQL statements. The invocation of a $begin\_operation$ submits the SQL statement to the database ($DB_k.submit(t, op)$) and sets its status to $submitted$. It is important to consider that a local transaction may conflict with a writeset application of a remote transaction once executed the $execute\_WS_k(t)$ action. Writeset modifications must be applied atomically in the database, without allowing other local or remote transactions

to conflict with the modified values, to achieve the corresponding isolation level and also to prevent distributed and local deadlock situations. This may happen when a write operation is submitted to the database ($type(op) =$ WRITE), as read operations will not conflict since we are considering databases complying with the SI level. Thus, a write 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 either the isolation level or integrity constrains, and also to local deadlock resolutions. The $end\_operation_k(t)$ action will be eventually invoked after the operation is successfully completed in the database. It sets the transaction status to $active$, enabling the local transaction to submit a new operation.

Once the transaction is done, it requests its commitment by means of the $begin\_commit_k(t)$ action, as $status =$ active after the last operation applied successfully in the local database. In this action, the transaction writeset needs to be 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 the transaction status will be changed to pre_commit.

Writeset message ($\langle$writeset$, t\rangle$) 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 $to\_commit_k(t)$ 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 ($\neg ws\_run_k$) and there is no other local transaction waiting for committing ($\neg local\_tocommit_k$) 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 $local\_tocommit_k$ 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 ($end\_commit_k(t)$) and will set the variable $local\_tocommit_k$ to $false$, 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 to apply it in that site if the $execute\_WS_k(t)$ 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 $execute\_WS_k(t)$ action requires that no other writeset is being applied in the database ($\neg ws\_run_k$) and also that no local transaction is waiting ($\neg local\_tocommit_k$) for commit.

The $execute\_WS_k(t)$ 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 to abort its remote transactions. This message will enable the $abort\_WS_k(t)$ at the sites where the writeset has been already submitted ($execute\_WS_k(t)$) and the remote transaction will be aborted in the local database. However, 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 abort decision may be delivered before the reception of the writeset message, which is broadcast in total order as the protocol states, or before its application in the local database. In both cases, where writeset application has not been performed ($status_k(t) =$ idle), this abort message will enable the $early\_decision_k(t)$ action and the remote transaction will immediately abort, setting its status to $aborted$ in order to discard the writeset message pending from reception. Thus, the $discard_k(t)$ action discards writeset messages ($\langle$writeset$, t\rangle$) of remote transactions that have been aborted by an early decision of the master site, in order to guarantee the progress allowing other writeset messages in the $TO\_channel_k$ to be processed. This action also allows to discard a writeset message related to a local transaction that has been aborted due to a conflict with a writeset of a remote transaction.

Once the writeset is successfully applied, if a $commit$ message has been received from the master site

Signature:
$\{\forall\, k \in N, t \in T, m \in M, op \subseteq OP$: $\mathbf{create}_k(t), \mathbf{end\_operation}_k(t, op), \mathbf{begin\_operation}_k(t, op)$,
$\mathbf{begin\_commit}_k(t), \mathbf{to\_commit}_k(t), \mathbf{end\_commit}_k(t), \mathbf{discard}_k(t, m), \mathbf{early\_decision}_k(t)$,
$\mathbf{execute\_WS}_k(t), \mathbf{end\_operation\_WS}_k(t, ws), \mathbf{local\_abort}_k(t), \mathbf{commit\_WS}_k(t), \mathbf{abort\_WS}_k(t)\}$

States:
$\forall\, k \in N, t \in T$: $status_k(t) \in \{idle, active, submitted, pre\_commit, await, tocommit, committed, aborted\}$, initially $status_k(t) =$ idle
$\forall\, k \in N$: $TO\_channel_k \subseteq \{m$: $m = \langle ws, t\rangle\ \forall t \in T\}$, initially $TO\_channel_k = \emptyset$
$\forall\, k \in N$: $R\_channel_k \subseteq \{m$: $m = \langle commit, t\rangle$ or $m = \langle abort, t\rangle\ \forall t \in T\}$, initially $R\_channel_k = \emptyset$
$\forall\, k \in N$: $local\_tocommit_k$: **boolean**, initially $local\_tocommit_k =$ false
$\forall\, k \in N$: ws_run_k : **boolean**, initially $ws\_run_k =$ false

Transitions:

$\mathbf{create}_k(t)$ // $t.site = k$ //
pre $\equiv status_k(t) =$ idle
eff $\equiv status_k(t) \leftarrow$ 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).
eff $\equiv status_k(t) \leftarrow$ submitted
$DB_k.submit(t, op)$

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

$\mathbf{begin\_commit}_k(t)$ // $t.site = k$ //
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
$TO_broadcast(\langle$writeset, $t\rangle)$

$\mathbf{to\_commit}_k(t)$ // $t.site = k$ //
pre $\equiv$ m=$\langle$writeset, $t\rangle$ first in $TO\_channel_k$
$\wedge \neg ws\_run_k\ \wedge\ \neg local\_tocommit_k$
$\wedge\ status_k(t) =$ pre_commit.
eff $\equiv remove(m)$ from $TO\_channel_k$
$status_k(t) \leftarrow$ tocommit
$local\_tocommit_k \leftarrow$ true
$R_broadcast(\langle$commit, $t\rangle)$

