Extending Mixed Serialisation Graphs to Replicated Environments

Josep M. Bernabé-Gisbert and Francesc D. Muñoz-Escoí

Instituto Tecnológico de Informática, Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, Spain

{jbgisber, fmunyoz}@iti.upv.es

Technical Report TR-ITI-ITE-07/20

Extending Mixed Serialisation Graphs to Replicated Environments

Josep M. Bernabé-Gisbert and Francesc D. Muñoz-Escoí

Instituto Tecnológico de Informática, Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, Spain

Technical Report TR-ITI-ITE-07/20

e-mail: {jbgisber, fmunyoz}@iti.upv.es

Abstract

A Database Management System normally deals with a heterogeneous set of transactions which do not necessarily need the same isolation guarantees if executed concurrently. Centralised DBMSs can manage this kind of situations since they normally use locks and every transaction implicitly requests the necessary locks to ensure its isolation needs. Nevertheless, in replicated environments this issue is not solved since the most used replication schemes can not be easily adapted to such a heterogeneous environment as in centralised ones. In fact, it is even hard to prove whether a replication protocol is ensuring every transaction isolation guarantees unless only one isolation level at a time is supported. In this document we extend Adya's Mixed Serialisation Graphs with more isolation levels and apply them to replicated environments to be able to know when a given replication protocol ensures every transaction guarantees.

1 Introduction

Nowadays, a lot of applications access and modify information stored in a database. These operations are normally packed in transactions which are a sequence of read and write operations. In theory, a DBMS must ensure every transaction ACID guarantees: Atomicity (all or none transaction effects are applied), Consistency (integrity constraints are respected), Isolation (a transaction can not see the effects of any unfinished concurrent transaction) and Durability (all committed transactions changes must be persistent in the database).

Nevertheless, DBMSes normally allow some relaxed versions of isolation guarantee. The more relaxed isolation level is used, the better performance will the system have, but some kind of interferences between concurrent transactions can appear. Obviously, some applications will tolerate a weaker isolation level but some others will need a strong one. In fact, some times not all transactions of the same application will need the same isolation level. A database can also be accessed by different applications with different isolation needs. If our DBMS can manage the concurrent execution of transactions with different isolation level we will improve our system performance. If not, all transactions will be executed with the strictest isolation level needed and this will increase the average response time of the whole system.

The most used centralised DBMSes use locks to control concurrency. With this technique, every transaction operation tries to obtain the required lock to its isolation level. If it conflicts with any other concurrent lock, its execution is blocked until that lock is released.

In replicated systems, the use of distributed locks is too expensive and normally optimistic techniques are used [12]. With them, a transaction is executed in one node until the commit request is received. Once this happens, a validation step starts to decide whether this transaction has come into conflict with any other one executed concurrently in other node. This validation step will depend on which technique is used but normally is based on searching conflicts with previously validated transactions. In existing protocols,

this conflict detection supposes that all transactions are executed with the same isolation level. In fact, replication protocols are normally oriented to only one isolation level (Snapshot Isolation or Serialisable, in most cases). Moreover, the most used theory defines isolation level as what a history (transaction set execution) must ensure and not a unique transaction so, how can we apply this theory in a heterogeneous system supporting the concurrent execution of transactions with different isolation level?

As an example, imagine a Read Committed (RC) transaction T_i reading an item x and, after that, a Serialisable one T_j writing it but before T_i commits: $H = r_i(x)\ w_j(x)\ c_i\ c_j$. Serialisable isolation level does not allow an item to be written if another uncommitted transaction has read it (which is called a Fuzzy Read [3]). Nevertheless, in Read Committed this phenomenon is allowed. So, is this history correct since T_i is RC or is incorrect since T_j is Serialisable?

