Providing Read Committed Isolation Level in Non-Blocking ROWA Database Replication Protocols*

J. M. Bernabé-Gisbert, J. E. Armendáriz-Iñigo, R. de Juan-Marín, F. D. Muñoz-Escoí

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

Abstract. Total order ROWA strategies are widely used in replicated protocol design. Normally, these protocols provide higher isolation guarantees like Serialisable or Snapshot Isolation but, for some applications, a more permissive isolation level like Read Committed fits better. Some centralised database management systems provide Read Committed as a default isolation level but in replicated systems it is rare to find proposals and systems supporting it. In this paper we extend the notion of Read Committed to a replicated environment, giving the necessary theoretical background to construct Read Committed ROWA database replication protocols.

1 Introduction

Since the starts of distributed and replicated database theory, Serialisable isolation level has been the main objective in concurrency control works. In the last half of the nineties, Snapshot Isolation (SI in the sequel) has also grown as a good isolation level in replication protocols since provides isolation guarantees near to serialisable but allows to develop better replication protocols in terms of performance due to its intrinsic optimistic orientation. Unlike serialisable, SI avoids the use of read locks, which improves performance in applications with a higher reads rate, something typical in most web applications.

However, in some applications it is required even less consistency than the one already provided by SI, since their main goal is to achieve a better performance in terms of response time. As an example, in web applications like street directories or news portals, users only can read information updated only by administrators or especial content provider users. In this case, normal users transactions are read-only and never will fall in phenomena involving their writes

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

like lost update [1] ($r_1(x)...w_2(x)...w_1(x)$, T_1 modifies x without seeing T_2 update). Seeing information a bit older or seeing updates between queries is also normal and, in some cases, even expected (for example, in a latest news list). In these environments, weaker isolation levels like Read Committed (RC in the sequel) make sense and this is the reason why it is normally supported by centralised database management systems (DBMS in the sequel). In some of them is in fact the default isolation level provided [2].

Unfortunately, few works have been done about providing RC isolation level in distributed environments ([3], [4]) and, as far as we know, only appears as a part of a more general study about isolation levels in DBMSs.

In this paper we give a new RC definition, named Loose Read Committed (LRC), between Berenson [1] and Adya [3] ones in terms of avoided phenomena. We define also Generalised Loose Read Committed (GLRC) isolation level definition as a relaxed version of LRC useful in replicated environments. We also proof the equivalence between Adya PL-2 and GLRC isolation levels and give the necessary background to provide LRC and GLRC in a replicated setting, defining the necessary concepts of *LRC-equivalence*, *GLRC-equivalence*, *one-copy LRC* and *one-copy GLRC*. At the end, we give some guidelines to provide LRC and GLRC in the most typical database replication schemes, defined in [5], giving special attention to ROWA certification and weak-voting. All our work has been influenced by previous works like [3] or [6], that were centred in SI and Generalised SI.

The rest of the paper is divided as follows. In Section 2 we give some necessary definitions used in the rest of the document. Section 3 presents LRC definition. The notions of one-copy equivalence in LRC are introduced in Section 4. Section 5 introduces GLRC and translates the concept of one-copy equivalence to this isolation level. In Section 6, we talk briefly about providing LRC and GLRC in the most used database replication protocol schemes [5]. Finally, conclusions are presented in Section 7.

2 Basic Definitions

In this work, the database is composed by a set of items. An item x can be read ($r(x)$) and written ($w(x)$).

The database is accessed by way of transactions which are issued by user applications. A transaction is composed by a sequence of read and write operations and a commit c or an abort a, which must be always the last one. Given a transaction T_i, $r_i(x)$, $w_i(x)$, c_i and a_i represent a T_i read, write, commit and abort operations, respectively. In terms of the notation used in this paper, we will use o_i to represent a T_i operation without specifying its type. A T_i read of the last modification performed by T_j on item x is represented as $r_i(x_j)$.

