RELAXING CORRECTNESS CRITERIA IN DATABASE REPLICATION WITH SI REPLICAS

J. E. Armendáriz-Íñigo, J. R. González de Mendívil, J. R. Garitagoitia, J. R. Juárez-Rodríguez
Universidad Pública de Navarra, 31006 Pamplona, Spain

F. D. Muñoz-Escoí, L. Irún-Briz
Instituto Tecnológico de Informática, 46022 Valencia, Spain

Keywords: Database replication, distributed databases, snapshot isolation, read one write all, correctness criteria, formal proofs.

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. We study and formally proof a sufficient condition that replication protocols with SI replicas following the deferred update technique must obey to achieve GSI. They must provide global atomicity and commit update transactions in the very same order at all sites. However, as this is a sufficient condition, it is possible to obtain GSI by relaxing certain assumptions about the commit ordering of certain update transactions.

1 INTRODUCTION

Snapshot Isolation (SI) is the isolation level provided by several commercial database systems, such as Oracle, PostgreSQL, Microsoft SQL Server or InterBase. Transactions executed under SI allows to read from the last committed snapshot and, hence, read operations are never blocked nor conflict with any other update transaction. In order to prevent the lost update phenomenon (Berenson et al., 1995), concurrent update transactions (read-only transactions are always committed) modifying the same data item apply the *first-committer-wins* rule: only the first transaction that commits is allowed to proceed the remainder are aborted. This turns out into a nice feature because it provides sufficient data consistency (though not serializable (Fekete et al., 2005; Elnikety et al., 2005)) for non-critical applications while it maintains a good performance, since read-only transactions are neither delayed, blocked nor aborted and they never cause update transactions to block or abort. This behavior is important for workloads dominated by read-only transactions, such as those resulting from dynamic content Web servers (Plattner et al., 2008).

Many enterprise applications demand high availability since they have to provide continuous service to their users. This also implies to replicate the information being used; i.e., to manage replicated databases. The concept of Generalized Snapshot Isolation (GSI, concurrently to this a similar definition denoted as 1-copy-SI was proposed in (Lin et al., 2005)) has been recently proposed (Elnikety et al., 2005) 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 (setting up the latest snapshot in a distributed setting is not trivial). Actually, authors of (Elnikety et al., 2005) 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*" which has been formally justified in (González de Mendívil et al., 2007).

The deferred update technique (Pedone, 1999) consists in executing transactions at their delegate replicas (obtaining their corresponding snapshot) and setting up a commit ordering for update transactions which is mainly done thanks to the total order broadcast (Chockler et al., 2001). When a trans-

E. Armendáriz-Íñigo J., R. González de Mendívil J., R. Garitagoitia J., R. Juárez-Rodríguez J., D. Muñoz-Escoí F. and Irún-Briz L. (2008).
RELAXING CORRECTNESS CRITERIA IN DATABASE REPLICATION WITH SI REPLICAS.
In *Proceedings of the Third International Conference on Software and Data Technologies - ISDM/ABF*, pages 45-53
DOI: 10.5220/0001877700450053

action requests its commitment (read-only transactions are committed right away) its updates are collected and broadcast (using the total order primitive) to the rest of replicas. Upon its delivery at replicas a validation test (i.e. to detect conflicts with other concurrent transactions in the system) is performed; namely a certification test (Wiesmann and Schiper, 2005) that performs the distributed first-committer-wins rule (Elnikety et al., 2005; Lin et al., 2005) in the same way at al replicas and ensures the same order of the commit process of transactions. The main advantage of these replication protocols is that transactions can start at any time without restriction or delay.

In this paper, we formalize the requirements for achieving GSI over SI replicas using *non-blocking* protocols. Thus, 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; i.e. giving the logical vision of a one copy scheduler (1-Copy-GSI). Whereas atomicity guarantees that all replicas take the same actions regarding each transaction, so their states should be consistent, once each transaction has been terminated.