Adya has been one of the few authors trying to deal with this issue [1]. Nevertheless, his main objective was to give new isolation level definitions independent to any technology and without falling in ambiguity, in contrast to previous proposals [3]. In his work, Adya extends Serialisation Graphs (SG), presented by Bernstein [6], as a way to represent dependencies between transactions. Isolation levels where defined as properties to be accomplished in such graphs. Adya presented also some extension of his own graph. One of them, named Mixed Serialisation Graph (MSG), able to represent dependencies between transaction with different isolation level. Unfortunately, in MSG graphs only the three basic Adya's isolation levels (PL-1, PL-2 and PL-3) were supported and no adaptation to replication environments was proposed. In fact, Snapshot Isolation (PL-SI in Adya work), an isolation level used in a lot of replication protocols, was unsupported. Also, some classical problems in replication, like consistency between replicas, were impossible to represent since only dependencies between transactions are considered and no information about nodes is considered.

The main goal of this work is to extend and adapt Adya's MSG to replicated environments.

In the following section, we will present the system model used in this paper. In Section 3 we will formalise transaction and history concepts and their replicated versions. In the next one (Section 4) we will present Adya's work which will be extended in Section 5 to support Snapshot Isolation level and adapted to replicated environments in Section 6. In Section 7, we discuss about how can be proved the correctness of a replication protocol supporting more than one isolation level at a time. An example will be presented in Section 8. Finally, in Section 9 the paper concludes.

2 System Model

Some definitions and concepts given and used in this work (like isolation level definitions) are independent on whether we have or not an underlying replicated system. In these cases, we assume the existence of some kind of database (replicated or not) composed by a set of items which can be read or modified. These operations are invoked by a client using transactions. A transaction T_i is a set of read r_i and write w_i operations executed atomically (a formal definition is given in section 3), that is, all or none of them are executed. Every transaction ends with a special operation which can be a commit c_i or an abort a_i. The commit operation makes persistent all T_i writes and the abort one invalidates them. With $r_i(x)$ and $w_i(x)$ we represent a T_i read or write operation over an item x. $r_i(x_j)$ indicates that the value read is the final modification of x performed by transaction T_j. $w_j(x_j)$ represents then T_j last modification of x. If the item value read or written is the l-th modification we represent it by $r_i(x_{j.l})$ and $w_j(x_{j.l})$. Finally, o_i represents a T_i operation without specifying its type.

When needed, we will represent as $r_i(P, Vset(P))$ a T_i transaction read based on some predicate P. $VSet(P)$ represents all the items which are or can be in P's relations. This includes all items, existent or not when the write is performed, that can be potentially checked to resolve P, regardless of whether they fulfill the predicate or not. A write based on a predicate will be represented as a predicate read followed by every item write.

In the cases we refer explicitly to a replicated system, we suppose a fully replicated one composed by N nodes. Since it is fully replicated, every node N_a has a copy of every item x, represented by x^a, stored in a local DBMS. With T_i^a we represent the operations subset of T_i executed in N_a. Read, write, commit and abort operations notation is also extended in the same way, so, for example, $r_i^a(x_j^a)$ represents a T_i read operation executed in node N_a over the last modification of x_j performed by T_j in N_a. Since all operations

performed in a node N_a will be over N_a item copies, the last example will be also represented as $r_i^a(x_j)$. We suppose that every local DBMS provides locally every isolation level supported by the whole system. Replication is provided by a middleware deployed on top of the DBMS. This middleware has access to a group communication system with atomic broadcast [9] support (or uniform atomic broadcast if failures are considered).

Notice that this terminology is the same used by Adya in his work [1] but extended to easily represent transactions execution in replicated environments.

3 Definitions

In the system model we have given a brief idea of some basic concepts. In this section we will formalise them and give some other ones useful to this work.

First of all, we have formalised Adya [1] transaction definition as a sequence of operations:

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

- $c_i \in T_i \vee a_i \in T_i$
- $c_i \in T$ *iff* $a_i \notin T_i$
- *If* $c_i\ (a_i) \in T_i, \forall o_i \neq c_i\ (a_i) \in T_i, o_i < c_i\ (a_i)$
- *Given* $o1_i, o2_i \in T_i, o1_i < o2_i \vee o2_i < o1_i$

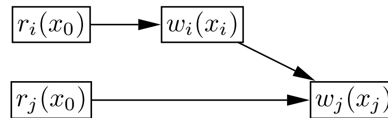
A *committed transaction* is a transaction whose final operation is a commit. In the same way, an *aborted transaction* is a transaction which ends with an abort.