$r_i(x_{j,l})$ represents the read by T_i of the l-th modification of x made by transaction T_j. If no item index is specified, the read is over the last item value written by any transaction before the present read. With $w_j(x_j)$ we represent the final modification of x made by T_j and $w_j(x_{j,l})$ the l-th one. Formally:

Definition 1 (Transaction) *A transaction over a set of operations O is a total order $<$ which:*

- $c \in T \ \vee \ a \in T$
- $c \in T$ *iff* $a \notin T$
- *If* $c \in T, \forall o \neq c \in T, o < c$
- *If* $a \in T, \forall o \neq a \in T, o < a$
- *Given* $o_1, o_2 \in T, o_1 < o_2 \ \vee o_2 < o_1$

A *committed transaction* is a transaction whose final operation is a commit. In the same way, an *aborted transaction* is a transaction which ends with an abort.

Definition 2 (History) *A history H over a set of transactions $T = T_1, ..., T_n$ represents a possible execution of T. Formally, a history is a partial order $<_H$ where:*

- *For every* $T_i \in T$ *and every* $o_i \in T_i$, $o_i \in H$.
- *For every* $T_i \in T$ *and every* $o_{i1}, o_{i2} \in T_i$ *If* $o_{i1} < o_{i2} \in T_i$, $o_{i1} <_H o_{i2} \in H$.
- *If* $r_i(x_j) \in H$ *then* $w_j(x_j) \in H \wedge w_j(x_j) <_H r_i(x_j)$.

3 Read Committed Definitions

3.1 Previous Definitions

Since ANSI proposed its SQL isolation level definitions, some authors have proposed their own ones trying to eliminate ANSI weaknesses as long as its strengths are kept. One of the most referenced works is the revision made by Berenson et al. [1]. This work avoids the ambiguities of ANSI definitions at the cost of losing implementation independence since their definitions were based on locking techniques. To solve this, Adya et al. [3] proposed a new set of definitions in his thesis trying to be precise and implementation independent at the same time. To do that, Adya defines a new kind of dependency graph (named *Direct Serialisation Graph*), an extension of the *Serialisation Graph* used by Bernstein in [7] , to represent dependencies between transactions in histories. In Adya's work, the equivalent isolation level to RC was PL-2. Actually, this level is a bit more relaxed than RC because allows a transaction to see a non-committed value, only if it belongs to a transaction that will eventually commit. In PL-2, a transaction can also read older item values instead of the latest one, not like in Berenson RC definition. For example, the following history:

$w_j(x_j)\ w_k(x_k)\ r_i(x_j)\ c_j\ c_k\ c_i$

Is not a valid RC history since T_i read loses $T_k\, x$ update and the value read belongs to T_j, which has not committed at the time the read is performed.

We have taken Adya's isolation level PL-2 and redefined it avoiding the use of graphs. This allows us to differentiate two isolation levels: LRC and GLRC. The first one represents the classic isolation level supported by centralised DBMSs where a read over an item sees its last value written at the time the read is performed. In GLRC we allow a transaction to see an old value of an item so, in this case, is equivalent to Adya's PL-2. This is useful in replicated environments because a node can start a transaction before applying writesets of previous transactions already executed in other nodes ([6]). It is important to notice that LRC is a particular case of GLRC.

3.2 Loose Read Committed definition

Before introducing the LRC definition, we need to introduce some concepts. Given a history H over a set of transactions T, given $T_i, T_j, T_k \in T$ and given an item x of the database:

Definition 3 (reads from) *T_i reads from T_j if $r_i(x_j) \in H$*

Definition 4 (overwrites) *T_i overwrites T_j if $w_i(x) \in H \wedge w_j(x) \in H \wedge w_j(x) <_H w_i(x) \wedge \nexists\, w_k(x) \in H\colon w_j(x) <_H w_k(x) <_H w_i(x)$*

Definition 5 (depends on) *T_i depends on T_j if T_i reads from T_j or T_i overwrites T_j.*

Therefore, we can then establish a partial order between transactions in a history by using the *depends on* relation:

Definition 6 (dependency order) *Given a history H over a set of transactions T, given T_i, $T_j \in T$, we say that $T_j \rightarrow T_i$ if T_i depends on T_j. By the transitive rule, if $T_j \rightarrow T_i$ and $T_k \rightarrow T_j$ then $T_k \rightarrow T_i$.*

With these definitions in mind, we can give a formal LRC history definition.