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

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

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

$\mathbf{execute\_WS}_k(t)$ // $t.site \neq k$ //
pre $\equiv$ m=$\langle$writeset, $t\rangle$ first in $TO\_channel_k$
$\wedge \neg ws\_run_k\ \wedge\ \neg local\_tocommit_k$
$\wedge\ status_k(t) \in \{$idle, tocommit$\}$.
eff $\equiv remove(m)$ from $TO\_channel_k$
for each t' **in** $DB_k.getConflicts(t.WS)$
$DB_k.abort(t')$
if $status_k(t') =$ pre_commit **then**
$R_broadcast(\langle$abort, $t'\rangle)$
$status_k(t') \leftarrow$ aborted
$DB_k.begin(t)$
$DB_k.submit(t, t.WS)$
if $status_k(t) =$ idle **then**
$status_k(t) \leftarrow$ submitted
$ws\_run_k \leftarrow$ true

$\mathbf{end\_operation\_WS}_k(t, ws)$ // $t.site \neq k$ //
pre $\equiv DB_k.notify(t, ws) =$ run
$\wedge\ status_k(t) =$ submitted.
eff $\equiv status_k(t) \leftarrow$ await

$\mathbf{early\_decision}_k(t)$ // $t.site \neq k$ //
pre $\equiv$ m=$\langle ., t\rangle$ in $R\_channel_k$
$\wedge\ status_k(t) =$ idle.
eff $\equiv remove(m)$ from $R\_channel_k$
if $m = \langle$abort, $t\rangle$ **then**
$status_k(t) \leftarrow$ aborted
else if $m = \langle$commit, $t\rangle$ **then**
$status_k(t) \leftarrow$ tocommit

$\mathbf{abort\_WS}_k(t)$ // $t.site \neq k$ //
pre $\equiv$ m=$\langle$abort, $t\rangle$ in $R\_channel_k$
$\wedge\ status_t(k) \in \{$await, submitted$\}$.
eff $\equiv remove(m)$ from $R\_channel_k$
$status_k(t) \leftarrow$ aborted
$DB_k.abort(t)$
$ws\_run_k \leftarrow$ false

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

Figure 6: The state transition system of the basic sequential weak voting protocol

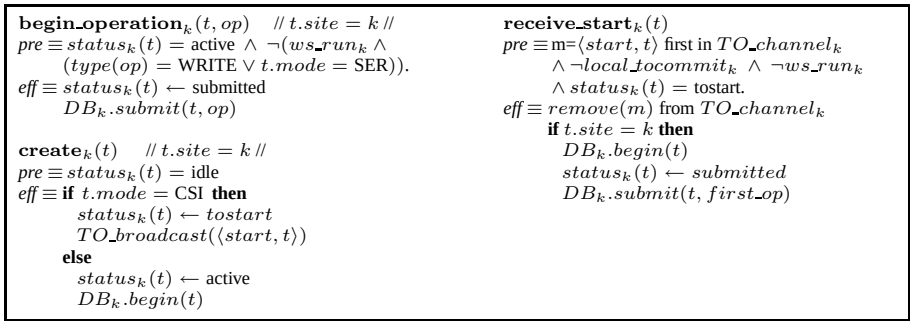

Figure 7: Weak voting protocol modifications for CSI and SER level support

in $early\_decision$ action, transaction status will have been modified ($tocommit$) and it will be waiting for commit. Thus, remote transaction will finally commit ($commit\_WS_k(t)$) locally in that replica. Otherwise, the $end\_operation\_WS_k(t, t.ws)$ for that site becomes enabled and it changes its status to $await$, waiting for the master site decision. The reception of a $commit$ or $abort$ message will enable the corresponding actions ($commit\_WS$ or $abort\_WS$) and remote transaction will finally either commit or abort in that replica. In both actions, the writeset application process finishes and other writesets must be allowed to be applied into the database ($ws\_run_k \leftarrow false$).

6.1.2 Multiple Isolation Levels

The proposed protocol is able to satisfy by default the GSI level, given that all transactions are atomically committed at all sites and their commit is totally ordered. In order to provide higher isolation level, such as CSI or serializable, some simple modifications for restricting the protocol operation can be included.

In order to obtain a serializable level, transaction readsets must not intersect with the writesets of other transactions committed previously, as it was pointed in Section 2.3. When a transaction requires a serializable environment ($t.mode =$ SER), read operations must be considered as write operations in order to guarantee the isolation level. Therefore, we need to avoid submitting read operations to the database if a writeset is just being applied by modifying the precondition of the $begin\_operation_k(t, op)$ action.

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.

On the other hand, a CSI transaction isolation level may be 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, the protocol must be modified in order to obtain CSI level when required, including a new action called $receive\_start_k(t)$ and also minor changes in the $create_k(t)$ action, as seen in Figure 7.

If a CSI level is established for the transaction, a $start$ message must be broadcast to all the replicas at its beginning, using a total order primitive. Afterward, the transaction must remain blocked ($status_k(t) =$ tostart), preventing new operations from being submitted 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 and therefore ensuring the CSI level. Otherwise, in GSI or serializable mode, the transaction can start straight away its reading and writing phase.

6.1.3 Discussion

As seen before, the protocol presented in this Section is a weak voting replication protocol which follows the eager-update-everywhere strategy. Thus, transactions are locally executed and then changes are propagated before committing, 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. Local conflicting transactions are aborted to ensure a correct writeset application, and no other local transaction or remote writeset application is allowed to interact with the database. 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.

Whilst most existing protocols are only able to provide a single isolation level (usually GSI with database replicas supporting CSI level), this replication protocol offers a greater flexibility to applications since it can operate with different isolation levels to transactions (GSI, CSI and SER) in a very simple way. This 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 other mechanism such as certification.