One can think that these assumptions are rather intuitive but they constitute the milestone for our contribution of the paper. It consists in somehow relaxing the assumption of the total order of committed transactions. If a protocol is not careful about that, 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. However, this optimization is important since processing messages serially as supposed for replication protocols deployed over a group communication system (Chockler et al., 2001) would result in significantly lower throughput rates. A relaxing assumption has been already presented in (Lin et al., 2005), still using the total order broadcast, it lets validated transactions to apply (and commit) transactions concurrently as long as their respective updates do not intersect. However, this protocol needs to *block* the execution of the first operation of any starting transaction until the concurrent application of transactions finishes. Thus, it is easy to see that there are multiple approaches to obtain GSI at the price of imposing certain restrictions, in particular, the need to block the start of transactions to obtain a global consistent snapshot. Finally, we take a look and discuss how to relax this last contribution, which is actually too strong, for deploying GSI non-blocking protocols.

The rest of the work is organized as follows[1]. Section 2 introduces the concept of multiversion histories based on (Bernstein et al., 1987). Sections 3 and 4 give the concepts of SI and GSI respectively. In Section 5, the structure of deferred update replication protocols is introduced. Conditions for 1-Copy-GSI is introduced in 6. We take a look at how to relax conditions for 1-Copy-GSI in Section 7. 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 (Bernstein et al., 1987). The properties studied in our paper only require to deal with committed transactions. 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. The results for this kind of transactions are seamlessly extensible to more general models. In any case, redundant operations can be removed using local variables in the program of the transaction (Papadimitriou, 1986).

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

[1]Due to space constraints, the reader is referred to (González de Mendívil et al., 2007) for a thorough explanation of the correctness proof.

(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 $T = \{T_1, \ldots, T_n\}$ 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 operations from 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 $W_i(X_i)$, and each $R_i(X)$ into $R_i(X_j)$ for some $T_j \in T$.

Definition 1. *A Complete Committed Multiversion (CCMV) history H over T is a partial order with order relation $\prec$ such that:*
(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) 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) indicates 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 if 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 (Bernstein et al., 1987), 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 the paper, we use the following conventions: (i) $T = \{T_1, \ldots, T_n\}$ is the set of committed transactions for every defined history; and, (ii) any history H is a CCMV history over T. Note that these conventions will be also applicable when a superscript is used to denote the site of the database where the history is generated.

In general, two histories $(H, \prec)$ and $(H', \prec')$ over the same set of transactions are *view equivalent* (Bernstein et al., 1987), denoted as $H \equiv H'$ 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 the simple condition, $H = H'$, if the following *reads-from* relation is used: T_i *reads* X *from* T_j in a CCMV history $(H, \prec)$, if and only if $R_i(X_j) \in H$.

3 SNAPSHOT ISOLATION

In SI reading from a snapshot means that a transaction T_i sees all the updates done by transactions that committed before the transaction started its first operation. The results of its 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 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.

Definition 2. *Let $(H, \prec)$ be a history and $t\colon H \to \mathbb{R}^+$ 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) item 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 additional 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 $\prec_{T_i}$. In the following, we formalize the concept of snapshot of the database. Intuitively, it comprises the latest version of each data item. Firstly, we will see an example of this:

Example 1. *Let us consider the following transac-*

tions 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 possible 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 example 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 $Ver(X,H) = \{X_j : W_j(X_j) \in H\} \cup X_0$, 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: $latestVer(X, H_t, \tau) = \{X_p \in Ver(X,H) : (\nexists X_k \in Ver(X,H) : c_p < c_k \leq \tau)\}$*

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 the concept of SI-schedule, we utilize a slight variation of the predicate *impacts* for update transactions presented in (Elnikety et al., 2005). Two transactions T_j, $T_i \in T$ impact at time $\tau \in \mathbb{R}^+$ in a schedule H_t, denoted T_j *impacts* T_i *at* τ, if the following predicate holds: $WS_j \bigcap WS_i \neq \emptyset \wedge \tau < c_j < c_i$.

Definition 4. *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 $X_j \in Snapshot(DB, H_t, b_i)$; and,
(2) for each $T_j \in T : \neg(T_j$ impacts T_i at $b_i)$.