Given a set of transactions T, its execution result depends on the order the transactions and their operations are executed. This is normally represented as a history. We have taken the definition given in [6], adapting it to fit with the previous transaction definition.

Definition 2 (History) *A history H [6] 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 $w_j(x_j) \in H \wedge w_j(x_j) <_H r_i(x_j)$.*
- $\forall o_i(x), o_j(x) \in H$ *where at least one of them is a write* : $o_i(x) <_H o_j(x) \vee o_j(x) <_H o_i(x)$.

Notice that a History is a partial order and two operations may not be ordered. For example, two read operations of different transactions are never ordered, even if they are over the same item, since the result of every one of them never depends on the other one. For example, imagine two concurrent transactions both reading and writing the item x. A possible execution history can be:



If it is possible, we will represent this sequentially to make the notation more understandable. For example, the previous example will be normally represented as:

$$r_i(x_0)r_j(x_0)w_i(x_i)w_j(x_j).$$

In a fully replicated environment, a transaction is initially forwarded to one of the nodes, named **local node**. Nevertheless, all committed transaction writes are applied eventually in all nodes. Therefore, given a transaction T_i, it can be executed only in one node if it is read-only or aborted. In other case, at least all T_i writes will be applied in all nodes. We define as *Node Transaction* T_i^a the T_i operations subset executed in node N_a:

Definition 3 (Node Transaction) *Given a transaction T_i and a node N_a, we define T_i^a as:*

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

Remember that not all transactions must be executed in all nodes (only committed non read-only ones). We will refer as T^a the subset of T executed in N_a.

We can also define a new kind of replicated histories in a similar way as Bernstein does in [6]. In our work these histories will be named as Replicated Histories or R-Histories.

Definition 4 (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 $w_j^a(x_j) <_r r_i^a(x_j)$.*
- *$\forall N_a \in N \wedge \forall o_i^a(x), o_j^a(x) \in H_r$ where at least one of them is a write: $o_i^a(x) <_r o_j^a(x) \vee o_j^a(x) <_r o_i^a(x)$.*

Then, a replicated history, is in fact, the union of every one of the local histories produced by every node.

4 Mixed Serialisation Graph (MSG)

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. [3]. 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. [1] proposed a new set of definitions 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 [6], to represent dependencies between transactions in histories.

In a given history H DSG, every vertex represents a committed transaction in H and every edge a dependency between two transactions. There are three kinds of dependencies:

- Directly Read-Depends: T_j *directly read-depends* on T_i if $r_j(x_i)$ or $r_j(P, VSet(P)) \wedge x_i \in VSet(P)$.
- Directly Write-Depends: T_j *directly write-depends* on T_i if $w_i(x_i)$, $w_j(x_j)$, $w_i(x_i) < w_j(x_j)$ and does not exist $w_k(x_k)$: $w_i(x_i) < w_k(x_k) < w_j(x_j)$.
- Directly Anti-Depends: T_j *directly anti-depends* on T_i if $r_i(x_0) \wedge w_j(x_j) \wedge r_i(x_0) < w_j(x_j)$ and does not exist $w_k(x_k)$: $r_i(x_0) < w_k(x_k) < w_j(x_j)$. The same holds if $r_i(P, VSet(P)) < w_j(x_j)$ and $x_o \in VSet(P)$).

Notice that we have used the definitions given in [2] instead of those proposed in [1].

These dependencies are represented in the DSG as follows:

- T_i *directly read-depends* on T_j: $T_j \xrightarrow{WR} T_i$
- T_i *directly write-depends* on T_j: $T_j \xrightarrow{WW} T_i$
- T_i *directly anti-depends* on T_j: $T_j \xrightarrow{RW} T_i$

In general, we will refer as *dependency* when we do not need to differentiate between direct read-dependency or direct write-dependency.