However, this initial approximation is fairly inefficient. Transactions are committed sequentially and it does not allow local transaction to operate while a writeset is being applied in the database. Besides, since local conflicting transactions must be aborted in order to apply successfully a writeset, it is necessary to call costly database methods so as to obtain the conflicting candidates. Thus, this protocol provides very poor concurrency and therefore system performance becomes fairly degraded. Moreover, this protocol does not guarantee the referential integrity nor the system consistency, since it only takes care of resolving isolation conflicts between transactions, what actually limits its practical application.

6.2 Enhanced Weak Voting Protocol with Block Detection

In essence, the protocol presented in the previous Section is quite pessimistic, as pointed out just above. 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 nature, the expected performance is quite poor. Nevertheless, several optimizations can be taken into account in order to improve significantly its performance.

The first proposed protocol includes a deadlock prevention schema in order to avoid that transactions become blocked in the local database replicas. An initial improvement of this protocol is to consider the replacement of this deadlock prevention mechanism with a detection mechanism as the one stated in [29] that has been successfully applied in several works with satisfying results [56].

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. In the following, we describe the necessary modifications for including this mechanism in the basic replication protocol.

6.2.1 Protocol Description

The required modifications of the protocol in order to work with a block detector are minimal regarding the original approach that tries to avoid blocking situations. Lines related with the block prevention must be replaced with the appropriated actions for dealing with blocks notified by the block detection mechanism. Thus, in the $execute\_WS_k(t)$ action, the operations for aborting all possible conflicting transactions before submitting a remote writeset disappear. We do not need this time to make a database call in order to get the existing transactions that conflicts with the writeset to be submitted ($getConflicts_k(t.ws)$).

Instead of this, we need a new action called $block\_detection_k(t, t')$. This action permits addressing issues related with blocking situations caused in the database. The block detector mechanism will notify the protocol automatically whenever a conflict is detected in the local database ($DB_k.blockDetector()$) and then the protocol will decide which transactions must abort and which not.

Signature:
$\{\forall\, k \in N, t \in T, m \in M, op \subseteq OP$: $\mathbf{create}_k(t), \mathbf{end\_operation}_k(t, op), \mathbf{begin\_operation}_k(t, op)$,
$\mathbf{begin\_commit}_k(t), \mathbf{to\_commit}_k(t), \mathbf{end\_commit}_k(t), \mathbf{discard}_k(t, m), \mathbf{early\_decision}_k(t)$,
$\mathbf{execute\_WS}_k(t), \mathbf{end\_operation\_WS}_k(t, ws), \mathbf{local\_abort}_k(t), \mathbf{commit\_WS}_k(t), \mathbf{abort\_WS}_k(t)$
$\mathbf{reexecute\_WS}_k(t), \mathbf{block\_detection}_k(t, t')\}$

States:
$\forall\, k \in N, t \in T$: $status_k(t) \in \{idle, active, submitted, pre\_commit, await, tocommit, committed, aborted\}$,
initially $status_k(t) =$ idle
$\forall\, k \in N$: $TO\_channel_k \subseteq \{m$: $m = \langle ws, t\rangle\ \forall t \in T\}$, initially $TO\_channel_k = \emptyset$
$\forall\, k \in N$: $R\_channel_k \subseteq \{m$: $m = \langle commit, t\rangle$ or $m = \langle abort, t\rangle\ \forall t \in T\}$, initially $R\_channel_k = \emptyset$
$\forall\, k \in N$: $local\_tocommit_k$: **boolean**, initially $local\_tocommit_k =$ false
$\forall\, k \in N$: ws_run_k : **boolean**, initially $ws\_run_k =$ false

Transitions:

$\mathbf{create}_k(t)$ // $t.site = k$ //
pre $\equiv status_k(t) =$ idle
eff $\equiv status_k(t) \leftarrow$ 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)
eff $\equiv status_k(t) \leftarrow$ submitted
$DB_k.submit(t, op)$

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

$\mathbf{begin\_commit}_k(t)$ // $t.site = k$ //
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
$TO_broadcast(\langle$writeset, $t\rangle)$

$\mathbf{to\_commit}_k(t)$ // $t.site = k$ //
pre $\equiv$ m=$\langle$writeset, $t\rangle$ first in $TO\_channel_k\ \wedge\ \neg ws\_run_k$
$\wedge\ \neg local\_tocommit_k\ \wedge\ status_k(t) =$ pre_commit
eff $\equiv remove(m)$ from $TO\_channel_k$
$status_k(t) \leftarrow$ tocommit
$local\_tocommit_k \leftarrow$ true
$R_broadcast(\langle$commit, $t\rangle)$

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

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

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

$\mathbf{execute\_WS}_k(t)$ // $t.site \neq k$ //
pre $\equiv$ m=$\langle$writeset, $t\rangle$ first in $TO\_channel_k\ \wedge\ \neg ws\_run_k$
$\wedge\ \neg local_tocommit_k\ \wedge\ status_k(t) \in \{$idle, tocommit$\}$.
eff $\equiv remove(m)$ from $TO\_channel_k$
$DB_k.begin(t)$
$DB_k.submit(t, t.WS)$
if $status_k(t) =$ idle **then**
$status_k(t) \leftarrow$ submitted
$ws\_run_k \leftarrow$ true

$\mathbf{reexecute\_WS}_k(t)$ // $t.site \neq k$ //
pre $\equiv DB_k.notify(t, t.WS) =$ abort
eff $\equiv DB_k.submit(t, t.WS)$