Definition 7 (LRC History) *A history H over a set of transactions T is a LRC History if, given T_i, $T_j \in T$, these conditions hold:*

1. *If T_i reads from T_j and $a_j \in T_j\colon a_i \in T_i$.*
2. *If $r_i(x_{j,l}) \in H, \nexists\, w_j(x_{j,m}) \in H\colon m > l$.*
3. *If $T_j \rightarrow T_i$ then $\neg(T_i \rightarrow T_j)$.*
4. *If $r_i(x_j) \in H, \nexists\, w_k(x_k)$ such that $w_j(x_j) < w_k(x_k) < r_i(x_j)$.*

The first condition prevents a transaction to read an item value written by an aborted transaction to avoid dirty reads. The second condition ensures that all reads are always performed over final transaction values and never over intermediate ones. In the third condition, all dependencies within two transactions must follow the same direction. This forces LRC-histories to serialise transactions by its *reads from* and *overwrite* dependencies. Note that this simulates locking RC effects without being dependent on locking techniques. Finally, condition four remarks that every read over any database item must always see its last modification performed by any transaction.

LRC definition is very close to RC, as defined in [1], but is not exactly the same. In ours, a committed transaction must always see item values written by transactions which will eventually commit. In [1], a transaction can only see committed transactions values and an attempt to read a non committed one results in a lock of such transaction until commits or aborts. Basically, this is the difference between the loose and the strict Dirty Read phenomenon interpretation in [1]. For example, the following history:

$$H1 = r_i(x_0)\, w_i(x_i)\, r_j(x_i)\, r_j(y_0)\, r_i(y_0)\, c_j\, w_i(y_i)\, c_i$$

is forbidden in RC but is allowed in LRC isolation level. Suppose that $x_0 = 50, y_0 = 50$, T_i writes $x_i = 10, y_i = 90$ and exists the integrity constraint $ic : x + y = 100$. In H, T_j reads $x_i = 10$ and $y_0 = 50$ and sees an inconsistency in ic since $x_i + y_0 = 60 \neq 100$.

This phenomenon could seem important enough to justify the use of RC instead of LRC but there are at least two reasons against this thought. First of all, RC avoids $H1$ but does not avoid the following one, which has the same effects (i.e., transaction T_j reads values y_0 and x_i, being 50 and 10, respectively, and violating the IC requiring $x + y = 100$):

$$H2 = r_j(y_0)\, r_i(x_0)\, w_i(x_i)\, r_i(y_0)\, w_i(y_i)\, c_i\, r_j(x_i)\, c_j$$

In fact, similar examples with integrity constraint violations can be found in practically all isolation levels different from a serial execution in a centralised DBMS.

On the other hand, this work is addressed to replicated environments (See Section 8) and every node is supposed to have a local DBMS providing RC. Therefore, $H1$ will be avoided in local nodes and hence in the replicated system.

We think that this kind of phenomenon must be taken into account in applications that need high isolation guarantees as, for example, bank account managers. This kind of applications needs higher isolation levels like Snapshot Isolation or Serialisable. This paper is addressed to a kind of applications where

these integrity constraints are not normally used and is preferable avoid locks in order to increase performance.

Until now, we have defined some useful general database concepts. From now on, we are going to apply and extend these concepts to replicated environments in order to present GLRC as an isolation level near to LRC useful in such environments and give some guides about implementing LRC and GLRC in database replication protocols.

4 Generalised Loose Read Committed

As it has seen before, the single difference between LRC and GLRC is the fact that GLRC allows a transaction to read an older version than the latest one at the time the read operation is performed.

Definition 8 (GLRC History) *A history H over a set of transactions T is Generalised Loose Read Committed (GLRC) iff, given T_i, $T_j \in T$ and an item x of the database:*

1. *If T_i reads from T_j and $a_j \in T_j : a_i \in T_i$.*
2. *If $r_i(x_{j,l}) \in H, \nexists\, w_j(x_{j,m}) \in H : m > l$.*
3. *If $T_j \rightarrow T_i$ then $\neg(T_i \rightarrow T_j)$.*

Note that LRC definition 7 is GLRC one plus condition four.