As an example, consider the following history: $H_1 = r_i(x_0)w_i(x_i)r_i(y_0)w_j(y_j)w_j(x_j)$. The associated DSG will be:

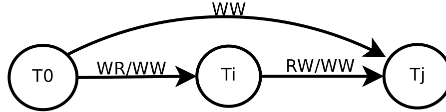


Figure 1: DSG of H_1

Adya uses DSGes to define his own basic isolation level definitions PL-1, PL-2 and PL-3. Given a history H:

- PL-1: DSG(H) can not have any cycle composed by direct write-dependencies.
- PL-2: (a) A committed transaction can not read a value written by an aborted one (aborted value in the sequel), (b) for every $r_i(x_{j.l}) \in H$, T_j commits in H, does not exist $r_i(x_{j.m})$ with $l < m$ (intermediate value), and (c) DSG(H) can not have any cycle composed by dependency cycles.
- PL-3: PL-2 restrictions extended to any cycle, including anti-dependencies.

These definitions substitute old Read Uncommitted (RU), RC and Serialisable (S) respectively, but are not exactly the same. As an example, PL-2 isolation level is weaker that RC [1, 4].

Notice that this abstraction does not include anything related to Snapshot Isolation (SI) and this level is the one ensured by most replication protocols ([7, 10, 11]) due to its optimistic orientation. To support it, Adya introduces an extension of DSG, named Start-order Serialisation Graph (SSG). This graph includes all DSG edges and vertices plus start-dependency edges. Given a history H and $T_i, T_j \in H$, T_j *start-depends* on T_i if $c_i <_h b_j$.

Given a history H, H is a PL-SI history if the following conditions hold:

- H is a PL-2 history.
- For any dependency edge from any transaction T_i to any other T_j, it exists also a start-dependency from T_i to T_j.
- There is not any cycle in SSG composed only by dependency edges and with a single anti-dependency one.

Notice that normally the isolation level supported in replication protocols is not SI but Generalised SI (GSI) [7]. GSI avoids reads to be blocked in replication protocols, nor like strict SI ones [8], by allowing a transaction to see an older snapshot. In these cases, the start time refers to the moment the snapshot belongs and not to the time the first transaction operation is executed. In this paper we will use SI to refer to both kinds of protocols.

To study the correctness of a history with different isolation level transactions, Adya introduced the Mixed Isolation Graph concept. This graph is based on DSG (not SSG). Given a history H, a MSG(H) has the same vertices than DSG(H) and a subset of DSG(H) edges. A DSG(H) edge from a given transaction T_i to another T_j is also in MSG(H) if:

- it is a direct write-dependency edge.
- it is a direct read-dependency edge and T_j is a PL-2 or PL-3 transaction.
- it is a direct anti-dependency edge and T_i is a PL-3 transaction.

A history H ensures every transaction isolation level if its MSG(H) has not any cycle and there are not any reads of aborted or intermediate values.

Again, this abstraction does not include Snapshot Isolation (SI) transactions and, in this case, Adya did not make any extension to support it. In fact, it is not clear how can this extension be made. For example, imagine a cycle with an anti-dependency starting from a PL-SI transaction. Is it valid since PL-SI allows it or is it invalid since PL-3 does not?

In conclusion, Adya's proposal is a good starting point but there is still important work to do. In the following section, we are going to revisit Adya's MSG definitions in order to be able to support SI transactions. In the next one, we will adapt it to replicated environments in order to be a useful tool to decide whether a given replication protocol produces valid histories.

5 Extended MSG

In a first step, we will give our own SI definition using DSGs instead of SSGs. With this definition, we will extend MSG graphs to allow SI transactions.

5.1 SI Isolation Level

Given a history H, this history is SI if:

(a) It is PL-2.

(b) A transaction T_i always sees all values written by transactions committed before T_i start.

(c) For any dependency edge from any transaction T_i to any other T_j, T_i commits before T_j starts.

It is easy to prove that SI and PL-SI are equivalent.