$\mathbf{end\_operation\_WS}_k(t)$ // $t.site \neq k$ //
pre $\equiv DB_k.notify(t, t.WS) =$ run $\wedge\ status_k(t) =$ submitted
eff $\equiv status_k(t) \leftarrow$ await

$\mathbf{early\_decision}_k(t)$ // $t.site \neq k$ //
pre $\equiv$ m=$\langle ., t\rangle$ in $R\_channel_k\ \wedge\ status_k(t) =$ idle
eff $\equiv remove(m)$ from $R\_channel_k$
if $m = \langle$abort, $t\rangle$ **then** $status_k(t) \leftarrow$ aborted
else if $m = \langle$commit, $t\rangle$ **then** $status_k(t) \leftarrow$ tocommit

$\mathbf{abort\_WS}_k(t)$ // $t.site \neq k$ //
pre $\equiv$ m=$\langle$abort, $t\rangle$ in $R\_channel_k$
$\wedge\ status_t(k) \in \{$await, submitted$\}$
eff $\equiv remove(m)$ from $R\_channel_k$
$status_k(t) \leftarrow$ aborted
$DB_k.abort(t)$
$ws\_run_k \leftarrow$ false

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

$\mathbf{block\_detection}_k(t, t')$
pre $\equiv t \rightarrow t' \in DB_k.blockDetector()\ \wedge\ t.site \neq$ k
$\wedge\ t'.site =$ k $\wedge\ status_k(t) \in \{$submitted, tocommit$\}$
eff $\equiv DB_k.abort(t')$
if $status_k(t') =$ pre_commit **then** $R_broadcast(\langle$abort, $t'\rangle)$
$status_k(t') \leftarrow$ aborted

Figure 8: The state transition system of the enhanced sequential weak voting protocol

However, we do not need to deal with any block that may arise in the local databases of a replicated system. Only block situations involving remote transactions with local transactions must be considered. The protocol must guarantee that remote transactions are finally applied. Thus, when a local transaction blocks a remote one ($t.site \neq k \wedge t'.site = k$), the local transaction must be aborted in order to guarantee that the remote one makes progress. If the transaction has reached the pre_commit state, and therefore writeset has been already sent to all sites, it is necessary to broadcast an $abort$ message ($<$ abort, $t'>$) in order to abort the transaction execution at all sites. Blocks between local transactions are not considered in this action. We let them be resolved as each local DBMS considers appropriate. Note that only local transactions that have not reached the pre_commit state may become blocked themselves and therefore their resolution does not matter to the replication algorithm. Blocks between remote transaction may never happen since remote writesets are sequentially submitted to the database and therefore there is no need to worry about them.

We do not make any considerations about the internals of the database replicas. Therefore, a remote transaction submitted to a local database may abort by a local deadlock with operations from other local transactions exiting in the database depending on how it is planned. We a writeset is applied we do not allow other operations to be submitted to the database. However, existing operations from local transactions may have not been yet planned by the database and therefore unexpected situations may arise. Thus, if a transaction associated to a remote writeset is aborted, then it will be necessary to reattempt to apply the writeset in the database until succeed ($reexecute\_WS_k(t)$). This is necessary to enforce that when a remote writeset is submitted to the local database, it will finally be applied in order to guarantee the transaction atomicity globally.

6.2.2 Discussion

The main improvement of this protocol is the block detection mechanism. This detection mechanism allows remote writesets to be directly submitted to the database replicas without worrying about checking anything in the database. Conflicts with existing transactions will not be prevented and instead they are detected on the fly. This reduces the protocol overhead, since unnecessary calls to database primitives are avoided when there is no conflicting local transaction. The block detector notifies the replication protocol when two transactions become blocked and therefore it interacts with the database only when it is strictly necessary, i.e. when a blocking situation occurs.

The use of this detection mechanism is not a problem for achieving multiple isolation levels. In fact, the same modifications proposed in Section 6.1.2 are also suitable for this case since its operation has no influence on the necessary modifications.

However, transactions are still submitted sequentially to the local replicas. This becomes a mayor drawback when the system load increases as it limits its multiprogramming level. Thus, in order to increase its performance when working with heavy loads of transactions, we should increase its concurrency level by allowing different transactions to be submitted to the database.

6.3 Enhanced Concurrent Weak Voting Protocol

The inclusion of the block detector mechanism in the previous protocol enhances its performance since reduces overhead related to communication with DBMS internals. However, as pointed out before, transactions are executed sequentially in the local database replicas and this becomes a burden on protocol performance when working with heavy loads. Thus, allowing several transactions to be executed concurrently in a local replica would increase the throughput of the replicated system.

Local transactions can run concurrently among them with no problem since local conflicting situations are resolved locally and do not affect to the rest of the replicas. Nevertheless, we have to be careful with the remote writesets submission. In order to keep data consistency among the replicas, conflicting writesets must be applied in the same order in all the sites. This applies to both remote writesets and local writesets of transactions that have request the commit. Thus, conflicting writesets must be applied in a row, one at a time. However, non-conflicting writesets can be concurrently submitted to the database. To that end, it is only necessary to keep a log with the writesets submitted to the database and not yet committed. This allows to check in advance whether there is any conflicting transaction and if that is not the case transaction

may be progress concurrently in the local replica. 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. In any case, it is very important to ensure that transactions finally commit in the very same order that the total order broadcast establishes since otherwise inconsistencies may arise in the replicated system.