5 Replicated System Model

We suppose a fully replicated system composed by N nodes interconnected by a reliable network. We suppose the existence of a group communication service with atomic total order multicast support. Every node has a local copy of every item in the database.

A history H represents the user view of the execution of transactions. From the server side every replica N_a views its own execution and hence its own history H_a. To decide where all node executions drive to a global Loose Read Committed equivalent execution we need to introduce the concept of *one-copy Loose Read Committed equivalence*.

6 One-Copy Loose Read Committed equivalence

In a fully replicated environment, every node stores a copy of every item. When a transaction is executed, at least its writes must be eventually propagated to all replicas. Its reads must be performed at least in one replica, maybe in more.

In the first case, we have a Read One Write All strategy (ROWA), commonly used in replicated environments due to its good performance [8] since read-only transactions can be executed locally in one of the nodes without any further communication.

We represent as $r_i^a(x)$, $w_i^a(x)$, c_i^a and a_i^a a read, write, commit and abort operation of a given transaction T_i executed in node N_a. A transaction T_i *is local* to node N_a if T_i's reads are performed in N_a.

We define T_i^a as the transaction composed by the operations of T_i executed in node N_a. Formally:

Definition 9 *Given a transaction T_i and a node N_a, we define T_i^a as a transaction composed by:*

- *If $w_i(x) \in T_i$, $w_i^a(x) \in T_i^a$.*
- *If $r_i(x) \in T_i \wedge T_i$ is local to N_a, $r_i^a(x) \in T_i^a$.*
- *If $c_i \in T_i$, c_i^a in T_i^a.*
- *If $a_i \in T_i$, a_i^a in T_i^a.*
- *If $o_{i1} < o_{i2}$ in T_i, and $o_{i1}, o_{i2} \in T_i^a$ then $o_{i1}^a < o_{i2}^a in T_i^a$*

We can define a new kind of histories in a similar way as Bernstein does in [7]. In our work these histories will be named as Replicated Histories or R-Histories.

Definition 10 (R-History) *An R-History H_r over a transactions set T and a nodes set N is a partial order $<_r$ where:*

- *For every T_i^a of $T_i \in T$ and every $o_i^a \in T_i^a$, $o_i^a \in H_r$.*
- *For every T_i^a of $T_i \in T$ and every $o_{i1}^a, o_{i2}^a \in T_i^a$ If $o_{i1}^a < o_{i2}^a \in T_i^a$, $o_{i1}^a <_r o_{i2}^a \in H_r$.*
- *If $r_i^a(x_j) \in H_r$ then it exists $w_j^a(x_j) \in H_r$ such that $w_j^a(x_j) <_r r_i^a(x_j)$.*

We say that an R-History is *one-copy loose read committed* (1C-LRC) if it is LRC-equivalent to an LRC history. To define LRC-equivalence we use as basis the view equivalence definition made by Bernstein in [7].

Definition 11 (LRC-equivalence) *Given an R-history H_r over N and T and a LRC history H, H_r is LRC-equivalent to H over T if:*

- *For every node $N_a \in N$, if T_i^a reads from T_j^a in H_r, T_i reads from T_j in H.*
- *For every node $N_a \in N$, if $w_i^a(x)$ is the last write in H_r, $w_i(x)$ is the last write in H.*

Notice that H is a LRC history so all reads over an item see always its last update made before the read in $<_H$ order.