A SI history is also PL-SI. By absurd reduction, suppose that a SI history exists which is not PL-SI. A history is not PL-SI if either:

- It is not PL-2: if it is not PL-2, it is not SI by definition, or
- If any edge exists from any transaction T_i to any other T_j then T_i does not commit before T_j starts. If this happens, condition (b) of our previous SI definition is also violated; or, ...
- There is a cycle with a unique anti-dependency edge. Imagine that the cycle is composed by $T_1; T_2; ...; T_n; T_1$ and the anti-dependency exists from T_n to T_1. As we have seen in the previous point, for all T_i, T_{i+1} joined by a dependency edge we know that $c_i < b_{i+1}$. Since the path from T_1 to T_n is joined by dependency edges we know that $c_1 < b_2 < c_2 < .... < b_n < c_n$ so, $c_1 < b_n$. Since there is an anti-dependency edge from T_n to T_1, T_n has read a value overwritten by T_1 but this is not possible since $c_1 < b_n$ and the second SI condition says that T_n must see all T_1 updates.

So, a SI history is also PL-SI because we have reached to a contradiction.

A PL-SI history is also SI. Again by absurd reduction; suppose that a PL-SI history exists which is not SI. This time, a history is not SI if either:

- It is not PL-2: if it is not PL-2, it is not PL-SI by definition.
- A transaction T_i does not see at least one value written by a transaction committed before T_i start. If T_i does not see a value written by a committed transaction T_j we will have an anti-dependency edge from T_i to T_j in SSG. But we said that T_j has committed before T_i starts so there is another Start-dependency from T_j to T_i which closes a cycle with a unique anti-dependency edge and this is forbidden in a PL-SI history, or

- For any dependency edge from any transaction T_i to any other T_j, T_i commits before T_j starts. This restriction exists also in PL-SI.

So, again we have reached to a contradiction and we can say that PL-2 and SI isolation levels are equivalent.

5.2 Extended MSG

The main advantage of our SI definition is that it is based on DSG graphs instead of on SSG and forbids the same cycles than PL-2. This makes MSG graphs easy to extend in order to support SI transactions. An Extended Mixed Serialisation Graph (EMSG) is defined as follows:

Given a history H, an EMSG(H) has the same vertices than DSG(H) and a subset of DSG(H) edges. A DSG(H) edge from a given transaction T_i to another T_j is in EMSG(H) if either:

- *it is a direct write-dependency edge,*
- *it is a direct read-dependency edge and T_j is PL-2, PL-3* **or SI transaction***, or*
- *it is a direct anti-dependency edge and T_i is a PL-3 transaction.*

A history H ensures every transaction isolation level if its EMSG(H) has not any cycle, there are not reads of aborted or intermediate values, every SI T_i transaction sees all values written by any transaction committed before T_i start and, for every dependency edge between a transaction T_i and a SI transaction T_j, $c_i < b_j$.

It is easy to see that, given a history H without SI transactions, MSG(H) ensures all H transactions isolation level iff EMSG(H) ensures it.

6 EMSG in Replicated Systems

As we said in Section 2, a replicated system is composed by a set of nodes. Every node must eventually have a copy of every item value written in the database. This implies that all write operations of committed transactions must be transmitted to all nodes but reads can be performed in only one node. So, given a committed transaction T_i, every node N_a will execute a subset of T_i operations, named T_i^a, with at least all its writes.

So, given a set T of transactions, H_a will represent the history produced in N_a due to T^a execution (remember that T^a is the subset of T executed in N_a). As we have seen in Section 3, we represent as H_r the union of all nodes histories. With H_r, we can construct a graph G similar to EMSG, named RMSG, in the following way:

- Every committed transaction in T is a vertex in G.
- There is an edge from T_i to T_j in G if in at least one EMSG(H_a) exists an edge from T_i^a to T_j^a.

A RMSG(H_r) is valid if there is not any cycle and, in any node N_a: (a) no committed transaction reads any aborted or intermediate value, (b) for every SI transaction T_i, T_i^a sees all values written by any transaction committed before T_i^a start and, (c) for every dependency edge from T_i to a SI transaction T_j, in any node N_a with this dependency, $c_i^a < b_j^a$.