6.3.1 Protocol Description

The deadlock detection mechanism introduced in the previous protocol, not only avoids performance overload but 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 simply it will be necessary to reattempt to apply the writeset in the database until succeed ($reexecute\_WS_k(t)$).

In order to apply a remote writeset concurrently to other transactions, we must ensure that there is no conflicting transaction in the database. Thus, it is necessary to keep track of writesets received in total order through a list of writesets ($WS\_submitted$), either from local transactions intending to commit or remote writesets submitted to the database. In both cases, there must be no conflicting transaction ($t.WS \cap WS\_submitted = \emptyset$) so as to process the writeset in the corresponding action, to_commit for writesets from local transactions and $execute\_WS$.

The list of writesets must be conveniently handle whenever either a writeset message is processed, including the received writeset ($WS\_submitted \leftarrow WS\_submitted \cup t.WS$) or whenever a transaction finally commits or aborts, removing the corresponding writeset from it ($WS\_submitted \leftarrow WS\_submitted - t.WS$). This allows to keep an updated list of the writesets from transactions that should finally commit, unless their master site decides to abort. So, other writesets may run concurrently in the local database, after checking whether they conflict with existing transactions that should commit or not. Notice that it is not necessary to worry about local transactions that are performing operations (not yet trying to commit) since they will be aborted when a conflict with a remote transaction is detected by the block detector ($block\_detection_k(t, t')$).

Finally, it is of vital importance that transactions commit its changes in all replicas in the same order so as to guarantee the consistency of the system. This is guaranteed thanks to the total order broadcast of the writeset messages, that sets the order in which transactions should commit in all the replicas to keep data consistency. In this protocol, we allow several transactions to be submitted and be running concurrently in a local replica. When concurrent transactions applies changes according to their respective remote writesets, they may finish their application in a different order from the sequence established by the total order delivery. Therefore, we use a sequence of transaction identifiers (sq_commit) that keeps the order in which writeset messages are received. Later, transactions are only allowed to commit, after applying their changes, when they are the first in that sequence ($\langle t \rangle$ first in sq_commit). This ensures that concurrent transactions finally commit in the very same order in which the total order delivery established for all the sites.

6.3.2 Discussion

The concurrent version of the protocol provides a greater performance specially when higher loads of transactions are submitted to the replicated system. Thus, allowing non-conflicting transactions to be executed concurrently increases the concurrency level and therefore system throughput becomes increased.

In this case, in order to achieve multiple isolation levels minor changes in the modifications for CSI and SER level support proposed in Section 6.1.2 are required. We may use the same modifications, but instead of the local variables used to control the access to the local database replicas ($local\_tocommit_k$ and ws_run_k) when a $start$ message is received ($receive\_start_k(t)$), we only need to wait on the commit of the transactions submitted to database. Considering the local variables used in this last protocol, this would imply wait until the list of writesets submitted to the database becomes empty ($WS\_submitted_k = \emptyset$). Thus, we ensure that the transaction waiting for the start message will see the latest changes performed in the database when it begins its operations.

Signature:

$\{\forall\, k \in N, t \in T, m \in M, op \subseteq OP\colon \mathbf{create}_k(t), \mathbf{end\_operation}_k(t, op), \mathbf{begin\_operation}_k(t, op),$
$\mathbf{begin\_commit}_k(t), \mathbf{to\_commit}_k(t), \mathbf{end\_commit}_k(t), \mathbf{discard}_k(t), \mathbf{ahead\_decision}_k(t),$
$\mathbf{execute\_WS}_k(t), \mathbf{end\_operation\_WS}_k(t, ws), \mathbf{local\_abort}_k(t), \mathbf{commit\_WS}_k(t), \mathbf{abort\_WS}_k(t)$
$\mathbf{reexecute\_WS}_k(t), \mathbf{block\_detection}_k(t, t')\}$

States:

$\forall\, k \in N, t \in T\colon status_k(t) \in \{idle, active, submitted, pre\_commit, await, tocommit, committed, aborted\}$, initially $status_k(t) =$ idle

$\forall\, k \in N\colon TO\_channel_k \subseteq \{m\colon m = \langle ws, t\rangle\ \forall t \in T\}$, 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 $R\_channel_k = \emptyset$

$\forall\, k \in N\colon sq\_commit_k \subseteq \{t\ \forall t \in T\}$, initially $sq\_commit_k = \emptyset$

$\forall\, k \in N\colon WS\_submitted_k \subseteq \{t.WS\ \forall t \in T\}$, initially $WS\_submitted_k = \emptyset$

Transitions:

$\mathbf{create}_k(t)$ // $t.site = k$ //
pre $\equiv status_k(t) =$ idle
eff $\equiv status_k(t) \leftarrow$ active
$DB_k.begin(t)$

$\mathbf{begin\_operation}_k(t, op)$ // $t.site = k$ //
pre $\equiv status_k(t) =$ active
eff $\equiv status_k(t) \leftarrow$ submitted
$DB_k.submit(t, op)$

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

$\mathbf{begin\_commit}_k(t)$ // $t.site = k$ //
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
$TO_broadcast(\langle$writeset, $t\rangle)$

$\mathbf{to\_commit}_k(t)$ // $t.site = k$ //
pre $\equiv$ m=$\langle$writeset, $t\rangle$ first in $TO\_channel_k$
$\wedge\ status_k(t) =$ pre_commit
$\wedge\ t.WS \cap WS\_submitted_k = \emptyset$
eff $\equiv remove(m)$ from $TO\_channel_k$
$status_k(t) \leftarrow$ tocommit
$WS\_submitted_k \leftarrow WS\_submitted_k \cup t.WS$
$sq\_commit_k \leftarrow sq\_commit_k.\langle t\rangle$
$R_broadcast(\langle$commit, $t\rangle)$