7 One-Copy Generalised Loose Read Committed equivalence

We can also extend the *one-copy loose read committed* definition to *one-copy generalised loose read committed* (1C-GLRC) extending also LRC-equivalence definition to GLRC-equivalence by defining H as a GLRC history instead of a LRC one. Formally:

Definition 12 (GLRC-Equivalence) *An R-History is one-copy generalised loose read committed if is GLRC-equivalent to a GLRC history. Given an R-history H_r over N and T and a GLRC history H, H_r is GLRC-equivalent to H over T if:*

- *For every node $N_a \in N$, if T_i^a reads from T_j^a in H_r, T_i reads from T_j in H.*
- *For every node $N_a \in N$, if $w_i^a(x)$ is the last write in H_r, $w_i(x)$ is the last write in H.*

8 Providing Read Committed in Some Typical Database Replication Techniques

At this point, it has been introduced how to evaluate whether a replicated database history fulfils 1C-LRC or 1C-GLRC. In a replicated system, the way transactions are executed, and hence, histories are built, are managed by a replication protocol. In the literature there have been lots of database replication protocol proposals. As it has been pointed out before, the most effective way to achieve database replication is by ROWA protocols. In the same way, there has been may attempts to classify them according to several factors [9,10,5]: server architecture, update propagation, replica interaction, transaction termination, ... In this section, we will follow the database replication schemes presented in [5].

We will always name as *delegate server* or N_d the node to which the client sends the transaction. We also suppose in every node a local DBMS which provides (locally) RC isolation level.

8.1 Active Replication

With Active Replication, once the delegate server receives a client transaction, propagates it in a message using total order broadcast. Once a node delivers the message, executes the transaction and, if it is the delegate server, sends the response to the client.

First of all, we know that all local DBMSs provide the RC level. By definition of active replication, we also know that all messages are delivered in the same order to all replicas and all of them execute transactions in delivery order. So, given a set of transactions T, to proof that every node sees the same *reads from*

dependencies and obtains the same final values, we only need to ensure that every node DBMS executes the transaction in a deterministic way, that is, given the same set of transactions provided in the same order the result is always the same.

It is easy to prove that if every node executes the same transactions in the same order and the local DBMS of every node is deterministic and provides RC, all nodes histories will be RC and equivalent among them.

Read-only transactions can be executed only in its delegate replica without being broadcast to the rest of the nodes. LRC guarantees are ensured to these transactions since the delegate node local DBMS provides RC and no consistency checks have to be made between nodes because no writes are performed.

8.2 Primary Copy Replication

In Primary Copy Replication, all transactions are executed in the same delegate server, named primary node. Once a transaction is executed and commits, its writeset is broadcast to the other nodes, named secondary nodes. In any case, the response is forwarded to the client by the primary node. Since all local DBMSs provide RC, the primary copy clearly provides the same level. Total order broadcasts and applying writesets in delivery order ensures consistency in secondary nodes. Since all broadcast messages have the primary node as the sender, providing total order is as simple as numbering messages in incremental order.

8.3 Certification-Based Replication

In this case, the delegate server executes locally the transaction operations until a commit is requested. Before applying it, the node collects all written items (writeset) and all accessed items (readset) in order to broadcast them to all nodes. Once a node receives a transaction writeset and readset, it tries to certify them against previous committed ones to decide if the former must commit or abort. Since all nodes have the same information, every node will take the same decision without any consensus phase. In a non delegate server, the writeset of a committing transaction must be applied. In the delegate server, the transaction is committed and the result returned to the client.

But, what must our protocol do to produce LRC histories? And GLRC? Since all nodes fulfil locally RC guarantees and all nodes deliver the readsets and writesets in the same order, if we apply the writesets in the same delivery order we will ensure that all *overwrite* conflicts are solved in the same way in all nodes. To do that, every delivered transaction must have higher priority than local transactions in the local database. If a writeset conflicts with a local transaction then the last one must be aborted in the local DBMS. If this transaction has already sent its writeset and readset, they will be eventually delivered and certified, so the local abort does not need to be transmitted to the client. In

other cases, the client must be informed.

We also need to ensure that all *reads from* conflicts are solved in the same way in all nodes. Since the reads are performed only in the delegate server, we do not need to check if all nodes have the same *reads from* dependencies, like in active replication, but we need to ensure that every read sees the last applied update. In GLRC this last condition is not necessary, which is an important difference, as we can see in the following example.

Suppose that a transaction T_i starts in a node N_a, reads T_j modification of item x and requests its commit. Since local DBMS provides RC, x_j is the last update of x performed in N_a at the time T_i reads it. Before T_i writeset and readset are delivered, transaction T_k's ones arrive modifying item x. Since all nodes apply writesets and readsets in the same order, in all nodes T_k will be applied after T_j but before T_i. So, T_i has lost T_k's x update.

