Non-blocking ROWA Protocols Implement Generalized Snapshot Isolation Using Snapshot Isolation Replicas

J.R. González de Mendívil[1], J.E. Armendáriz-Iñigo[1], J.R. Garitagoitia[1],
L. Irún-Briz[2] and F.D. Muñoz-Escoí[2]
[1]*Dpto. Matemática e Informática*
Universidad Pública de Navarra
Campus Arrosadía s/n, 31006 Pamplona (Spain)
{mendivil, enrique.armendariz, joserra}@unavarra.es
[2]*Instituto Tecnológico de Informática*
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia (Spain)
{lirun, fmunyoz}@iti.upv.es

Technical Report ITI-ITE-06/04

Non-blocking ROWA Protocols Implement Generalized Snapshot Isolation Using Snapshot Isolation Replicas

J.R. González de Mendívil - J.E. Armendáriz-Iñigo - J.R. Garitagoitia
Dpto. Matemática e Informática
Universidad Pública de Navarra
Campus Arrosadía s/n, 31006 Pamplona (Spain)
{mendivil, enrique.armendariz, joserra}@unavarra.es

L. Irún-Briz - F. D. Muñoz-Escoí
Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia (Spain)
{lirun, fmunyoz}@iti.upv.es

July 11, 2006

Abstract

The concept of Generalized Snapshot Isolation (GSI) has been recently proposed as a suitable extension of conventional Snapshot Isolation (SI) for replicated databases. In GSI, transactions may use older snapshots instead of the latest snapshot required in SI, being able to provide better performance without significantly increasing the abortion rate when write/write conflicts among transactions are low. Its authors also state that GSI is needed because there is no non-blocking implementation of SI in asynchronous systems, even when databases never fail, but they do not prove such statement. We justify such property for ROWA (Read One, Write All) protocols in this paper by using the equivalence between SI-schedules. Additionally, we show and prove that a replication protocol that uses SI replicas provides GSI if and only if it provides overall atomicity and update transactions are committed in the same order at all sites. This last property prohibits the usage of those mechanisms that exclusively order write/write conflicting transactions instead of all transactions because they do not guarantee GSI.

1 Introduction

Snapshot Isolation (SI) is a transaction isolation level introduced in [2] and implemented (using multiversion concurrency control) in several commercial database systems as Oracle, PostgreSQL, Microsoft SQL Server or InterBase. SI provides a weaker form of consistency than serializability [3]. Indeed, SI allows some read-only anomalies analyzed in [11]. Several researchers [9, 10] have recently demonstrated that, under certain conditions on the workload, transactions executing on a database with SI produce serializable histories. Nevertheless, in practice most applications run serializably under SI, including the most widely-used database benchmarks TPC-B, TPC-C, and TPC-W. These characteristics turn SI into an attractive isolation level for a database programmer because it provides sufficient data consistency for non critical applications while it maintains a good performance, since read-only activity introduces lower overheads in the protocol. This property allows read-only transactions to be never either delayed, blocked or aborted under SI, since they do not need read-locks, and they never cause update transactions to block or abort. This behavior is important for workloads dominated by read-only transactions, such as those resulting from dynamic content Web servers.

Many enterprise applications demand high availability since they have to provide continuous service to their users. For achieving such availability, the common solution consists in deploying multiple replicas

of such application. This leads also to the replication of the information being used; i.e., to managing replicated databases. The concept of Generalized Snapshot Isolation (GSI) has been recently proposed [8] in order to provide a suitable extension of conventional SI for replicated databases based on multiversion concurrency control. In GSI, transactions may use older snapshots instead of the latest snapshot required in SI. Authors of [8] outline an impossibility result which justifies the use of GSI in database replication: "*there is no non-blocking implementation of SI in an asynchronous system, even if databases never fail*". In a *non-blocking* replication protocol, transactions can start at any time without restriction or delay (including those delays produced by group communication primitives).

Concurrently with that paper, Lin et al. [18] propose a definition of One Copy Snapshot Isolation (1CSI) for ROWA (Read One Write All) protocols. From the previous impossibility result, it is not possible to obtain the given isolation level with the replication protocols stated in [18]. Thus, their protocols should be classified as GSI, instead of a strict SI for a replicated system.

In this paper, we prove that a non-blocking ROWA protocol can not implement SI. The proof is simple and it is based on a condition that restricts the transaction start time in order to provide SI. The proof forces the resulting protocol to be a blocking one. As SI is not implemented by non-blocking ROWA protocols, we also study the basic requirements that such kind of protocols must verify in order to provide GSI using SI replicas. The criteria for implementing GSI 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, read-only transactions would be able to provide different results 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 papers discussed above, ROWA certification-based replication protocols are provided. These protocols are based on the use of atomic broadcast [5] to deliver in total order the update operations of transactions for passing the certification at every database replica. The distributed protocol in [8] applies the update operations of transactions in the same total order in all replicas while the protocol in [18] allows more concurrency in the execution of the update operations of transactions at each replica, although it needs to *block* the execution of starting transactions under certain circumstances. So, these protocols comply with the requirements of GSI.

The contributions of this paper are twofold. First, a detailed formalization of the GSI definition and its requirements is presented. Second, and as a result of the first one, we provide a complete characterization of the GSI protocols that allows us to state that several kinds of optimizations are not possible if GSI must be ensured. For instance, the use of simpler types of broadcast protocols (those that do not enforce a total order), or the design of replication protocols that allow the concurrent execution of transactions without write/write conflicts (and their application at remote sites in different orders) and that never block the start of a transaction, such as some implementations of O2PL [1].

The rest of the work is organized as follows. Section 2 introduces the concept of multiversion histories based on [3]. Sections 3 and 4 give the concepts of Snapshot Isolation and Generalized Snapshot Isolation, using a similar notation to that of [8]. In Section 5, the structure of ROWA protocols is introduced. Conditions for One Copy Snapshot Isolation and One Copy Generalized Snapshot Isolation are introduced in Sections 6 and 7 respectively. Finally, conclusions end the paper.

2 Multiversion Histories

In the following, we define the concept of multiversion history for committed transactions using the theory provided in [3]. To this end, we first define the basic building blocks for our formalizations, and then the different definitions and properties will be shown.

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 concurrently executed within the *context* of their corresponding transactions. Thus, a multiversion history generalizes a history where the database items are versioned.

