Design of a MidO2PL Database Replication Protocol in a Middleware Architecture

J.E. Armendáriz[1], F.D. Muñoz-Escoí[2], J.R. Garitagoitia[1], J.R. González de Mendívil[1]

Technical Report ITI-ITE-05/09

[1] Dpto. Matemática e Informática
Universidad Pública de Navarra
Campus de Arrosadía 31006 Pamplona, Spain
Ph./Fax: (+34) 948 16 80 56/95 21

[2] Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Camino de Vera s/n 46022 Valencia, Spain
Ph./Fax: (+34) 96 387 72 45/72 39

Email: {enrique.armendariz, joserra, mendivil}@unavarra.es, fmunyoz@iti.upv.es

Abstract

Middleware database replication techniques is a way to increase performance and fault tolerance without modifying the Database Management System (DBMS) internals. However, it introduces an additional overhead that may lead to poor response times. In this paper we present a modification of the Optimistic Two Phase Locking (O2PL) protocol [CL91] that orders transactions by way of a deadlock prevention schema, instead of using the total order transaction delivery obtained by Group Communication Systems (GCSs) [CKV01] techniques, and do not need the 2 Phase Commit (2PC) rule [BHG87]. We formalize its definition as a state transition system [Sha93] and show that it is 1-Copy-Serializable (1CS) [BHG87]. An outline of its adaptation to Snapshot Isolation (SI) [BBG+95] DBMSs is also provided.

1 Introduction

Database replication is a very attractive way for enterprises in order to increase their performance and afford site failures. These advantages imply the price of maintaining data consistency. Traditionally, database replication has been achieved by the modification of the DBMS internals, such as [CL91, BHG87, KPA+03]. This approach presents good performance but lacks of compatibility between several DBMS vendors. The alternative approach is to deploy a middleware architecture that creates an intermediate layer that features data consistency, the original database schema has to be modified with standard database standard features such as functions, triggers, stored procedures, etc. [JPPMKA02] in order to facilitate additional metadata that eases replication. This alternative introduces additional overhead that penalizes performance but it permits to get rid of DBMSs' dependencies.

The strongest correctness criterion for database replication is the 1CS [BHG87] that implies a serial execution over the logical data unit although there are many physical copies. In [JPPMKA02] a middleware architecture is introduced providing 1CS by way of a GCS [DSU04, CKV01]. GCS is used so as to determine the order in which transactions are executed on the system. The total order delivery guarantee establishes that all messages are delivered in the same order to all available sites [CKV01]. This is an interesting approach since transactions does not have to wait for applying the updates at the rest of sites in order to commit a transaction, as the 2PC rule states [BHG87], which increases its performance. However, we thought that relying on this strong GCS primitives, whose latencies and extra message rounds in environments where conflicts are rare it is a high price

to pay [DSU04, KPA+03]. Besides, the order of committed transactions is imposed by a "*black-box*" with a consensus algorithm that does not have any information about transactions, such as the number of objects read, written or the number of restarts. This may lead to the penalization of certain transaction patterns.

In this paper we propose an evolution of the O2PL [CL91] adapted to our middleware architecture, that only needs a reliable multicast to the rest of sites [HT94]. We have changed O2PL philosophy, since we do not need to wait for applying the updates at the rest of nodes in order to commit a transaction. Besides, we have added a deadlock prevention schema so as to avoid distributed deadlock, which is a dynamic function based on the transaction priority and its state; moreover, it imposes the order on which transactions are applied. We give the correctness proof of this replication protocol (MidO2PL) proposal, this correctness proof implies the interaction of the DBMS module, the GCS, the user transaction and the interaction of the replication protocol itself in another site. This can be done if we provide a formalization of the MidO2PL. We have used a state transition system similar to the one proposed in [Sha93]. This approach may be viewed as the I/O automata [Lyn96] composition of all its components. Finally, as most commercial DBMSs provide Snapshot Isolation [BBG+95], several isolation levels have been proposed for replicated databases such as Generalized Snapshot Isolation (GSI) [EPZ05]. We propose a MidO2PL modification in order to guarantee GSI. The rest of this paper is organized as follows: The system model is introduced in Section 2. The formalization of our MidO2PL protocol is presented in Section 3. Section 4 shows its correctness proof. Modifications of the MidO2PL to feature GSI are outlined in Section 5. Finally, conclusions end the paper.