In GLRC, this effect is allowed so we can guarantee GLRC ensuring RC in every local DBMS and applying the writesets in the same order they are delivered. Notice that readsets are not needed at all so we do not have to broadcast them.

In LRC a transaction is not allowed to lose updates, so every node must check every transaction read to ensure that has not seen older values and abort it if this happens. To do that, the protocol needs every transaction readset and they must be broadcast. In order to check if some read has been made over an older value, we can include in the readsets and writesets the versions read and written. This implies a global version management support in our database replication protocol. To detect if some read is over an older item version, we must check if some previous writesets have established a newer one over the same item.

8.4 Weak Voting Replication

As we have seen, the certification technique is based on broadcasting writesets and readsets to allow every node to check conflicts by itself in a certification step. In weak voting only writesets are broadcast so reads are only seen in the delegate node and hence only this node can detect a transaction *reads from* dependencies.

Again, overwrite dependencies can be guaranteed applying transactions in the same order they are delivered. Like in certification, *reads from* guarantees are also ensured by local DBMSs except by the fact that a transaction can lose updates. Since GLRC allows losing updates, with this isolation level we do not have to do anything more.

On the contrary, in LRC losing updates is not allowed and we need to take care about it. Unfortunately, in this case every node does not have every transaction readset (readsets are not broadcast) and only every transaction delegate

server knows its reads. Therefore, if the delegate server detects that some transaction T_i has lost an update, it must advise the rest of the nodes with an abort message before they apply T_i writeset. In other case, a T_i commit message must be sent. In consequence, once a writeset is delivered, every node must wait until a confirmation or abort message –associated to this writeset– arrives from the delegate server therefore a new broadcast is necessary to provide LRC.

Notice that a certified-based GLRC database replication protocol is also a weak-voting one and vice versa. Note that the single difference between these two schemes resides in how reads over older item values are detected and just this feature is not needed in the GLRC isolation level. This implies an important simplification on the normal weak voting behaviour. When a weak voting protocol is used for providing the serialisable isolation level, a second multicast (reliable but without total order) is always needed in order to notify whether the transaction has been committed or aborted. For GLRC, that multicast is not needed, since no information regarding readsets is needed in order to decide how a transaction should be terminated.

9 Conclusions

In this paper, we have presented the new *Generalised Loose Read Committed* isolation level, a weaker version of RC useful in replicated environments since a read is allowed to see older versions. We have also presented the necessary one-copy equivalence for LRC and GLRC as a basis to deduce where some database replication protocol fulfils these isolation levels restrictions. Finally, we have used this knowledge to define the steps to be followed to obtain LRC or GLRC with the most typical database replication protocol schemes defined in [5].

10 Appendix 1: PL-2 and GLRC equivalence

The PL-2 isolation level was presented by Adya in [3]. The Adya isolation level definitions were based on DSG graphs. In a DSG graph, every vertex is a committed transaction and every edge represents some kind of dependency between its two vertices.

Definition 13 (DSG dependency edges) *There are three kinds of dependencies:*

- *Direct read-dependency: T_i directly read-depends on T_j if reads some item value written by T_j.*
- *Direct write-dependency: T_i directly write-depends on T_j if overwrites some item value written by T_j.*
- *Direct anti-dependency: T_i directly anti-depends on T_j if overwrites some item value read by T_j.*

Direct read-dependency and direct write-dependency are considered as dependency edges. Notice that these definitions are analog to *reads from* and *overwrite* relations we have defined previously (page 4).