7 Proving Correctness of Replication Protocols

We say that a replicated system execution H_r of a set of transactions T is correct if the following conditions hold:

- Completeness: all committed transaction operations are executed in at least one node.

- Total replication: all nodes execute all committed transaction writes.
- Consistency: when executing a set of transactions T, all nodes eventually reach to the same final state.
- Isolation: all transactions isolation level guarantees are ensured.
- Equivalence: the result of whole system execution of T is equivalent to a correct centralised execution of T.

So, a given replication protocol is correct if all of its executions are also correct, that is, all conditions are held in any possible history produced by the protocol. In this section, we are going to study how every condition can be proven to obtain a list of what must be made in most cases to ensure a given protocol correctness.

Completeness should be easy to prove because in all schemes a transaction is totally executed at least in one node.

To ensure the total replication condition, we have to prove that all nodes commit the same transactions. In optimistic replication protocols we normally can ensure this by proving that all nodes receive and validate the same writesets.

Consistency is ensured by proving that all nodes apply the same writes, which has been proved in total replication condition, and that all transactions apply conflicting writes (writes over the same item) in the same order. If two nodes apply two conflicting writes in different order, it must exist a cycle in RMSG involving those two write committed transactions. As an example, imagine a system composed by two nodes (A and B) executing three transactions. Every one of them writes the same item x. Both nodes execute all writes but in different orders: $H_A = w_i(x_i)w_j(x_j)w_k(x_k)c_ic_jc_k$ and $H_B = w_j(x_j)w_k(x_k)w_i(x_i)c_jc_kc_i$. The EMSGs associated to these histories are:

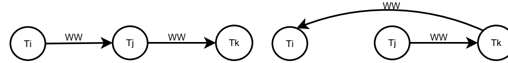


Figure 2: Node A and B EMSGs

which produce a cycle when the RMSG is constructed:

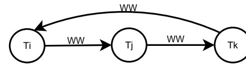


Figure 3: RMSG

So, if we can prove that any cycle is impossible in RMSG and total replication is guaranteed, consistency will be ensured.

To ensure all transactions isolation guarantees, we can normally rely (as replication protocols normally do) on supposing that the local DBMS ensures those isolation levels locally. In this case, some effects like reads of aborted or intermediate values will be directly avoided by the local DBMS. Proving that any cycle is impossible in RMSG will do the rest except if SI level is supported. In such case, we need to ensure also that every SI transaction sees all updates committed before its start and never observes those uncommitted. Recall that the SI validation process consists in finding conflicts with writes of previously validated conflicting transactions. Since two transactions T_i and T_j conflict if $b_i <_r c_j$ and $c_j <_r b_i$, this implicitly defines a global clock able to order at least all start and commit timestamps of all transactions. Finding this global clock will help to prove SI transactions isolation correctness. We will see that in the next section example.

So, ensuring consistency and isolation conditions correctness is hard related on ensuring the absence of cycles in any RMSG produced by the protocol. To prove this, we can use what we define as **increasing property**. For any edge, say from T_i to T_j, an increasing property is something which is always bigger in T_j than in T_i. If this happens, a cycle is impossible since it would drive us to a contradiction (a transaction

is bigger and lesser than some other at the same time). As an example, take now the cycle in Figure 3 and imagine that every edge implies a commit ordering, that is, for any edge, say from T_i to T_j, $c_i <_r c_j$. We will conclude that $c_i <_r c_j <_r c_k <_r c_i$, which is a contradiction.

About equivalence, there are two possible equivalence definitions: view equivalence or conflict equivalence. In the first case we need to prove that the final state is the same in all nodes (which is already ensured in our consistency condition) and all reads see the same values in all nodes in which are executed. If reads are executed only in one node, this is automatically ensured. About conflict equivalence, we need to prove that if some operation leads to an edge in some node EMSG, the same edge will appear in all nodes where the operation is performed. Again, if every transaction reads are executed only in one node and consistency is guaranteed, equivalence is automatically proved. A centralised one-copy equivalent history can be constructed maintaining every operation dependencies in RMSG and adding all non-dependent operations respecting only the ordering in its own transaction.