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

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

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

$\mathbf{execute\_WS}_k(t)$ // $t.site \neq k$ //
pre $\equiv$m=$\langle$writeset, $t\rangle$ first in $TO\_channel_k$
$\wedge\ status_k(t) \in \{$idle, tocommit$\}$
$\wedge\ t.WS \cap WS\_submitted_k = \emptyset$
eff $\equiv remove(m)$ from $TO\_channel_k$
$DB_k.begin(t)$
$DB_k.submit(t, t.WS)$
$WS\_submitted_k \leftarrow WS\_submitted_k \cup t.WS$
$sq\_commit_k \leftarrow sq\_commit_k.\langle t\rangle$
if $status_k(t) =$ idle **then** $status_k(t) \leftarrow$ submitted

$\mathbf{reexecute\_WS}_k(t)$ // $t.site \neq k$ //
pre $\equiv DB_k.notify(t, t.WS) =$ abort
eff $\equiv DB_k.submit(t, t.WS)$

$\mathbf{end\_operation\_WS}_k(t)$ // $t.site \neq k$ //
pre $\equiv DB_k.notify(t, t.WS) =$ run $\wedge\ status_k(t) =$ submitted
eff $\equiv status_k(t) \leftarrow$ await

$\mathbf{early\_decision}_k(t)$ // $t.site \neq k$ //
pre $\equiv$m=$\langle ., t\rangle$ in $R\_channel_k\ \wedge\ status_k(t) =$ idle
eff $\equiv remove(m)$ from $R\_channel_k$
if $m = \langle$abort, $t\rangle$ **then** $status_k(t) \leftarrow$ aborted
else if $m = \langle$commit, $t\rangle$ **then** $status_k(t) \leftarrow$ tocommit

$\mathbf{abort\_WS}_k(t)$ // $t.site \neq k$ //
pre $\equiv$m=$\langle$abort, $t\rangle$ in $R\_channel_k$
$\wedge\ status_t(k) \in \{$await, submitted$\}$
eff $\equiv remove(m)$ from $R\_channel_k$
$remove(\langle t\rangle)$ from sq_commit_k
$WS\_submitted_k \leftarrow WS\_submitted_k - t.WS$
$status_k(t) \leftarrow$ aborted
$DB_k.abort(t)$

$\mathbf{commit\_WS}_k(t)$ // $t.site \neq k$ //
pre $\equiv$ (m=$\langle$commit, $t\rangle$ in $R\_channel_k\ \wedge\ status_k(t) =$ await) $\vee$
$(DB_k.notify(t, t.WS) =$ run $\wedge\ status_k(t) =$ tocommit)
$\wedge\ \langle t\rangle$ first in sq_commit_k
eff $\equiv$ **if** $status_k(t) =$ await **then**
$remove(m)$ from $R\_channel_k$
$remove(\langle t\rangle)$ from sq_commit_k
$WS\_submitted_k \leftarrow WS\_submitted_k - t.WS$
$status_k(t) \leftarrow$ committed
$DB_k.commit(t)$

$\mathbf{block\_detection}_k(t, t')$
pre $\equiv t \rightarrow t' \in DB_k.blockDetector() \wedge t.site \neq$ k
$\wedge\ t'.site =$ k $\wedge\ status_k(t) \in \{$submitted, tocommit$\}$
eff $\equiv DB_k.abort(t')$
if $status_k(t') =$ pre_commit **then** $R_broadcast(\langle$abort, $t'\rangle)$
$status_k(t') \leftarrow$ aborted

Figure 9: The state transition system of the enhanced concurrent weak voting protocol

7 Conclusions

7.1 Summary

In this paper, we study ROWA protocols for database replication, where each replica uses a DBMS providing CSI isolation level. Other works have proved that ROWA replication protocols can not achieve the 1C-CSI isolation level unless they do block the beginning of transactions until they get the latest system snapshot. This potential blocking of transactions is a great drawback for its main advantage of non-blocking executions of read operations.

This is the main reason for introducing GSI in database replication scenarios. This paper establishes that the sufficient condition for obtaining a 1C-GSI consistency level is ensuring that transactions commit in the same total order in all replicas. All the properties that have been formalized in our paper seem to be assumed in some previous works, but none of them carefully identified nor formalized such properties. So, we have provided a solid theoretical basis for designing and developing replication protocols with GSI, and also some assumptions that may ease the implementation of replication protocols.

As a result, we have also proposed a database replication protocol based on a middleware architecture that is able to support different degrees of isolation (CSI, GSI and SER) on top of DBMSs supporting CSI. 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 CSI replicas, since it usually reduces the abortion rate and avoids the drawbacks certification presents, such as keeping track of its log. Since the original proposed protocol is rather pessimistic, we have also pointed out other enhanced protocols which include some optimizations for increasing the performance.

7.2 Future Lines

This paper has revisited some well-known ideas related to isolation levels of replicated databases and we have made the proof of one of these replicated isolation: the 1C-GSI level. All this work has brought out new questions that may serve as the guideline for future research.

We have worked with a concept of serializability achieved by a dynamic rule that as said before it seems to be more restrictive than the strict serializable level. Thus, it would be interesting to study formally which is the relationship of the strict serializable level with the ones presented in this paper.

Besides, using many of the explained concepts, we have proposed a set of protocols that are able to provide theorically different isolation levels. It is necessary however to make their correctness proofs to ensure that they work as were defined.