Definition 14 (PL-2 isolation level) *PL-2 is defined as the level which avoids the following three phenomena:*

- *G1a: Aborted Reads: a committed transaction T_i reads some item value written by an aborted transaction T_j.*
- *G1b: Intermediate Reads: a committed transaction T_i reads some item value written by T_j which is not its final item modification.*
- *G1c: Circular Information Flow: if a history H contains a directed cycle consisting entirely of dependency edges.*

Theorem 1 *Given a history H, H is a PL-2 history iff is also GLRC.*

proof 1: A GLRC history H is also a PL-2 history. By absurd reduction, we suppose the existence of at least one GLRC history H which is not PL-2. $G1a$ and $G1b$ are avoided since they are identical to GLRC conditions 1 and 2 (page 6). Hence, $G1c$ phenomena must appear since we have supposed that H is not PL-2. $G1c$ rises when DSG(H) has a dependency cycle [3]. A dependency cycle is composed only by dependency edges. As we said before, if a transaction T_a directly read-depends on T_b then T_a reads from T_b. In the same way, if T_a directly write-depends from T_b then T_a overwrites T_b. So, taking randomly two transactions T_i and T_j of the dependency cycle, it exists a dependency path between T_i and T_j ($T_i \neq T_j$) and another dependency path between T_j and T_i. Since every edge of the path is a *depends on* relation and applying transitivity, $T_i \rightarrow T_j$ and $T_j \rightarrow T_i$, which contradicts condition 3 of GLRC.

proof 2: A PL-2 history H is also a GLRC history. By absurd reduction, suppose that exists a PL-2 history which is not GLRC. Again, H clearly fulfils 1 and 2 GLRC definition conditions (page 6) since they are equivalent to PL-2 $G1a$ and $G1b$ phenomena. As we suppose that H is not GLRC, 3 must not be fulfilled in H so there are at least two transactions T_i and T_j such $T_i \rightarrow T_j$ and $T_j \rightarrow T_i$. By definition of $\rightarrow$ partial order, if $T_i \rightarrow T_j$ then either T_i depends on T_j or exists some T_k: $T_i \rightarrow T_k$ and T_j depends on T_k. We can also apply the same reasoning to $T_i \rightarrow T_k$ so we finally have a path of dependencies among a set of transactions starting with T_i and ending with T_j. Remember that a *depends on* relation must be an *overwrite* or a *reads from* one. *reads from* is equivalent to *directly read-depends* relation defined by Adya, and *overwrite* is equivalent to *directly write-depends*. So, we have a dependency edges path between T_i and T_j. As $T_j \rightarrow T_i$ also, we can construct another dependency path from T_j to T_i, which closes a dependency cycle in DSG(H) and $G1c$ appears. So, H is not a PL-2 and we have reached a contradiction.

References

1. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. In: Proc. of the ACM SIGMOD International Conference on Management of Data, San José, CA, USA (1995) 1–10
2. PostgreSQL Global Development Group: PostgreSQL 8.2.4 documentation (2007) Accessible in URL: http://www.postgresql.org/docs/8.2/interactive/index.html.
3. Adya, A., Liskov, B., O'Neil, P.: Generalized isolation level definitions. In: IEEE Intnl. Conf. on Data Engineering, San Diego, CA, USA (2000) 67–78
4. Bernabé-Gisbert, J.M., Salinas-Monteagudo, R., Irún-Briz, L., Muñoz-Escoí, F.D.: Managing multiple isolation levels in middleware database replication protocols. In: Proc. of the 6th Intnl. ISPA Conf., Sorrento (Naples), Italy, Springer (2006) 511–523
5. Wiesmann, M., Schiper, A.: Comparison of database replication techniques based on total order broadcast. IEEE Trans. Knowl. Data Eng. **17**(4) (2005) 551–566
6. Elnikety, S., Pedone, F., Zwaenepoel, W.: Database replication providing generalized snapshot isolation. In: 24th IEEE Symposium on Reliable Distributed Systems, Orlando, FL, USA (2005) 73–84
7. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)
8. Jiménez-Peris, R., Patiño-Martínez, M., Alonso, G., Kemme, B.: Are quorums an alternative for data replication? ACM Trans. Database Syst. **28**(3) (2003) 257–294
9. Gray, J., Helland, P., O'Neil, P.E., Shasha, D.: The dangers of replication and a solution. In: SIGMOD Conference, Montreal, Canada (1996) 173–182
10. Wiesmann, M., Schiper, A., Pedone, F., Kemme, B., Alonso, G.: Database replication techniques: A three parameter classification. In: SRDS. (2000) 206–215