Summarising, to prove the correctness of a replication protocol we normally must:

- Ensure that at every committed transaction is totally executed in at least one node.
- Ensure that every local DBMS ensures locally all the isolation levels we want to support globally.
- Ensure that all nodes receive, validate and apply the same writes.
- Find an increasing property to ensure that any cycle can appear in any possible RMSG produced by the protocol.
- If SI is supported, define a global clock and ensure that every SI transaction sees all and only committed values of transactions committed before its start.

8 Example

In this section, we will take the SIRC protocol [11] to show how our theory can be used to prove its correctness.

8.1 SIRC Protocol

SIRC protocol was presented in [11] as an adaption of the GSI [7] SIR-SBD protocol presented in [10] in order to support also GLRC transactions [5] (equivalent to Adya's PL-2). This protocol supposes that all nodes have a local DBMS supporting RC and SI. It also assumes the existence of a group communication system able to broadcast messages in total order.

In SIRC, every transaction, for example T_i, is initially executed in its local node (N_l). As a start time for T_i, N_l assigns the number of committed transactions once T_i first operation is received. Once the commit request arrives, N_l gathers T_i writeset (WS_i) and broadcasts it to all active nodes (including itself). Due to the total order guarantees, all transactions will deliver all writesets in the same order. Once a node delivers a writeset, it is validated to decide if it conflicts with any other concurrent transaction and must be aborted or can commit. GLRC transactions are directly validated, GSI ones are validated only if its writes do not conflict with previous validated concurrent writesets. A writeset WS_i is concurrent with another previous validated WS_j if at least one item is written by both and end time of T_j is greater than start time of T_i. End time of a transaction T_i corresponds to the number of validated writesets when the T_i one is validated. Once a writeset is validated, it is enqueued in a list which is consumed by an asynchronous process in validation order following a FIFO criterion. Once a writeset is consumed, it is automatically applied in the local database, aborting any local conflicting transactions.

8.2 SIRC Correctness Proof

First of all, we need to prove that at all committed transaction operations are executed. Notice that every committed transaction is totally executed in its local node, so, this property is ensured.

In a second step, we need to be sure that all nodes deliver, validate and apply the same writes. Total order multicast ensures that all writesets are received in all active nodes in the same order. Once a node delivers WS_i it will validate it. If it corresponds to a RC transaction, WS_i is automatically validated. If it is SI, it will be validated against concurrent previously validated writesets. If this process is identical in all nodes, all of them will validate the same writesets. Since all nodes validate writesets in delivery order, and validation process depends, in the worst case, on previously validated transactions, is easy to see that all nodes will validate the same writesets. Nevertheless, a better proof can be found in [11]. Finally, all validated writesets are applied in validation order so, all nodes deliver, validate and apply the same writes.

In a third step, we need to define an increasing property to avoid cycles in all RMSG. In this case, we will use the committing ordering as our increasing property. To do that, we need to prove that every dependency edge from any transaction T_i to any other T_j implies that $c_i < c_j$. We will prove first that this happens with any write-dependency and, in a second step, with read-dependencies. Since SIRC do not support serialisable, we do not need to take care about anti-dependency edges. Remember that all nodes apply and commit validated writesets in validation order. Therefore, if WS_i overwrites some other transaction WS_j in one node, it will be applied in such order in all nodes and $c_i < c_j$ in the whole system.

Nevertheless, reads are only performed in its local node. Since local DBMS ensures RC and SI locally, if a SI transaction T_j reads from another T_i, T_i must have committed before T_j starts in this node so, $c_i < b_j < c_j$ in it. If T_j is not a read-only transaction, its writeset will be broadcast to all nodes. Since all nodes deliver, validate and apply writesets in the same order, they will apply T_i writeset before T_j one so, in all nodes $c_i < c_j$. If T_j is RC, since local DBMS uses locks, it will be locally blocked until T_i commits. After that, T_j will continue its execution and eventually commits so, $c_i < c_j$ locally. Again, if T_j local node has applied T_i updates before T_j ones, all nodes will apply its writesets in the same order, so, $c_i < c_j$ in all nodes and we have our increasing property:

Increasing property: every dependency edge leads to a commit ordering.