Another future line of work should be implement these protocols over an existing middleware architecture and test their performance not only among them, but also with other replication mechanism not based on a weak voting approach. For this purpose, it will be necessary to develop new replication protocols based on other replication techniques such as certification and study how this kind of protocols may provide different isolation levels in order to be able to compare with the ones proposed in this work.

References

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[2] S. Elnikety, F. Pedone, and W. Zwaenopoel, "Database replication using generalized snapshot isolation.," in *SRDS*, IEEE-CS, 2005.

[3] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris, "Middleware based data replication providing snapshot isolation.," in *SIGMOD Conference*, 2005.

[4] J. R. G. de Mendívil, J. E. Armendáriz, F. D. Muñoz, L. Irún, J. R. Garitagoitia, and J. R. Juárez, "Non-blocking ROWA Protocols Implement GSI Using SI Replicas," Tech. Rep. ITI-ITE-07/10, ITI, 2007.

[5] C. Plattner, G. Alonso, and M. Tamer-Özsu, "Extending DBMSs with satellite databases," *VLDB J.*, 2006. *Accepted for publication*.

[6] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha, "The dangers of replication and a solution.," in *SIGMOD Conference* (H. V. Jagadish and I. S. Mumick, eds.), pp. 173–182, ACM Press, 1996.

[7] M. Stonebraker, "Concurrency control and consistency of multiple copies of data in distributed ingres.," *IEEE Trans. Software Eng.*, vol. 5, no. 3, pp. 188–194, 1979.

[8] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems.," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 4, pp. 1018–1032, 2003.

[9] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso, "Understanding replication in databases and distributed systems.," in *ICDCS*, pp. 464–474, 2000.

[10] M. J. Carey and M. Livny, "Conflict detection tradeoffs for replicated data.," *ACM Trans. Database Syst.*, vol. 16, no. 4, pp. 703–746, 1991.

[11] K. Petersen, M. Spreitzer, D. B. Terry, M. Theimer, and A. J. Demers, "Flexible update propagation for weakly consistent replication.," in *SOSP*, pp. 288–301, 1997.

[12] J. Sidell, P. M. Aoki, A. Sah, C. Staelin, M. Stonebraker, and A. Yu, "Data replication in mariposa.," in *ICDE* (S. Y. W. Su, ed.), pp. 485–494, IEEE-CS, 1996.

[13] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi, "Exploiting atomic broadcast in replicated databases," *LNCS*, vol. 1300, pp. 496–503, 1997.

[14] Y. Amir and C. Tutu, "From total order to database replication.," in *ICDCS*, pp. 494–503, 2002.

[15] J. Holliday, R. C. Steinke, D. Agrawal, and A. E. Abbadi, "Epidemic algorithms for replicated databases.," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 5, pp. 1218–1238, 2003.

[16] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-R, a new way to implement database replication.," in *VLDB* (A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, eds.), pp. 134–143, Morgan Kaufmann, 2000.

[17] B. Kemme and G. Alonso, "A new approach to developing and implementing eager database replication protocols.," *ACM Trans. Database Syst.*, vol. 25, no. 3, pp. 333–379, 2000.

[18] S. Wu and B. Kemme, "Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation.," in *ICDE*, pp. 422–433, IEEE-CS, 2005.

[19] C. Amza, A. L. Cox, and W. Zwaenepoel, "Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites.," in *Middleware* (M. Endler and D. C. Schmidt, eds.), vol. 2672 of *Lecture Notes in Computer Science*, pp. 282–304, Springer, 2003.

[20] J. Armendáriz, J. González de Mendívil, and F. Muñoz-Escoí, "A lock-based algorithm for concurrency control and recovery in a middleware replication software architecture.," in *HICSS*, p. 291a, IEEE-CS, 2005.

[21] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "C-JDBC: Flexible database clustering middleware.," in *USENIX Annual Technical Conference, FREENIX Track*, pp. 9–18, USENIX, 2004.

[22] J. Esparza-Peidro, F. Muñoz-Escoí, L. Irún-Briz, and J. Bernabéu-Aubán, "Rjdbc: a simple database replication engine," in *Proc. of the 6th Int'l Conf. Enterprise Information Systems (ICEIS'04)*, 2004.

[23] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso, "Improving the scalability of fault-tolerant database clusters.," in *ICDCS*, pp. 477–484, 2002.

[24] F. Muñoz-Escoí, L. Irún-Briz, P. Galdámez, J. Bernabéu-Aubán, J. Bataller, and M. Bañuls, "GlobData: Consistency protocols for replicated databases.," in *YUFORIC'2001*, pp. 97–104, IEEE-CS, 2001.

[25] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso, "Scalable replication in database clusters.," in *DISC* (M. Herlihy, ed.), vol. 1914 of *Lecture Notes in Computer Science*, pp. 315–329, Springer, 2000.

[26] C. Plattner and G. Alonso, "Ganymed: Scalable replication for transactional web applications.," in *Middleware* (H.-A. Jacobsen, ed.), vol. 3231 of *Lecture Notes in Computer Science*, pp. 155–174, Springer, 2004.

[27] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente, "The globdata fault-tolerant replicated distributed object database.," in *EurAsia-ICT*, vol. 2510 of *LNCS*, pp. 426–433, Springer, 2002.

[28] U. Röhm, K. Böhm, H.-J. Schek, and H. Schuldt, "Fas - a freshness-sensitive coordination middleware for a cluster of olap components.," in *VLDB*, pp. 754–765, 2002.