2 System Model and Definitions

The system, shown in Figure 1, considered in this paper is a generalization of the system. It is composed by N sites (or nodes) which communicate among them using reliable multicast featured by a group communication system [CKV01]. We assume a fully replicated system. Each site contains a copy of the entire database and executes transactions on its data copies. A transaction is submitted for its execution over its local DBMS via the middleware module. The replication protocol coordinates the execution of transactions among different sites to ensure 1CS [BHG87]. We do not consider failures. Actions in Figure 1 are shown with arrows, they describe how the interaction between components is performed. They are an abstraction of our system. Thus, actions may easily be ported to the particular GCS primitives, JDBC methods, etc. For example, a transaction will see a JDBC interface to interact with the middleware, respectively the middleware will interact with the underlying DBMS using a JDBC interface and so on.

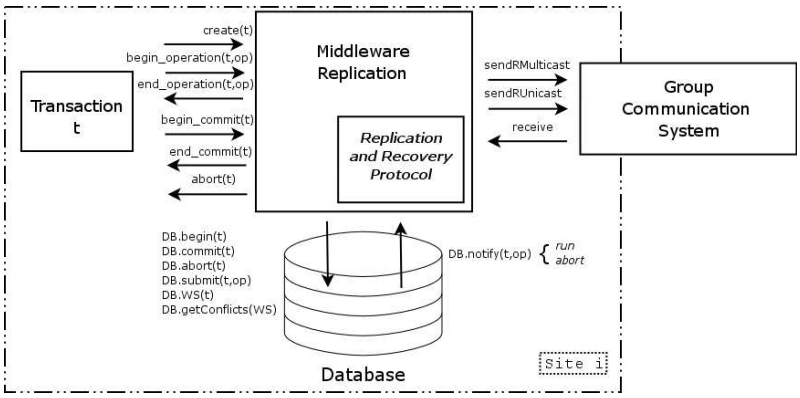

Figure 1: Main components of the system.

Communication system. Communication among sites is mainly based on reliable multicast [HT94]. Roughly speaking, reliable multicast guarantees three properties: (i) all correct processes agree on the set of messages they deliver; (ii) all messages multicast by correct processes are delivered; and, (iii) no spurious messages are ever delivered. These properties are enough for our replication protocol. Reliable broadcast imposes no description of the order in which messages are delivered. Besides, its cost is low in terms of physical messages per multicast.

Database. Each site includes a DBMS storing a physical copy of the replicated database. We assume that the DBMS ensures ACID properties of transactions and satisfies the ANSI SQL serializable transaction isolation level [BBG$^+$95]. The DBMS, as it is depicted in Figure 1 gives to the middleware some common actions. $DB.begin(t)$ begins a transaction t. $DB.submit(t, op)$, where op represents a set of SQL statements, submits an operation in the context of the given transaction. $DB.notify(t, op)$ informs about the success of an operation. It returns two possible values: run when the submitted operation has been successfully completed (the transaction submitting the operation will no longer perform an operation until it receives the run notification); or $abort$ due to DBMS internals, e.g. deadlock resolution, enforcing serialization, etc. As a remark, we also assume that after the successful completion of a submitted operation by a transaction, it can be committed at any time. In other words, a transaction may be unilaterally aborted by the DBMS only while it is performing a submitted operation. Finally, a transaction ends either by committing, $DB.commit(t)$, or rolling back, $DB.abort(t)$. We have added two functions which are not provided by DBMSs, but may easily be built by database triggers, procedures and functions [LKPMJP05]: $DB.WS(t)$ retrieves the set of objects written by t and the respective SQL update statements. In the same way, the set of conflictive transactions between a write set and current active transactions (an active transaction in this context is a transaction that has neither committed nor aborted) at a given site is provided by $getConflicts(WS(t)) = \{t' \in T\colon (WS(t') \cup RS(t')) \cap WS(t) \neq \emptyset\}$, where T is the set of transactions being executed in our system.

Transactions. Users access the system through their closest site to perform transactions by way of actions introduced in Figure 1. As it was pointed out, this is an abstraction. As a matter of fact, applications employ the same JDBC interface as the underlying DBMS, except actions to be performed when they wish to commit. Each transaction identifier includes the information about the site where it was first created ($t.site$), called its *transaction master site*. It allows the protocol to know if it is a local or a remote transaction. Each transaction has a unique priority value ($t.priority$) based on transaction information. A transaction t created at site i ($t.site = i$) is locally executed and follows a sequence initiated by $create(t)$ and continued by multiple $begin\_operation(t, op)$, $end\_operation(t,\ op)$ pairs actions in a normal behavior. The $begin\_commit(t)$ action makes the replication protocol start to manage the commit of t at the rest of replicas. The $end\_commit(t)$ notifies about the successful completion of the transaction on the replicated databases. However, an $abort(t)$ action may be generated by the local DBMS or by a replication protocol decision. For simplicity, we do not consider an application abort.