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. Each T_i's write operation on item X is denoted $W_i(X_i)$. A read operation on item X is denoted $R_i(X_j)$ stating that T_i reads the version of X installed by T_j. 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[1]. The properties studied in this paper only require to deal with committed transactions.

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 it is an *update* one, otherwise.

Let $T = \{T_1, ..., T_n\}$ be a set of *committed* transactions, where the operations of T_i are ordered by $\prec_{T_i}$. The last operation of a transaction is the commit operation. To process a transaction $T_i \in T$, a multiversion 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 $T_j \in T$. Each C_i remains untouched.

Once presented the basic elements relevant for our discussion, we define the Multiversion History as follows:

Definition 1. *A Complete Committed Multiversion (CCMV) history H over T is a partial order with ordering relation $\prec$ where:*

1. $H = h(\bigcup_{T_i \in T} T_i)$ *for some translation function* h.
2. $\prec \supseteq \bigcup_{T_i \in T} \prec_{T_i}$
3. $R_i(X_j) \in H, i \neq j \Rightarrow W_j(X_j) \in H \wedge C_j \prec R_i(X_j)$

In the previous Definition 1 the first condition indicates that each operation submitted by a transaction is mapped into an appropriate multiversion operation. The second one states that the CCMV history preserves all orderings stipulated by transactions, whilst the last condition says that if a transaction reads a concrete version of a data item, it was written by a transaction that committed before the item read.

Definition 1 is more specific than the one stated by Bernstein [3], since the former explicitly includes committed transactions. Hence, a new version may not be read until the transaction that installed the new version commits.

In general, two histories over the same set of transactions are *view equivalent* [3] if they contain the same operations, have the same *reads-from* relations, and produce the same final writes. The notion of equivalence of CCMV histories reduces to a simple condition, if the following *reads-from* relation is used: T_i *reads* X *from* T_j in a CCMV history H, if $R_i(X_j) \in H$.

Let H and H' be two CCMV histories over the same set of committed transactions T. Both of them have the same writes, moreover all write operations are final as all versions they produce are different. If $R_i(X_j) \in H$ and $R_i(X_j) \in H'$ then they will have the same *reads-from* relations. Thus, two CCMV histories H and H' are equivalent, denoted as $H \equiv H'$, if and only if they have the same operations.

In following sections, we use the following conventions:

- $T = \{T_1, ..., T_n\}$ the set of committed transactions.
- Any history H is a CCMV history over T.
- For each item $X \in DB$:
  $Ver(X, H) = \{X_j : W_j(X_j) \in H\} \bigcup \{X_0\}$ is the set of versions of the data item X installed in H, being X_0 its initial version.

3 Snapshot Isolation

In SI, reading from a snapshot means that a transaction T_i sees all the changes made by transactions that committed before the transaction started its first operation. The results of its writes are installed when the

[1] In fact, they can be removed using local variables in the program of the transaction [21].

transaction commits. However, a transaction T_i successfully commits if and only if there is not a concurrent transaction T_k that has already committed and some of the written items by T_k are also written by T_i. From our point of view, histories generated by a given concurrency control providing SI may be interpreted as multiversion histories with time restrictions.

Let H be a history and t: $H \rightarrow \Re^+$ a mapping such that it assigns to each operation $op \in H$ its real time occurrence $t(op) \in \Re^+$, verifying: $op \prec op'$ in $H \Rightarrow t(op) < t(op')$. The mapping $t()$ totally orders all operations of H, and the total order $<$ is compatible with the partial order $\prec$. For simplicity, we assume different times for different operations; that is, $t(op) = t(op') \Leftrightarrow op = op'$. The pair (H, t) defines a schedule of H, and it is denoted H_t. It is clear that each compatible mapping with the partial order of the history determines the obtained schedule. On the sequel we consider any schedule H_t as a schedule of a history H.

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 $\prec_{T_i}$.

In the following, we formalize the concept of snapshot of the database. Intuitively it comprises the latest version of each data item. A sample of a SI-schedule might be: $b_1 r_1(x_0)\, w_1(x_1)\, c_1\, b_2\, r_2(x_1)\, r_2(z_0)\, b_3 r_3(y_0)\, w_3(x_3)\, c_3\, w_2(y_2)\, c_2$.

As this example shows, each transaction is able to read the latest committed version of each item it accesses. 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 Y since no update of such item is seen by T_3. This is true despite transactions T_2 and T_3 are concurrent and T_2 updates X, because the snapshot taken for T_3 is previous to the beginning of T_2. This provides the basis for defining what a snapshot is.

Definition 2. *The snapshot of the database DB at time $\tau \in \Re^+$ for a schedule H_t, is:*

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

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

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

From the previous definition, it is simple 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 the concept of SI-schedule, we utilize a slight variation of the predicate *impacts* for update transactions presented in [8]: "Two transactions T_j, $T_i \in T$ impact at time $\tau \in \Re^+$ in a schedule H_t (denoted T_j *impacts* T_i *at* τ) if the predicate $WS_j \bigcap WS_i \neq \emptyset \wedge \tau < c_j < c_i$ holds."

Definition 3. *A schedule H_t is a SI-schedule if and only if for each $T_i \in T$:*

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

The previous definition is directly inspired from [8]. Its first condition states that all the versions read by a transaction T_i are obtained from $Snapshot(DB, H_t, b_i)$; that is, are obtained from the snapshot of the database DB at the time the transaction starts its first operation. The second condition states that any pair of transactions T_j and T_i, writing over some common data items, can not overlap their time intervals $[b_i, c_i]$ and $[b_j, c_j]$. In other words, they have to be executed one after the other.

Other definitions of SI have been provided in the literature. It was firstly introduced as a multiversion concurrency control in [2]. It states that transactions read operations obtain data versions committed when the transaction started. Read operations never blocked as long as the snapshot can be maintained. On the other hand, transaction's writes are reflected in its snapshot. Concurrent updates are invisible to the transaction. In order to prevent lost updates [2], it applies the *First-Committer-Wins* rule. A transaction will successfully commit only if no other transaction has already committed writes to items that the transaction intends to write. A similar definition to [2] has also been used in [10, 15, 18, 25]. In [8], where we have taken most of the notations of our work, the GSI concept is introduced (see Section 4) and by means of the notion of impacting transactions with the read and commit rules, it can be inferred SI (denoted as Conventional SI, CSI, in [8]). The definition of SI-schedule is also introduced in [18] as an execution allowed by a SI scheduler, but the notion of latest version is not explicitly introduced in that definition.