Condition (1) states that all the versions read by a transaction T_i are obtained from $Snapshot(DB,\, H_t, b_i)$; that is, versions are obtained from the snapshot of the database DB at the time the transaction starts its first operation. Condition (2) 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 in a serial way. Other equivalent definitions of SI have been provided in the literature (Berenson et al., 1995; Kemme, 2000; Lin et al., 2005; Fekete et al., 2005; Elnikety et al., 2005).

4 THE GSI LEVEL

The concept of Generalized Snapshot Isolation (or GSI, for short) was firstly applied to database replication in (Elnikety et al., 2005). 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 (instead of its current snapshot as in a SI concurrency control algorithm). 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 5. *A schedule H_t is a GSI-schedule if and only if for each $T_i \in T$ there exists a value $s_i \in \mathbb{R}^+$ 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. If for all $T_i \in T$, conditions (1) and (2) hold for $s_i = b_i$ then H_t is a SI-schedule. Thus, Definition 5 includes as a particular case the Definition 4. Another observation of the definition concludes that if there exists a transaction $T_i \in T$ such that conditions (1) and (2) 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. There was a transaction T_k with $W_k(X_k) \in H$ such that $s_i < c_k < b_i$. This can be best seen in the next example.

Example 2. *The following is an example of a 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: $b_1\, R_1(X_0)\, \mathbf{s_2}\, W_1(X_1)\, c_1\, b_2\, R_2(X_0)\, R_2(Z_0)\, b_3\, R_3(Y_0)\, W_3(X_3)\, c_3\, W_2(Y_2)\, c_2$. The intuition under this schedule in a distributed system is that the message containing the modifications of T_1 (the write operation on X) would have not yet arrived to the site at the time transaction T_2 began. This may be the reason for T_2 to see this previous version of item X. The fact that GSI captures* *these delays into schedules makes attractive its usage on distributed environments.*

The value s_i in Definition 5 plays the same role as b_i in Definition 4. 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.

Example 3. *Let us use Example 2 to show how a GSI-schedule can be transformed into a SI-schedule. Thus, to turn that 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 $R_1(X_0)$ $\mathbf{b_2}$ $R_2(X_0)$ $W_1(X_1)$ 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 $t' : H \to \mathbb{R}^+$ such that $H_{t'}$ is a SI-schedule.*

This last property states that if H_t is a GSI-schedule, there will exist a $H_{t'}$, which is actually a SI-schedule, and verify the following $H_t \equiv H_{t'}$ (in the sense of view-equivalence).

5 THE DEFERRED UPDATE TECHNIQUE

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 (Lin et al., 2005; Elnikety et al., 2005; Armendáriz-Iñigo et al., 2007). 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. In this Section we consider a distributed system that consists of m sites, being $I_m = \{1..m\}$ 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[2]. 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, denoted DB^k with $k \in I_m$, provides SI (Berenson et al., 1995).

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 deferred update technique 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 $WS_i^k = WS_i$. $T_i^{site(i)}$ determines the local transaction of T_i, i.e., the transaction executed at its delegate replica or 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.

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. In the next, the Assumption 1 implies that each transaction submitted to the system either commits at all replicas or in none of them. Thus, the updates applied in a delegate replica by a given transaction are also applied in the rest of replicas. Obviously, we consider a fully-replicated system. Since only committed transactions are relevant, the histories being generated at each site should be histories over T^k, as defined above.

Assumption 1 (Atomicity). *H^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 $t : \bigcup_{k \in I_m}(H^k) \to \mathbb{R}^+$ 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 SI-schedules under the previous time mapping.

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 this kind of non-blocking protocols is necessary to define the one copy schedule (1C-schedule) obtained from the schedules at each site. In the next

[2]Otherwise, 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.