3 Replication Protocol Description

The MidO2PL is a Read One Write All Available (ROWAA) [BHG87] one. Informally, each time a client application issues a transaction (*local transaction*), all its operations are locally performed over its master site. Conflict detection is managed by the DBMS that guarantees a serializable isolation level [BBG$^+$95], hence there is no need to implement any database specific data structure. The remaining sites enter in the context of this transaction when the application wants to commit, the write set is grouped and sent to the rest of available sites. These updates are executed in the context of another local transaction, called *remote transaction*, at the rest of nodes. If the given transaction is a read only one, then it will directly commit. We do not model it for simplicity. The MidO2PL is different from the eager update everywhere protocol model assumed by [GHOS96], as it does not send any message until the user wishes to commit. Hence, only three messages are needed per transaction: one containing the $remote$ updates, another one for the $ready$ message sent by each remote site, and, finally, a $commit$ message. As the $remote$ message is delivered, the delivered transaction will pass through a function that checks its priority

against the rest of conflicting local transactions. It will determine if it is allowed to proceed or not and will prevent the appearance of distributed deadlocks. If it proceeds, it will send back a $ready$ message to the transaction master site. When the reception of $ready$ messages is finished, that is, all nodes have answered to the master site, it will send a $commit$ message saying that the transaction is committed. We assume that unilateral aborts for remote transactions never occur. If the transaction has lower priority, it will be enqueued and its possible execution will be checked several times, until it becomes the highest priority transaction or its master site decides to roll it back, in order to prevent distributed deadlocks.

In the following, we present this replication protocol as a formal state transition system as in [Sha93]. In Figure 2, a description of the states and steps of the replication protocol for a site i is introduced. An action can be executed only if its precondition is enabled. The effects modify the state of the system as stated by the sequence of instructions included in the action effects. Actions are atomically executed. It is assumed weak fairness for the execution of each action.

We will start with the states defined for this replication protocol. Each state and action is subscripted by the site at which it is executed. Each site has its own state variables (i.e., they are not shared among other nodes). The $status_i(t)$ variable indicates the execution state of a given transaction. The $participants_i(t)$ variable keeps track of the sites that have not yet sent the $ready$ message to the master site of t. $\mathcal{V}_i$ is the system current view, with a failure-free assumption, is $\langle 0, N \rangle$. As we use priorities we have defined a prioritized $queue_i$ variable that stores remote transactions that may not be scheduled due to the fact that its associated priority is lower than some conflicting transaction executing on i. The $remove_i$ manages the DB_i submission of enqueued transactions. The set of actions includes: $create_i(t)$, $begin\_operation_i(t, op)$, $end\_operation_i(t, op)$, $begin\_commit_i(t)$ and $end\_commit_i(t)$. These are actions executed in the context of a local transaction. The $end\_operation_i(t, op)$ is also executed by remote transactions. The $begin\_commit_i(t)$ starts the interaction of the rest of nodes. This set of actions is entirely self-explanatory from inspection of Figure 2.

The key action of our replication protocol is the $execute\_remote_i$ one, this action is invoked, at least, each time a transaction is delivered. In fact, this action will be invoked several times, as it is invoked after a commit, abort or a $remote$ delivery, until a transaction, t, is submitted to the DB_i. The remote updates, for that $WS(t)$, will only be applied if there is no conflicting transaction at node i having a higher priority than the received one. The $higher\_priority(t, t')$ function defines a dynamic priority deadlock prevention function, it depends on the state of the transaction ($status_i(t)$) and its priority. A new incoming conflictive remote transaction whose priority is lower than any other executing transaction, will be inserted again in $queue_i$. Therefore, the correctness of our solution is not compromised by the queue usage, since the transaction master site decides whether a transaction aborts or not. Finally, if the remote transaction is the one with the highest priority among all at i then it will send the $ready$ message to the master site. It will abort every local conflictive transaction and submits t to DB_i. Aborted local transactions in pre_commit state with lower priority will multicast an $abort$ message to the rest of sites. The finalization of the remote transaction ($end\_operation_i(t, op)$) changes its $status_j(t) =$ pre_commit, $j \neq node(t)$. It has to wait for the reception of the $commit$ message from the master site (as it has received all $ready$ messages), or straightly commit if the message has arrived. The reception of this message commits the transaction at the remainder sites ($receive\_commit_i(t)$). Recall that each time a transaction commits or rolls back the $queue_i$ is inspected in order to wake up waiting remote transactions. Finally, transactions in the pre_commit state are committable at any point from the DBMS point of view.