In the previous Section, the concept of view equivalence between two histories has been introduced. We explore now a notion of equivalence between SI-schedules.

Definition 4. *Let H_t and $H'_{t'}$ be two SI-schedules. H_t is SI-equivalent to $H'_{t'}$, denoted $H_t \equiv_{SI} H'_{t'}$, if and only if for any T_j, $T_i \in T$ the following conditions hold:*

1. *If $WS_i \bigcap WS_j \neq \emptyset$: $c_i < c_j$ in $H_t \Leftrightarrow c'_i < c'_j$ in $H'_{t'}$*
2. *If $WS_i \bigcap RS_j \neq \emptyset$: $c_i < b_j$ in $H_t \Leftrightarrow c'_i < b'_j$ in $H'_{t'}$*

The first condition indicates that transactions with write/write conflicts must be committed in the same order in both schedules; and the second condition states that in the case of a transaction that reads a version installed by a previous transaction in one of the schedules, the same version will be read in the other schedule. Definition 4 of SI-equivalence is obtained from [18]. It is based on the fact that both schedules are SI-schedules and does not use the concrete operations in the histories; it only uses the definitions of items to be read or written by the transactions. This motivates the next property, stating that if two SI-schedules are SI-equivalent, all transactions contained in them have read and written the same item versions in both schedules. As a result, both schedules are also equivalent (in a general sense, as defined at the end of Section 2).

Property 1. *Let H_t and $H'_{t'}$ be two SI-schedules. If $H_t \equiv_{SI} H'_{t'}$ then $H \equiv H'$.*

Proof. Let $R_j(X_i) \in H$, H_t is a SI-schedule, thus Definition 3 states that $X_i = latestVer(X, H_t, b_j)$. By Definition 2, $c_i < b_j$ and $\nexists X_k \in Ver(X, H)$: $c_i < c_k < b_j$. Assume $c'_i < c'_k < b'_j$ in $H'_{t'}$ and $X_k \in Ver(X, H')$. In that case, X_i is not the latest version of X for the transaction T_j in $H'_{t'}$; and, as the RS_j is the same for the transaction T_j in H and H', it reads a different version of X in H and H'.
By first condition in Definition 4, $c'_i < c'_k \Rightarrow c_i < c_k$ in H_t, and by second condition in Definition 4, $c'_k < b'_j \Rightarrow c_k < b_j$ in H_t. As the WS_k is the same for the transaction T_k in H_t and $H'_{t'}$, $X_k \in Ver(X, H)$. Therefore, $\exists X_k \in Ver(X, H)$: $c_i < c_k < b_j$. By contradiction, $X_i \neq latestVer(X, H_t, b_j)$. In conclusion, for any $R_j(X_i) \in H$, the statement $R_j(X_i) \in H'$ holds.
Consequently, T_j reads the same versions in H and H', thus it produces the same writes in both histories. Considering $W_j(X_j) \in H$, and since the WS_j is the same for the transaction T_j in H' and H, then $W_j(X_j) \in H'$.
Thus, H and H' have the same operations, and as showed two CCMV histories are equivalent if they have the same set of operations. Therefore, we conclude that $H \equiv H'$. □

SI-equivalence allows SI-schedules to differ in the time occurrence of operations, but it has to maintain a relative order among certain key operations of transactions: commit and first operations. However, conditions in Definition 4 can not be used to show if an arbitrary schedule $H'_{t'}$ is equivalent to a given SI-schedule H_t. This is because in a multiversion history a transaction may read any available version of a data item, and conditions in Definition 4 do not restrict that fact.

It is simple to show that if a concurrency control algorithm returns to a transaction the current snapshot at the time of its first read operation and the algorithm updates the DB at commit time if no transaction impacts with it (or in the contrary case, aborts it), then the algorithm produces SI-schedules.

4 Generalized Snapshot Isolation

The concept of Generalized Snapshot Isolation (GSI) was firstly applied to replicated databases in [8]. A hypothetical concurrency control algorithm could have stored some past snapshots. A transaction may receive a snapshot that happened in the system at the time of its first operation, and the algorithm may commit the transaction if no other transaction impacts with it from that past snapshot. Thus, in GSI, 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.

Definition 5. *A schedule H_t is a GSI-schedule if and only if for each $T_i \in T$ there exists a value $s_i \in \Re^+$ such that $s_i \leq b_i$ and:*

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

If for all $T_i \in T$, $s_i = b_i$, then H_t is a SI-schedule. Thus, Definition 5 includes as particular case Definition 3. A simple observation of the definition concludes that if $s_i < b_i$ for a $T_i \in T$ such that $RS_i \neq \emptyset$ then there exists 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 time b_i. There was a transaction T_k with $W_k(X_k) \in H$ such that $s_i < c_k < b_i$. Condition (2) also establishes that the time intervals $[s_i, c_i]$ and $[s_j, c_j]$ do not overlap if $WS_i \bigcap WS_j \neq \emptyset$.

The value s_i in Definition 5 plays the same role as b_i in Definition 3. Thus, it is possible to think that if the operations in the GSI-schedule obtained from the history H had been 'on time' then the schedule would have been a SI-schedule.

Let's see an example of how a GSI-schedule can be transformed into a SI-schedule. Suppose the following GSI-schedule: $b_1\, r_1(x_0)\, w_1(x_1)\, c_1\, b_2\, r_2(x_0)\, r_2(z_0)\, b_3\, r_3(y_0)\ w_3(x_3)\, c_3\, w_2(y_2)\ c_2$.

In this schedule, transaction T_2 reads x_0 after the commit of T_1 appears. This would not be correct for a SI-schedule (since the read version of X is not the latest one), but it is perfectly valid for a GSI-schedule, taken the time point of the snapshot provided to T_2 (i.e. s_2) previous to the commit of T_1, as it is shown: $b_1\, r_1(x_0)\, \mathbf{s_2}\, w_1(x_1)\, c_1\, b_2\, r_2(x_0)\, r_2(z_0)\ b_3\, r_3(y_0)\, w_3(x_3) c_3\ w_2(y_2)\, c_2$.