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 6 (1C-schedule)**.** *Let $T = \{T_i \colon i \in I_n\}$ be the set of submitted transactions to a replicated database system with a non-blocking deferred update 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 histories H^k over $T^k = \{T_i^k \colon i \in I_n\}$. And let $t \colon S \to \mathbb{R}^+$ be the mapping that totally orders the operations in S. The 1C-schedule, $H_{t'} = (H, t' \colon H \to \mathbb{R}^+)$, 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 of $H_{t'}$. In the 1C-schedule H_t, for each transaction T_i, is trivially verified $b_i < c_i$ because this technique 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 history over T. We prove this fact informally. By the renaming (2) in Definition 6, 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 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 history over T.

Transformation (2) on Definition 6 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 6, they do not guarantee that the obtained 1C-schedule is a SI-schedule. This is best illustrated in the following example, where it is also shown how the 1C-schedule may be built from each site SI-schedules.

Example 4. *In this example two sites (A, B) and the next set of transactions T_1, T_2, T_3, T_4 are considered: $T_1 = \{R_1(Y), W_1(X)\}, T_2 = \{R_2(Z), W_2(X)\}$, $T_3 = \{R_3(X), W_3(Z)\}$, $T_4 = \{R_4(X), R_4(Z), W_4(Y)\}$. Figure 1 illustrates the mapping described in Definition 6 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 A ($RS_2 \neq \emptyset$ and $RS_3 \neq \emptyset$) whilst T_1 and T_4 are executed at site B 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 6. For example, the commit of T_1 occurs for the 1C-schedule in the minimum of the interval between C_1^A and C_1^B and so on for the remaining transactions. 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 A). 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 1-COPY-GSI SCHEDULES

The 1C-schedule H_t obtained in Definition 6 will be a GSI-schedule if it verifies the conditions given in Definition 5. The question is what conditions local SI-schedules, H_t^k, have to verify in order to guarantee that H_t is a GSI-schedule. Taking into account the ordering of conflicting transactions in GSI-equivalence, we consider the kind of protocols that guarantee the same total order of the commit operations for the transactions with write/write conflicts at every site. 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 Example 4. Therefore, it is also necessary to consider the need of reading from a consistent snapshot from the notion of GSI-equivalence; i.e. all update transactions must be committed in the very same order at all sites. As a result, since all replicas generate SI-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 Committing Transactions)**.** *For each pair $T_i, T_j \in T$, a unique order relation $c_i^k < c_j^k$ holds for all SI-schedules H_t^k with $k \in I_m$.*

The SI-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: $c_1^k < c_2^k < \ldots < c_n^k$ for every $k \in I_m$. In the next property we are going to verify that, thanks to

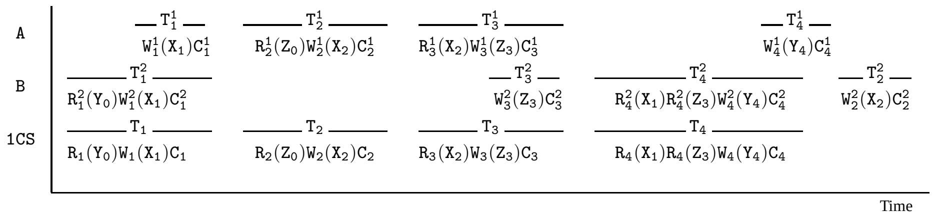

Figure 1: Replicated one-copy execution not providing CSI nor GSI.

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 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, after T_{i_0}, that verifies $WS_{i_1} \cap RS_i \neq \emptyset$ and hence modifying 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 3. *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 $c_{i_0} \leq \tau < c_{i_1} \leq b_i$.*

The aim of the next theorem is to prove that the 1C-schedules generated by any deferred update protocol that verifies Assumption 3 are actually GSI-schedules; i.e., they comply with all conditions stated in Definition 5. Whilst proving that a transaction always reads from the same snapshot in a particular time interval is easy, it is not trivial to prove that for a given transaction T_i there has not been any other transaction T_j that has impacted T_i and that has been committed whilst T_i was being executed. 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.*

This theorem formally justifies such protocols correctness and establishes that their resulting isolation level is GSI; the proof of it is given in (González de Mendívil et al., 2007). 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 the next section, we analyze how to relax this assumption while obtaining GSI schedules with non-blocking protocols.