[29] F. D. Muñoz, J. Pla, M. I. Ruiz, L. Irún, H. Decker, J. E. Armendáriz, and J. R. G. de Mendívil, "Managing transaction conflicts in middleware-based database replication architectures," in *SRDS*, pp. 401–410, IEEE-CS, 2006.

[30] V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," Tech. Rep. TR94-1425, Dep. of Computer Science, Cornell University, Ithaca, New York (USA), 1994.

[31] G. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: a comprehensive study.," *ACM Comput. Surv.*, vol. 33, no. 4, pp. 427–469, 2001.

[32] M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso, "Database replication techniques: A three parameter classification," in *SRDS*, pp. 206–217, 2000.

[33] M. Wiesmann and A. Schiper, "Comparison of database replication techniques based on total order broadcast.," *IEEE TKDE.*, vol. 17, no. 4, pp. 551–566, 2005.

[34] F. Pedone, *The database state machine and group communication issues (Thèse N. 2090)*. PhD thesis, École Polytecnique Fédérale de Lausanne, Lausanne, Switzerland, 1999.

[35] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil, "A critique of ansi sql isolation levels.," in *SIGMOD Conference*, pp. 1–10, 1995.

[36] K. Daudjee and K. Salem, "Lazy database replication with snapshot isolation.," in *VLDB* (U. Dayal, K.-Y. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y.-K. Kim, eds.), pp. 715–726, ACM, 2006.

[37] J. E. Armendáriz-Íñigo, J. R. Juárez-Rodríguez, J. R. González de Mendívil, H. Decker, and F. D. Muñoz-Escoí, "k-Bound GSI: A flexible database replication protocol," in *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, vol. 1, (New York, NY, USA), pp. 556–560, ACM Press, 2007.

[38] J. Juárez, J. Armendáriz, J. G. de Mendívil, F. Muñoz-Escoí, and J. Garitagoitia, "A weak voting database replication protocol providing different isolation levels," in *NOTERE'07: Proceeding of the 7th International Conference on New Technologies of Distributed Systems*, 2007.

[39] J. Melton and A. R. Simon, *Understanding the New SQL: A Complete Guide*. M. Kaufmann, 1993.

[40] ANSI. Database Language, ANSI Document X3.135-1992, November, 1992.

[41] A. Fekete, E. J. O'Neil, and P. E. O'Neil, "A read-only transaction anomaly under snapshot isolation.," *SIGMOD Record*, vol. 33, no. 3, pp. 12–14, 2004.

[42] C. Papadimitriou, *The Theory of Database Concurrency Control*. Computer Science Press, 1986.

[43] K. Loney and G. Koch, *Oracle8i: The Complete Reference (Book/CD-ROM Package)*. McGraw-Hill Professional, 2000.

[44] PostgreSQL, "The world's most advance open source database web site." Accessible in URL: `http://www.postgresql.org`, 2005.

[45] K. Delaney, *Inside Microsoft SQL Server 2000*. Redmond, WA, USA: Microsoft Press, 2000.

[46] S. Wu and B. Kemme, "Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation," *icde*, vol. 0, pp. 422–433, 2005.

[47] S. Elnikety, F. Pedone, and W. Zwaenopoel, "Generalized snapshot isolation and a prefix-consistent implementation," EPFL-Tech-Rep IC/2004/21, School of Computer and Communication Sciences (EPFL), Lausanne (Switzerland), Mar. 2004.

[48] S. Elnikety, S. Dropsho, and F. Pedone, "Tashkent: uniting durability with transaction ordering for high-performance scalable database replication," in *EuroSys '06: Proceedings of the 2006 EuroSys conference*, (New York, NY, USA), pp. 117–130, ACM Press, 2006.

[49] S. Elnikety, S. Dropsho, and W. Zwaenepoel, "Tashkent+: Memory-Aware Load Balancing and Update Filtering in Replicated Databases," in *EuroSys*, 2007.

[50] J. Salas, R. Jimenez-Peris, M. Patino-Martinez, and B. Kemme, "Lightweight reflection for middleware-based database replication," *srds*, vol. 00, pp. 377–390, 2006.

[51] J. E. Armendáriz-Íñigo, J. R. Juárez-Rodríguez, J. R. González de Mendívil, H. Decker, and F. D. Muñoz-Escoí, "k -bound gsi: A flexible database replication protocol," in *22nd Symposium on Applied Computing (SAC 2007), Dependable and Adaptive Distributed Systems (DADS)*, ACM Press, 2007.

[52] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris, "Consistent data replication: Is it feasible in wans?," in *Euro-Par*, pp. 633–643, 2005.

[53] K. Daudjee and K. Salem, "Lazy database replication with snapshot isolation," in *VLDB'2006: Proceedings of the 32nd international conference on Very large data bases*, pp. 715–726, VLDB Endowment, 2006.

[54] J. Juárez, J. G. de Mendívil, J. Garitagoitia, J. Armendáriz, and F. Muñoz-Escoí, "A middleware database replication protocol providing different isolation levels," in *PDP'07: Proceedings of the Work in Progess Session*, pp. 7–8, SEA-Publications, 2007.

[55] A. Bartoli, "Implementing a replicated service with group communication.," *Journal of Systems Architecture*, vol. 50, no. 8, pp. 493–519, 2004.

[56] J. E. Armendáriz, *Design and Implementation of Database Replication Protocols in the MADIS Architecture*. PhD thesis, Universidad Pública de Navarra, 2006.