Thus, to turn this GSI-schedule into a SI-schedule, it is just needed to move the beginning of T_2 back to s_2, and consequently, the resulting schedule will be a SI-schedule: $b_1\, r_1(x_0)\, \mathbf{b_2}\, w_1(x_1)\, c_1\, r_2(x_0)\, r_2(z_0)\, b_3\, r_3(y_0)\, w_3(x_3)\, c_3\ w_2(y_2)\, c_2$.

However, this schedule does not fit the definition of b_i, which was described as the time of the first operation a transaction performs. Thus, such first operation of transaction T_2 must be also moved in the SI-schedule, resulting in the following: $b_1\, r_1(x_0)\, \mathbf{b_2}\, \mathbf{r_2(x_0)}\, w_1(x_1)\, c_1\, r_2(z_0)\, b_3\, r_3(y_0)\, w_3(x_3)\, c_3\ w_2(y_2)\, c_2$.

The following property describes the previous transformation in a formal way:

Property 2. *Let H_t be a GSI-schedule. There is a mapping $t' : H \rightarrow \Re^+$ such that $H_{t'}$ is a SI-schedule.*

Proof. Let T_i be a transaction with $RS_i \neq \emptyset$ and $s_i < b_i$ in H_t. In order to make the proof simple we consider transactions in which all read operations are done before any write operation. Let $R_i(X_{j_1})...R_i(P_{j_p})...R_i(Z_{j_z})$ be the sequence of read operations in the same order imposed by $\prec_{T_i}$. Let T_{j_s} be the first transaction with $W_{j_s}(S_{j_s}) \in H$ such that $s_i < c_{j_s} < b_i$ and $S \in RS_i$. We move the time occurrence of the read operations of the transaction T_i in the same order $\prec_{T_i}$ as follows: $t'(R_i(P_{j_p})) = s_i + \epsilon_{j_p}$ where $\epsilon_{j_1} = 0$ for the first read, and $\sum_{j_p} \epsilon_{j_p} < c_{j_s} - s_i$. For the rest of operations $op \in H$, $t'(op) = t(op)$. It is simple to show that this new mapping is compatible with $\prec$ of H.

For this transaction T_i, $\neg(T_j\ impacts\ T_i\ at\ b'_i)$ in $H_{t'}$ because $\neg(T_j\ impacts\ T_i\ at\ s_i)$ for every $T_j \in T$ and being $b'_i = s_i$. Only the read operations of T_i have been moved; the commit operations, that may modify the predicate *impacts*, have not changed their time occurrences. In $H_{t'}$, for each $R_i(P_{j_p})$ of T_i, $P_{j_p} = latestVer(P, H_{t'}, b'_i)$. Again, $b'_i = s_i$, and $P_{j_p} = latestVer(P, H_t, s_i)$.

The previous process is done for any transaction T_i such that in $H_{t'}$, $s'_i < b'_i$. This process is finite and the resulting schedule is a SI-schedule. □

The next definition is the generalization of SI-equivalence (Definition 4) between GSI-schedules.

Definition 6. *Let H_t and $H'_{t'}$ be two GSI-schedules. H_t is GSI-equivalent to $H'_{t'}$ (denoted $H_t \equiv_{GSI} H'_{t'}$) if and only if, for any T_j, $T_i \in T$ the following conditions hold:*

1. *If $WS_i \bigcap WS_j \neq \emptyset$: $c_i < c_j$ in $H_t \Leftrightarrow c'_i < c'_j$ in $H'_{t'}$*
2. *If $WS_i \bigcap RS_j \neq \emptyset$: $c_i < s_j$ in $H_t \Leftrightarrow c'_i < s'_j$ in $H'_{t'}$*

If two GSI-schedules are GSI-equivalent then their CCMV histories are also view equivalent.

Property 3. *Let H_t and $H'_{t'}$ be two GSI-schedules. If $H_t \equiv_{GSI} H'_{t'}$ then $H \equiv H'$.*

Proof. This proof is the same as the proof of Property 1 if we substitute b_j by s_j, b'_j by s'_j and Conditions (1) and (2) of Definition 4 by Conditions (1) and (2) of Definition 6 respectively. □

We remark that the definition of GSI-equivalence allows a GSI-schedule to be equivalent to a SI-schedule, but the reverse case is, in general, not true. In particular the SI-schedule $H_{t'}$ obtained in Property 2 for H_t satisfies $H_t \equiv_{GSI} H_{t'}$.

Finally, it is also important to note that GSI-schedules allow read-only transactions to obtain versions of their accessed items arbitrarily old. This could be an inconvenience for many applications, since the freshness of the accessed data is unknown. An appropriate generalization of Definition 5 would include a third property of a GSI-schedule, in order to bound the freshness of the snapshot provided to the transaction, in terms of a *distance function* d:

$$3. \quad d(Snapshot(s_i), Snapshot(b_i)) \in [0, k_i]$$

The above mentioned *distance function* d could be defined in a variety of ways, ranging from a time-based one (e.g. $d(Snapshot(s_i), Snapshot(b_i)) = b_i - s_i$) to more complex value-based specifications, using the number of changed items, or even the relative value change on items read by a transaction.

5 The ROWA Strategy

The GSI concept is particularly interesting in replicated databases, since many replication protocols execute each transaction initially in a delegate replica, propagating later its updates to the rest of replicas. This means that transaction writesets cannot be immediately applied in all replicas at a time and, due to this, the snapshot being used in a transaction might be "previous" to the one that (regarding physical time in a hypothetical centralized system) would have been assigned to it. So, in this Section we consider a distributed system that consists of m sites. $I_m = \{1..m\}$ is the set of site identifiers. Sites communicate among them by reliable message passing. We make no assumptions about the time it takes for sites to execute and for messages to be transmitted. We assume a system free of failures. Each site k runs an instance of the database management system and maintains a copy of the database DB. We will assume that each database copy, DB^k with $k \in I_m$ provides the Snapshot Isolation consistency 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.

The ROWA [12] strategy[2] is quite general and replication protocols implementing this strategy will vary in their concrete implementation [1, 4, 6, 7, 13, 14, 16, 17, 18, 20, 22, 23, 24, 25]. The ROWA approach may range according to the next two parameters: *when* updates take place, either before committing the transaction (*eager*) or after (*lazy*); and, *where* updates take place, either each database object has a primary replica where all updates are initially applied, propagating them to the secondary replicas (*primary copy*) or each replica may accept updates (*update everywhere*).