7 RELAXING ASSUMPTIONS

Assumption 3 (Total order of committing transactions) is very strong. It forces to install the same snapshots in the same order at every replica. Thus, Theorem 1 guarantees that the 1C-schedule H_t is a GSI-schedule. On the contrary, the total order of conflicting transactions is not enough to guarantee SI nor GSI (see Example 4) and it requires a stronger condition: it is needed that the snapshot gotten by a transaction at its delegate replica matches the 1C-schedule, actually being the latter a GSI-schedule. However, this fact does not necessarily oblige each replica to install the same snapshots as in the 1C-schedule. That is, if $R_i(X_j)$ belongs to H_t then $X_j \in (Snapshot(DB, H_t, b_i) \cap Snapshot(DB, H_t^{site(i)}, b_i))$. From what it has been depicted before, it is clear that if you want to relax Assumption 3, you have to provide some property that sets a relation between the reads-from relationship of a transaction in the 1C-schedule and the reads-from relationship of the transaction local schedule at its delegate site. In the next, we provide more relaxing assumptions to obtain a 1C-schedule providing GSI.

Assumption 4. *For each pair $T_i, T_j \in T$ with $WS_i \cap WS_j \neq \emptyset$, a unique order relation $c_i^k < c_j^k$ holds for all SI-schedule H_t^k with $k \in I_m$; and, if there is some transaction $T_p \in T$ such that $c_i^k < c_p^k < c_j^k$ holds for some site $k \in I_m$ then it holds for every $k \in I_m$.*

This assumption states that between two conflicting transactions their commit ordering is the same at every site. Moreover, it also states that between both transactions, there are the same subset of committed transactions; no matter the order in which they occur.

Example 5. *Let us suppose that there are two replicas and the next set of transactions: $\{T_1, T_2, T_3, T_4, T_5, T_6, T_7\}$ with $WS_1 \cap WS_4 \neq \emptyset$, $WS_3 \cap WS_7 \neq \emptyset$ and the rest do not conflict among each other. At the first site you can find the following local SI-schedule: $c_1^1 < c_2^1 < c_3^1 < c_4^1 < c_5^1 < c_6^1 < c_7^1$ whilst at the second site the derived SI-schedule can be: $c_1^2 < c_2^2 < c_3^2 < c_4^2 < c_6^2 < c_5^2 < c_7^2$. In the latter, the* *commit ordering of transactions T_5 and T_6 is different from the scheduling of the former.*

As it may be inferred, Assumption 4 becomes Assumption 3 whenever the pattern of transactions do not allow to reorder the commit of transactions. In Example 5, it cannot happen without violating Assumption 4 the following: $c_4^2 < c_3^2$. On the other hand, taking Assumption 4 to the extreme, if all transactions do not conflict among them any committing order can be obtained at each site. To limit these situations from making their appearance, it is needed to enforce to each transaction to read from the same snapshot like for each pair of transactions $T_i, T_j \in T$ with $WS_j \setminus RS_i \neq \emptyset$: they verify that if $c_j < b_i$ in H_t then $c_j^{site(i)} < c_i^{site(i)}$ in $H_t^{site(i)}$. $WS_j \setminus RS_i \neq \emptyset$: they verify that if $c_j < b_i$ in H_t then $c_j^{site(i)} <_i^{site(i)}$ in $H_t^{site(i)}$ which is stated in the next assumption.

Assumption 5 (Compatible Snapshot Read)**.** *Let H_t be a 1C-schedule, for each $T_i \in T$ there exists $s_i \leq b_i$ such that if $R_i(X_j) \in H_t$ then $X_j \in (Snapshot(DB, H_t, s_i) \cap Snapshot(DB, H_t^{site(i)}, b_i))$.*

This last assumption means that each transaction reads data items that belong to a valid global snapshot from the 1C-schedule although their delegate site do not install the same snapshot version. On the other hand Assumption 4, it seems clear that a 1C-schedule serializes the execution of conflicting transactions.