Finally, we need to find our global clock and ensure that all SI transactions see all committed values in its start and do not see any non-committed value.

Recall that T_i start time corresponds to the number of transactions committed in T_i local node when T_i first operation arrives. Furthermore, T_i end time corresponds to the number of committed transactions when T_i is committed ([11] proves that this is equivalent to the number of validated transactions when T_i is validated). Notice that this conforms the global clock in this protocol! Therefore, if T_i start timestamp is greater than another transaction T_j end timestamp is because T_i sees, when it starts in its local node, T_j commit and its updates must be included in T_i local timestamp (local DBMS ensures it). This proves the first SI specific condition. If T_j end timestamp is greater than T_i start timestamp, T_j commit increment can not be included in T_i start timestamp which implies that T_i does not see T_j commit when it starts in its local node and T_j updates are not seen (again, local DBMS ensures it). This ensures the second condition (a better proof can be again found in [11].

Notice that this protocol works if the underlying DBMS supports SI and RC locally.

9 Conclusions

As we have seen, some applications do not need to ensure a higher isolation level in all of their transactions. If a DBMS can take advantage of this property, performance can be increased since higher isolation level checks are only applied when needed and not over all transactions. Nowadays, centralised DBMS can easily support that kind of executions due to the use of locks. Nevertheless, in replication systems this is still an unsolved task because replication protocols are normally based on optimistic approaches and their techniques complicate the validation process when more than one isolation level at a time is supported.

In this work, we have taken Adya's Mixed Serialisation Graph, extended it to support Snapshot Isolation level (ensured by a lot of replication protocols due to it intrinsic optimistic orientation) and used with one copy conflict equivalence definition to be able to decide whether a given replication protocol ensures every transaction isolation level guarantees.

References

[1] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Massachusetts Institute of Technology, March 1999.

[2] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *IEEE Intnl. Conf. on Data Engineering*, pages 67–78, San Diego, CA, USA, March 2000.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 1–10, San José, CA, USA, May 1995.

[4] J. M. Bernabé-Gisbert. Providing support for data replication protocols with multiple isolation levels. In *OTM 2007 Workshops*, Vilamoura, Algarbe, Portugal, November 2007. Springer.

[5] J. M. Bernabé-Gisbert, J. E. Armendáriz-Iñigo, R. de Juan-Marín, and F. D. Muñoz-Escoí. Providing read committed isolation level in non-blocking ROWA database replication protocols. In *JCSD*, pages 145 – 157, Torremolinos, Málaga, Spain, June 2007.

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

[7] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database replication providing generalized snapshot isolation. In *24th IEEE Symposium on Reliable Distributed Systems*, pages 73–84, Orlando, FL, USA, October 2005.

[8] J. R. González de Mendívil, J. E. Armendáriz-Iñigo, F. D. Muñoz-Escoí, L. Irún-Briz, J. R. Garitagoitia, and J. R. Juárez-Rodríguez. Non-blocking ROWA protocols implement GSI using SI replicas. Technical Report TR-ITI-ITE-06/04, Instituto Tecnolgico de Informtica, Valencia, Spain, May 2007.

[9] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. ACM Press, 2nd edition, 1993.

[10] Francesc D. Muñoz, J. Pla, María Idoia Ruiz, Luis Irún, Hendrik Decker, José Enrique Armendáriz, and J. R. González de Mendívil. Managing transaction conflicts in middleware-based database replication architectures. In *SRDS*, 2006.

[11] R. Salinas-Monteagudo, J. M. Bernabé-Gisbert, F. D. Muñoz-Escoí, J. E. Armendáriz-Iñigo, and J. R. González de Mendívil. SIRC: A multiple isolation level protocol for middleware-based data replication. In *22nd International Symposium on Computer Information Sciences*, Ankara, Turkey, November 2007. IEEE-CS Press.

[12] Matthias Wiesmann and André Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Trans. Knowl. Data Eng.*, 17(4):551–566, 2005.