An archetypal example of a replication protocol following the ROWA strategy is as follows: once all operations of a transaction have been locally applied, the writeset is collected and sent to the rest of replicas in order to commit the transaction. Atomic broadcast based replication protocols [16, 17, 22, 25, 18] ensure that all replicas receive writesets in the same order. When a writeset is delivered to a replica, it is firstly checked against local conflicting transactions. As a result of this, the writeset may be applied (aborting all local conflicting transactions) or not. In other works, either priorities are used to avoid the latency introduced by the atomic broadcast [1] or by epidemic propagation using vector clocks [13]. However, both approaches [1, 13] need a Two Phase Commit protocol or a quorum [13] for the commitment of a transaction. In all cases, a submitted transaction may become blocked as a consequence of the DBMS activity. Hence, a "*non-blocking*" ROWA strategy is defined as the one where a transaction never becomes blocked by the replication protocol during its execution at a replica. Finally, if we assume that every transaction is going to be committed at every site, we consider that is committed as soon as it has been firstly committed at any replica.

The ROWA strategy 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 $WS_i^{site(i)} = WS_i$; for the rest of the transactions, T_i^k, $k \neq site(i)$, $RS_i^k = \emptyset$ and $WS_i^k = WS_i$. An update transaction reads at one site and writes at every site, while a read-only transaction only exists at its local site. Without generalization loss, in the rest of the paper we only consider update transactions with non-empty readsets.

[2]Since we have assumed a system free of failures, we only consider a ROWA strategy. Otherwise, a ROWAA approach is needed, i.e., writes will only be applied on the available replicas, but all our discussion is orthogonal to failures and can be seamlessly extended to a system where failures might arise.

Let $T^k = \{T_i^k : i \in I_n\}$ be the set of transactions submitted at each site $k \in I_m$ for the set T. Some of these transactions are local at k while others are remote ones. Every submitted transaction will commit at all replicas or at none if we want to maintain the full replicated feature in the system.

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

We consider that each DB^k provides SI. In the considered distributed system there is not a common clock or a similar synchronization mechanism. However, we can use a real time mapping $t : \bigcup_{k \in I_m}(H^k) \rightarrow \Re^+$ 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 $k \in I_m$. Under this mapping, each DB^k generates SI-schedules.

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

In order to study the level of consistency implemented by a non-blocking ROWA protocol is necessary to define the one copy schedule (1C-schedule) obtained from the schedules at each site. 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 $t()$.

Definition 7. *Let $T = \{T_i : i \in I_n\}$ be the set of submitted transactions to a replicated database system with a non-blocking ROWA strategy that verifies Assumption 1 and Assumption 2. Let $S = \bigcup_{k \in I_m}(H^k)$ be the set formed by the union of the CCMV histories H^k over $T^k = \{T_i^k : i \in I_n\}$. And let $t : S \rightarrow \Re^+$ be the mapping that totally orders the operations in S.*

The 1C-schedule, $H_{t'} = (H, t' : H \rightarrow \Re^+)$, is built from S and $t()$ 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*
   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*
   $t'(C_i) = t(C_i^{min(i)})$

As $t'()$ receives its values from $t()$, we write, H_t instead $H_{t'}$. 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 $k \neq site(i)$, $b_i^{site(i)} < b_i^k$. The 1C history H, that is formed by the operations over the logical DB, is also a CCMV history over T. We prove this fact informally. By the renaming (3) in Definition 7, each transaction T_i, has its operations over the data items in RS_i and WS_i, and $\prec_{T_i}$ is trivially maintained in a partial order $\prec$ for H, because H_t contains the local operations of $T_i^{site(i)}$. H is also formed by committed transactions, under Assumption 1; for each T_i, $C_i \in H$. Finally, if $R_i(X_j) \in H$, then $R_i(X_j)^{site(i)} \in H^{site(i)}$. As $H^{site(i)}$ is a CCMV history over $T^{site(i)}$ then $C_j^{site(i)} \prec R_i(X_j)^{site(i)}$. By defining $C_j^{min(j)} \prec C_j^{site(i)}$ in S then $C_j^{min(j)} \prec R_i(X_j)^{site(i)}$ and so $C_j \prec R_i(X_j)$. Thus H can be defined as a CCMV history over T.

Condition (2) on Definition 7 ensures that a transaction is committed as soon as it has been committed at the first replica. Finally, no restriction about the beginning of a transaction is imposed in this definition. Hence, this definition is valid for the most general case of non-blocking protocols.

Although Assumptions 1 and 2 are included in Definition 7, they do not guarantee that the obtained 1C-schedule is a SI-schedule. This is best illustrated in the following example, where it is shown how the 1C-schedule may be built from each site SI-schedules. In this example two sites and the next set of transactions are considered:

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

Figure 1 illustrates the mapping described in Definition 7 for building a 1C-Schedule from the SI-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 SI-schedules, i.e. transactions read the latest version of the committed data at each site. The 1C-schedule is obtained from Definition 7. 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.

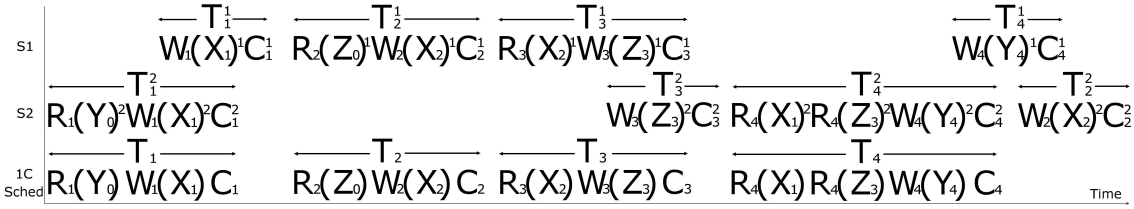

Figure 1: Execution not providing SI nor GSI.

In the 1C-schedule of Figure 1, 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 SI nor GSI.

6 One Copy Snapshot Isolation Schedules

The 1C-schedule H_t obtained in Definition 7 is a SI-schedule if it verifies the conditions given in Definition 3. The question is what conditions local SI-schedules, H_t^k, have to verify in order to guarantee that H_t is a SI-schedule. Definition 4 of SI-equivalence provides a starting point because if H_t is a SI-schedule, it would be SI-equivalent to each SI-schedule H_t^k. Taking into account the first condition of SI-equivalence in Definition 4, we consider a kind of ROWA protocols that guarantee the same total order of the commit operations for the transactions with write/write conflicts at every site.