Property 4. *Under Assumption 4, 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, $WS_j^k \cap WS_i^k = \emptyset \vee \neg(b_i^k < c_j^k < c_i^k)$.
(1) If $WS_j^k \cap WS_i^k = \emptyset$, by definition of T_j and T_i, $WS_j \cap WS_i = \emptyset$. Then, $\neg(T_j$ *impacts* T_i *at* $b_i)$.
(2) Let $WS_j^k \cap WS_i^k \neq \emptyset$. Again, by definition of T_j and T_i, $WS_j \cap WS_i \neq \emptyset$. 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 4, $c_i^k < c_j^k$ 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 in H_t. Suppose that T_j *impacts* T_i *at* b_i in H_t. That is, $WS_j \cap WS_i \neq \emptyset$ 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, $WS_j \cap WS_i \neq \emptyset$ 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* $b_j)$. ■

In the next theorem is proved that 1C-schedules generated by deferred update protocols following Assumption 4 and Assumption 5 verify Definition 5; i.e. they generate GSI schedules.

Theorem 2. *Under Assumption 4 and Assumption 5, the 1C-schedule H_t is a GSI-schedule.*

Proof: Firstly, notice that Assumption 4 implies total order of conflicting transactions. Given this total order of conflicting transactions, the 1C-schedule H_t, the 1C-schedule verifies for each $T_i \in T$ that $\neg(T_j$ impacts T_i at $b_i)$ for every $T_j \in T$. Additionally, by Assumption 5, for each $T_i \in T$, if $R_i(X_j) \in H_t$ then $X_j \in (Snapshot(DB, H_t, s_i) \cap Snapshot(DB, H_t^{site(i)}, b_i^{site(i)}))$ with $s_i \in \mathbb{R}^+$ and $s_i \leq b_i$ (recall that $b_i = b_i^{site(i)}$). This fact makes true Condition (1) in Definition 5. Therefore, if $s_i = b_i$ for every $T_i \in T$ then Condition (2) in Definition 5 trivially holds. We need to prove Condition (2) in general. Thus, consider $s_i < b_i$; there must be a transaction $T_m \in T$ such that $s_i < c_m < b_i$ and $WS_m \cap RS_i \neq \emptyset$. Let T_m be the first transaction in H_t verifying such condition. Therefore, by Assumption 1 and Assumption 2 ($H_t^{site(i)}$ is a SI-schedule), $b_i^{site(i)} < c_m^{site(i)}$ holds. As $c_m < b_i$ then $c_m < c_i$ also holds. Assume that $WS_m \cap WS_i \neq \emptyset$, if $c_i^{site(i)} < c_m^{site(i)}$ then by Assumption 4 and construction of H_t, $c_i < c_m$ leading to a contradiction. So, $b_i^{site(i)} < c_m^{site(i)} < c_i^{site(i)}$. This implies, by Assumption 2 that $WS_m \cap WS_i = \emptyset$ and T_m verifies Condition (2) in Definition 5($\neg(T_m \text{ impacts } T_i \text{ at } s_i)$).

Every transaction $T_p \in T$ such that $s_i < c_p < c_m < b_i$ verifies that $WS_p \cap RS_i = \emptyset$ since T_m is the first one such that $WS_m \cap RS_i \neq \emptyset$. So, if $WS_p \cap WS_i \neq \emptyset$ then you can find $s_i' \in \mathbb{R}^+$: $s_i < c_p < s_i' < c_m < b_i$. At s_i', Assumption 5 is verified again for T_i by Definition 3 of snapshot. Furthermore, if $c_p < b_i$ then $c_p^{site(i)} < b_i^{site(i)} < c_i^{site(i)}$ due to Assumption 4 and construction of H_t (recall that $c_= min_{k \in I_m}\{c_j^k\}$ after renaming $c_j^m in(j)$ for all T_j), $c_p < c_m < c_i$ in H_t that is a contradiction with the initial supposition of $c_m < c_p < b_i$. Thus, $WS_p \cap WS_i = \emptyset$ and Condition (2) in Definition 5 is verified for every transaction. The 1C-schedule is a GSI-schedule under the given assumptions. ■