Under the next assumption, it is ensured that conflicting transactions are executed as stated in Property 4.

Assumption 3 (Total order of conflicting transactions)**.** *For each pair T_i, $T_j \in T$ with $WS_i \bigcap WS_j \neq \emptyset$: $c_i^k < c_j^k$ holds for all SI-schedules H_t^k with $k \in I_m$.*

Property 4. *Under Assumption 3, the 1C-schedule H_t verifies that For each pair $T_i, T_j \in T$: $\neg(T_j \; impacts \; T_i \; at \; b_i)$.*

Proof. By Assumption 2, at any site $k \in I_m$, for each pair $T_j^k, T_i^k \in T^k$: $\neg(T_j^k \; impacts \; T_i^k \; at \; b_i^k)$. That is, $WS_j^k \bigcap WS_i^k = \emptyset \vee \neg(b_i^k < c_j^k < c_i^k)$.
(a) If $WS_j^k \bigcap WS_i^k = \emptyset$, by definition of T_j and T_i, $WS_j \bigcap WS_i = \emptyset$. Then, $\neg(T_j \; impacts \; T_i \; at \; b_i)$.
(b) Let $WS_j^k \bigcap WS_i^k \neq \emptyset$. Again, by definition of T_j and T_i, $WS_j \bigcap WS_i \neq \emptyset$. Hence, either $\neg(T_j^k \; impacts \; T_i^k \; at \; b_i^k)$ or $\neg(T_i^k \; impacts \; T_j^k \; at \; b_j^k)$. Thus, $c_i^k < b_j^k$ or $c_j^k < b_i^k$ holds. By Assumption 3, $c_i^k < c_j^k$ for all sites $k \in I_m$. Thus, $c_i^k < b_j^k$ for all $k \in I_m$. In particular, $c_i^{site(j)} < b_j^{site(j)}$. By definition of H_t: $c_i < c_j$ and $c_i \leq c_i^{site(j)} < b_j$ holds.
Suppose that T_j *impacts* T_i *at* b_i in H_t. That is, $WS_j \bigcap WS_i \neq \emptyset$ and $b_i < c_j < c_i$. A contradiction with $c_i < c_j$ is obtained. Therefore, $\neg(T_j \; impacts \; T_i \; at \; b_i)$.
Analogously, if T_i *impacts* T_j *at* b_j in H_t. That is, $WS_j \bigcap WS_i \neq \emptyset$ and $b_j < c_i < c_j$. A contradiction with $c_i < b_j$ is obtained again, and therefore, $\neg(T_i \; impacts \; T_j \; at \; b_j)$. □

However, the execution of write/write conflicting transactions in the same order at all sites does not offer SI nor GSI, as it has been shown in the example of Figure 1. In this example only T_1 and T_2 had

write/write conflicts and they have been executed in the same order at every site. Therefore, it is needed that a transaction reads the latest installed version in the system of a data item.

Assumption 4 (Latest-version read). *For each pair of transactions T_i, $T_j \in T$ with $WS_j \bigcap RS_i \neq \emptyset$: they verify that if $c_j < b_i$ in H_t then $c_j^{site(i)} < b_i^{site(i)}$ in $H_t^{site(i)}$.*

Under Assumptions 3 and 4 it is easy to proof the next Theorem. It states that the 1C-schedule is a SI-schedule.

Theorem 1. *Under Assumption 3 and Assumption 4, the 1C-Schedule H_t is a SI-schedule.*

Proof. By Property 4 the condition (2) in Definition 3 of SI-schedule is verified for H_t. We only need to prove the statement:
"$if\ R_i(X_j) \in H\ then\ X_j = latestVer(X, H_t, b_i)$".
Suppose $R_i(X_j) \in H$ and $X_j \neq latestVer(X, H_t, b_i)$. There is a version X_r installed by some transaction T_r such that $c_j < c_r < b_i$.
If $R_i(X_j) \in H$, by Definition 7 of H_t, $R_i(X_j)^{site(i)} \in H^{site(i)}$. From Assumption 2, it is always satisfied that $X_j^{site(i)} = latestVer(X^{site(i)}, H_t^{site(i)}, b_i^{site(i)})$.
Also, Assumption 1 ensures that $C_r^{site(i)} \in H^{site(i)}$ and $X_r^{site(i)} \in Ver(X^{site(i)}, H^{site(i)})$. As $X \in WS_j \bigcap WS_r$, by Assumption 3, if $c_r^{site(i)} < c_j^{site(i)}$ then $c_r < c_j$. Since this contradicts the initial supposition, we have that $c_j^{site(i)} < c_r^{site(i)}$.
Finally, $X \in WS_r \bigcap RS_i$ and $c_r < b_i$ in H_t by the supposition. Taken into account Assumption 4, if $c_r < b_i$ in H_t then $c_r^{site(i)} < b_i^{site(i)}$.
In conclusion, $X_r^{site(i)} \in Ver(X^{site(i)}, H^{site(i)})$ and $c_j^{site(i)} < c_r^{site(i)} < b_i^{site(i)}$. Therefore, $X_j^{site(i)} \neq latestVer(X^{site(i)}, H_t^{site(i)}, b_i^{site(i)})$ and $H_t^{site(i)}$ is not a SI-schedule against Assumption 2. The Theorem holds. □

It is easy to show that every 1C-schedule that is a SI-schedule satisfies Assumption 4. Thus, Assumption 4 is a sufficient and necessary condition while Assumption 3 is a sufficient condition. In the ROWA protocols considered in Section 5, Assumption 4 emphasizes that a transaction must see the latest installed version in the system. In other words, a transaction must remain blocked until Assumption 4 becomes true. As the considered ROWA protocol only sends the write set to remote sites, it can not be checked whether another site has installed a new version. As a straight consequence of this, it is not possible to abort the transaction violating the SI level. Thus, only a blocking ROWA protocol may obtain the SI level.

One approach to obtain SI is by sending the read and write sets of transactions to all sites in order to know if some transaction has missed a more recent version. Sending the read set to all sites is costly (leading to lower performance, poor scalability and higher abortion rates) but, on the other hand, stronger isolation levels than SI may be obtained, more precisely serializable as it has been pointed out in [8]. Another alternative approach is to atomic broadcast the transaction identifier before its execution at its local replica to all available nodes and delay its execution until the message is delivered.

7 One Copy Generalized Snapshot Isolation Schedules

If we assume a total order of the commitment of transactions at all sites, this will imply that each DB^k installs in the same total order the same snapshots. Under this assumption, a transaction may locally see an older snapshot as long as it does not impact with other transactions. In the following, we assume this total order of snapshots' installation. Under this assumption, it can be shown (Theorem 2) that the 1C-schedule obtained with SI replicas guarantees the conditions to become a GSI-schedule.

Assumption 5 (Total order of committed transactions). *For each pair T_i, $T_j \in T$: $c_i^k < c_j^k$ holds for all SI-schedules H_t^k with $k \in I_m$.*

Theorem 2. *Under Assumption 5, the 1C-schedule H_t is a GSI-schedule.*

Proof. The schedule H_t of the history H is a GSI-schedule if and only if for each $T_i \in T$ there exists a value $s_i \in \Re^+$ with $s_i \leq b_i$ and:

1. If $R_i(X_j) \in H$, then $X_j \in Snapshot(X, H_t, s_i)$.
2. For each pair $T_j, T_i \in T$: $\neg(T_j \; impacts \; T_i \; at \; s_i)$

Let us examine each case in detail:

Suppose $R_i(X_j) \in H$, then $R_i(X_j)^{site(i)} \in H^{site(i)}$. Thus, by Assumption 2, it is always satisfied that $X_j^{site(i)} = latestVer(X^{site(i)}, H_t^{site(i)}, b_i^{site(i)})$.
Let $T_{p_0}^{site(i)}$ be a transaction with $R_i(Y_{p_0})^{site(i)} \in H^{site(i)}$ for some item $Y^{site(i)} \in DB^{site(i)}$ and $c_j^{site(i)} < c_{p_0}^{site(i)}$. The time $c_{p_0}^{site(i)}$ defines the last time in $H_t^{site(i)}$ from which the transaction $T_i^{site(i)}$ no longer reads a version of a data item. By Assumption 1, $T_{p_0} \in T$ and $R_i(Y_{p_0}) \in H$.
We first prove that $\nexists T_r \in T$: $X_r \in Ver(X, H) \wedge c_j < c_r < c_{p_0}$. One can note that if this property is false then the version X_r will be more up-to-date than X_j when T_i read Y_{p_0}. The 1C-schedule H_t will not be a GSI-schedule nor SI-schedule. The proof is done by contradiction in a very simple way from Assumption 5. Suppose, there exists such a transaction $T_r \in T$ with $X_r \in Ver(X, H)$ and $c_j < c_r < c_{p_0}$. If $j = p_0$, then there is not such a $T_r \in T$. If $j \neq p_0$, by the total order Assumption 5 and Assumption 1: $c_j^{site(i)} < c_r^{site(i)} < c_{p_0}^{site(i)}$. Consequently, $X_j^{site(i)} \neq latestVer(X^{site(i)}, H_t^{site(i)}, b_i^{site(i)})$.
It is important to note that $c_{p_0}^{site(i)}$ defines the moment where $T_i^{site(i)}$ reads the latest version for $H_t^{site(i)}$. Hence, c_{p_0} will define for H_t the time instant of T_i's snapshot. If there exists a transaction T_{p_1} with $WS_{p_1} \bigcap RS_i \neq \emptyset$, then T_i will not see the versions installed by T_{p_1}. As a result, we have that $b_i^{site(i)} < c_{p_1}^{site(i)}$.
In the following, we define the possible time interval for s_i so that condition (1) of a GSI-schedule holds. Assume that there exists a subset of transactions $\{T_{p_x} : x = 0, .., L\} \subseteq T$ such that in H_t: $c_{p_0} < c_{p_1} < ... < c_{p_L} < b_i$ with $WS_{p_x} \bigcap RS_i \neq \emptyset$ for $x = 0, .., L$. We define $s_i \in (c_{p_0}, c_{p_1})$ if $L \neq 0$, or $s_i = b_i$ if $L = 0$.
For any s_i previously defined $X_j = latestVer(X, H_t, s_i)$. By the first proved property $\nexists T_r \in T$: $X_r \in Ver(X, H) \wedge c_j < c_r < c_{p_0}$. Therefore, $X_j \in Snapshot(X, H_t, s_i)$.

The second case may be split as well in two different cases: whether the snapshot has been taken at the begin of the transaction or earlier.
(a) Let $T_i \in T$ **be a transaction such that** $s_i = b_i$. Since $s_i = b_i$, we have that $L = 0$. As Assumption 5 implies Assumption 3 when $WS_j \bigcap WS_i \neq \emptyset$; then by Property 4, $\neg(T_j \; impacts \; T_i \; at \; s_i)$ holds.
(b) Let $T_i \in T$ **be a transaction with** $s_i \neq b_i$ **and** $s_i \in (c_{p_0}, c_{p_1})$. That is, $L \neq 0$. Suppose there is a transaction $T_j \in T$ such that $T_j \; impacts \; T_i \; at \; s_i$. Thus, $WS_j \bigcap WS_i \neq \emptyset \wedge s_i < c_j < c_i$. By Assumption 5, $c_j < c_i$ implies $c_j^{site(i)} < c_i^{site(i)}$. By Assumption 2, as $H_t^{site(i)}$ is a SI-schedule, the predicate $\neg(T_j^{site(i)} \; impacts \; T_i^{site(i)} \; at \; b_i^{site(i)})$ holds. Therefore, $c_j^{site(i)} < b_i^{site(i)}$. If $c_j^{site(i)} < c_{p_0}^{site(i)}$ then, by Assumption 5, $c_j < c_{p_0} < s_i$ holds in H_t. A contradiction with the supposition. Then, $c_{p_0}^{site(i)} < c_j^{site(i)}$ and $c_{p_0} < c_j$. By the definition of c_{p_0} and c_{p_1}, since $b_i^{site(i)} < c_{p_1}^{site(i)}$, the predicate $c_{p_0} < c_j < c_{p_1}$ also holds. Redefine s_i to be in the sub-interval (c_j, c_{p_1}) of (c_{p_0}, c_{p_1}). Both conditions of Definition 5 are verified. In particular, $\neg(T_j \; impacts \; T_i \; at \; s_i)$, and thus, for each possible T_j that may potentially impact with T_i at s_i we may proceed in the same way as we previously did and select the adequate sub-interval of (c_{p_0}, c_{p_1}) for s_i at which $\neg(T_j \; impacts \; T_i \; at \; s_i)$ holds. □

The assumption of total order of committed transactions has been recently used by some database replication protocols that work under SI replicas [8, 15, 18, 19]. Assumption 5 is a sufficient condition but not a necessary one, because the GSI level obtained by ROWA protocols depends on the starting point of transactions. These protocols use an atomic broadcast protocol for writeset propagation, and thus all committed transactions are totally ordered. In [18] those delivered writesets that do not intersect are allowed to proceed concurrently, and this may imply that they might be applied in different orders in different replicas. However, this creates *holes* in the writeset list being managed by this protocol and, as a result, local transactions are blocked until these holes disappear. When this happens, the effects of these concurrent writeset applications are the same as those of an application in total order, and the local transactions are then allowed to begin.