From all discussed throughout this section, one can infer that a replication protocol that respects Assumption 4 and Assumption 4 will provide GSI to its executed transactions without needing to block transactions. The simplest, and most straightforward, solution is to define a conflict class (Patiño-Martínez et al., 2005; Amza et al., 2003) and each site is responsible for one (or several) conflict class. Thus, transactions belonging to different conflict classes will commit in any order at remote replicas while conflicting transactions belonging to the same conflict class are managed by the underlying DBMS of its delegate replica. Of course, this solution has its own pros and cons, we assume that each transaction exclusively belongs to a conflict class, i.e. no compound conflict classes, and it will read data and write data belonging to that class. However, it is a high application dependent and the granularity of the conflict class is undefined: it can range from coarse (at table level) to fine (at row level) granularity.

8 CONCLUSIONS

It has been formalized the sufficient conditions to achieve 1-copy-GSI for non-blocking replication protocols following the deferred update technique that exclusively broadcast the writeset of transactions with SI replicas. They consist in providing global atomicity and applying (and committing) transactions in the very same order at all replicas. This means that there are other means to provide GSI in a replicated setting, some come at the cost of blocking the start of transactions (Lin et al., 2005) (which goes against the non-blocking nature of SI (Berenson et al., 1995)) or by way of relaxing the total order of committed transactions given here. In particular, that between two conflicting transactions the same set of non-conflicting transactions must be committed and transactions started while applying in different order these writesets have read data items that belong to global valid versions. 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 sound theoretical basis for designing and developing future replication protocols with GSI.

ACKNOWLEDGEMENTS

This work has been supported by the EU FEDER and Spanish MEC under grant TIN2006-14738-C02.

REFERENCES

Amza, C., Cox, A. L., and Zwaenepoel, W. (2003). Conflict-aware scheduling for dynamic content applications. In *USENIX*.

Armendáriz-Iñigo, J. E., Juárez-Rodríguez, J. R., de Mendívil, J. R. G., Decker, H., and Muñoz-Escoí, F. D. (2007). *K*-bound GSI: a flexible database replication protocol. In *SAC*, pages 556–560. ACM.

Berenson, H., Bernstein, P. A., Gray, J., Melton, J., O'Neil, E. J., and O'Neil, P. E. (1995). A critique of ANSI SQL isolation levels. In *SIGMOD*, pages 1–10.

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

Chockler, G., Keidar, I., and Vitenberg, R. (2001). Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469.

Elnikety, S., Pedone, F., and Zwaenopoel, W. (2005). Database replication using generalized snapshot isolation. In *SRDS*, pages 73–84. IEEE-CS.

Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., and Shasha, D. (2005). Making snapshot isolation serializable. *ACM TODS*, 30(2):492–528.

González de Mendívil, J. R., Armendáriz-Iñigo, J. E., Muñoz-Escoí, F. D., Irún-Briz, L., Garitagoitia, J. R., and Juárez-Rodríguez, J. R. (2007). Non-blocking ROWA protocols implement GSI using SI replicas. Technical Report ITI-ITE-07/10, ITI.

Kemme, B. (2000). *Database Replication for Clusters of Workstations (Nr. 13864)*. PhD thesis, ETHZ.

Lin, Y., Kemme, B., Patiño-Martínez, M., and Jiménez-Peris, R. (2005). Middleware based data replication providing snapshot isolation. In *SIGMOD*, pages 419–430. ACM.

Papadimitriou, C. (1986). *The Theory of Database Concurrency Control*. Computer Science Press.

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

Pedone, F. (1999). *The database state machine and group communication issues (N. 2090)*. PhD thesis, EPFL.

Plattner, C., Alonso, G., and Özsu, M. T. (2008). Extending DBMSs with satellite databases. *VLDB J.*, Accepted for publication.

Wiesmann, M. and Schiper, A. (2005). Comparison of database replication techniques based on total order broadcast. *IEEE TKDE*, 17(4):551–566.