The assurance of this total order in the application of transactions in all replicas also leads to the typical certification carried out in this kind of protocols. All delivered writesets are ordered in the same way in

all replicas and a conflict checking is made in order to certify each transaction. This implies that local transactions that collide with such writesets must be aborted. Moreover, this kind of certification can be locally performed, without needing an additional voting round among replicas (the needed coordination has been achieved using the total order broadcast).

Unfortunately, in some cases a local DBMS may abort the application of some writeset (e.g., due to a deadlock, or due to a local failure), but the replication protocol has to be prepared to manage appropriately these events. To this end, such writeset applications must be retried until they are successful, and the order of their application must be correctly ensured. Some protocols have already described this kind of events (e.g., [18]).

Finally, the need of blocking local transactions when the total order cannot be ensured is a little flaw that might generate a serious performance loss for these blocked transactions. To reduce this problem, some complementary techniques must be used to ensure that writesets are quickly applied [19].

8 Conclusions

This paper studies ROWA protocols for database replication, where each replica uses a DBMS providing SI isolation level. ROWA replication protocols exclusively based on propagating the writeset of transactions may not achieve one copy SI consistency level unless they do block the beginning of transactions until they get the latest system snapshot. This potential blocking of transactions makes the main attraction of SI, the non-blocking execution of read operations, not feasible. This is the main reason for introducing GSI in database replication scenarios.

This paper establishes that the sufficient condition for obtaining a GSI consistency level is the same total order of committed transactions. This fact limits the kind of replications protocols to be implemented in a replicated setting in order to obtain GSI.

To sum up, 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. As a result, we have provided a tight theoretical basis for designing and developing future replication protocols with GSI.

Acknowledgment

This work has been supported by the Spanish Government under research grant TIC2003-09420-C02.

References

[1] José Enrique Armendáriz, José Ramón Garitagoitia, José Ramón González de Mendívil, and Francesc D. Muñoz-Escoí. Design of a MidO2PL database replication protocol in the MADIS middleware architecture. In *AINA*, volume 2, pages 861–865. IEEE-CS, 2006.

[2] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, pages 1–10. ACM Press, 1995.

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

[4] Yuri Breitbart, Raghavan Komondoor, Rajeev Rastogi, S. Seshadri, and Abraham Silberschatz. Update propagation protocols for replicated databases. In *SIGMOD*, pages 97–108, 1999.

[5] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[6] Parvathi Chundi, Daniel J. Rosenkrantz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In Stanley Y. W. Su, editor, *ICDE*, pages 469–476. IEEE-CS, 1996.

[7] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with ordering guarantees. In *ICDE*, pages 424–435. IEEE-CS, 2004.

[8] Sameh Elnikety, Fernando Pedone, and Willy Zwaenopoel. Database replication using generalized snapshot isolation. In *SRDS*. IEEE-CS, 2005.

[9] Alan Fekete. Allocating isolation levels to transactions. In *PODS*, pages 206–215, 2005.

[10] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM TODS*, 30(2):492–528, 2005.

[11] Alan Fekete, Elizabeth J. O'Neil, and Patrick E. O'Neil. A read-only transaction anomaly under snapshot isolation. *SIGMOD Record*, 33(3):12–14, 2004.

[12] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The dangers of replication and a solution. In H. V. Jagadish and Inderpal Singh Mumick, editors, *SIGMOD*, pages 173–182. ACM Press, 1996.

[13] JoAnne Holliday, Robert C. Steinke, Divyakant Agrawal, and Amr El Abbadi. Epidemic algorithms for replicated databases. *IEEE TKDE*, 15(5):1218–1238, 2003.

[14] L. Irún, F. Muñoz, H. Decker, and J. M. Bernabéu-Aubán. COPLA: A platform for eager and lazy replication in networked databases. In *ICEIS*, volume 1, pages 273–278, April 2003.

[15] B. Kemme. *Database Replication for Clusters of Workstations (ETH Nr. 13864)*. PhD thesis, ETHZ, Zurich, Switzerland, 2000.

[16] Bettina Kemme and Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. *ACM TODS*, 25(3):333–379, 2000.

[17] Bettina Kemme, Fernando Pedone, Gustavo Alonso, André Schiper, and Matthias Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE TKDE*, 15(4):1018–1032, 2003.

[18] Yi Lin, Bettina Kemme, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD*, 2005.

[19] Francesc D. Muñoz-Escoí, Jerónimo Pla-Civera, María Idoia Ruiz-Fuertes, Luis Irún-Briz, Hendrik Decker, José Enrique Armendáriz-Iñigo, and José Ramón González de Mendívil. Managing transaction conflicts in middleware-based database replication architectures. In *SRDS*, 2006. *Accepted for publication*.

[20] Esther Pacitti and Eric Simon. Update propagation strategies to improve freshness in lazy master replicated databases. *VLDB J.*, 8(3-4):305–318, 2000.

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

[22] Marta Patiño-Martínez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. MIDDLE-R: Consistent database replication at the middleware level. *ACM TOCS*, 23(4):375–423, 2005.

[23] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In *Middleware*, pages 155–174, 2004.

[24] Uwe Röhm, Klemens Böhm, Hans-Jörg Schek, and Heiko Schuldt. FAS - a freshness-sensitive coordination middleware for a cluster of OLAP components. In *VLDB*, pages 754–765, 2002.

[25] Shuqing Wu and Bettina Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *ICDE*, pages 422–433. IEEE-CS, 2005.