4 Correctness Proof

This section contains the proofs (atomicity and 1CS) of the MidO2PL automaton, introduced in Figure 2, in a failure free environment. Let us start showing that MidO2PL is deadlock free, assuming that deadlocks involving exclusively local transactions at a given site are directly resolved by the underlying local DBMS executing the $local\_abort_i(t)$ action. In case of remote transactions, we assume that as the $execute\_remote_i$ action abort all

Signature:
$\{\forall\, i \in N, t \in T, m \in M, op \in OP$: $\mathbf{create}_i(t), \mathbf{begin\_operation}_i(t, op), \mathbf{end\_operation}_i(t, op), \mathbf{begin\_commit}_i(t), \mathbf{end\_commit}_i(t),$
$\mathbf{local\_abort}_i(t), \mathbf{receive\_remote}_i(t, m), \mathbf{receive\_ready}_i(t, m), \mathbf{receive\_commit}_i(t, m), \mathbf{receive\_abort}_i(t, m), \mathbf{execute\_remote}_i,$
$\mathbf{discard}_i(t, m)\}$.

States:
$\forall\, i \in N, \forall\, t \in T$: $status_i(t) \in \{$idle, start, active, blocked, pre_commit, aborted, committed$\}$,
initially $(node(t) = i \Rightarrow status_i(t) =$ start$) \wedge (node(t) \neq i \Rightarrow status_i(t) =$ idle$)$.
$\forall\, i \in N, \forall\, t \in T$: $participants_i(t) \subseteq N$, initially $participants_i(t) = \emptyset$.
$\forall\, i \in N$: $queue_i \subseteq \{\langle t, WS\rangle : t \in T, WS \in OP\}$, initially $queue_i = \emptyset$.
$\forall\, i \in N$: $remove_i$: **boolean**, initially $remove_i =$ false.
$\forall\, i \in N$: $channel_i \subseteq \{m : m \in M\}$, initially $channel_i = \emptyset$.
$\forall\, i \in N$: $\mathcal{V}_i \in \{\langle id, availableNodes\rangle : id \in \mathbb{Z}, availableNodes \subseteq N\}$, initially $\mathcal{V}_i = \langle 0, N\rangle$.

Transitions:

$\mathbf{create}_i(t)$ // $node(t) = i$ //
pre $\equiv status_i(t) =$ start.
eff $\equiv DB_i.begin(t)$; $status_i(t) \leftarrow$ active.

$\mathbf{begin\_operation}_i(t, op)$ // $node(t) = i$ //
pre $\equiv status_i(t) =$ active.
eff $\equiv DB_i.submit(t, op)$; $status_i(t) \leftarrow$ blocked.

$\mathbf{end\_operation}_i(t, op)$
pre $\equiv status_i(t) =$ blocked $\wedge\ DB_i.notify(t, op) =$ run.
eff $\equiv$ **if** $node(t) = i$ **then** $status_i(t) \leftarrow$ active
else $status_i(t) \leftarrow$ pre_commit.

$\mathbf{begin\_commit}_i(t)$ // $node(t) = i$ //
pre $\equiv status_i(t) =$ active.
eff $\equiv status_i(t) \leftarrow$ pre_commit;
$participants_i(t) \leftarrow \mathcal{V}_i.availableNodes \setminus \{i\}$;
$sendRMulticast(\langle remote, t, DB_i.WS(t)\rangle, participants_i(t))$.

$\mathbf{end\_commit}_i(t)$ // $t \in T \wedge node(t) = i$ //
pre$\equiv$ $status_i(t) =$ pre_commit $\wedge\ participants_i(t) = \emptyset$.
eff $\equiv sendRMulticast(\langle commit, t\rangle, \mathcal{V}_i.availableNodes \setminus \{i\})$;
$DB_i.commit(t)$; $status_i(t) \leftarrow$ committed;
if $\neg empty(queue_i)$ **then** $remove_i \leftarrow$ true.

$\mathbf{receive\_ready}_i(t, m)$ // $t \in T \wedge node(t) = i$ //
pre $\equiv status_i(t) =$ pre_commit $\wedge\ participants_i(t) \neq \emptyset \wedge$
$m = \langle ready, t, source\rangle \in channel_i$.
eff $\equiv receive_i(m)$; $participants_i(t) \leftarrow participants_i(t) \setminus \{source\}$.

$\mathbf{local\_abort}_i(t)$
pre $\equiv status_i(t) =$ blocked $\wedge\ DB_i.notify(t, op) =$ abort.
eff $\equiv status_i(t) \leftarrow$ aborted; $DB_i.abort(t)$; $remove_i \leftarrow$ true.

$\mathbf{discard}_i(t, m)$ // $t \in T$ //
pre $\equiv status_i(t) =$ aborted $\wedge\ m \in channel_i$.
eff $\equiv receive_i(m)$.

$\mathbf{receive\_commit}_i(t, m)$ // $t \in T \wedge node(t) \neq i$ //
pre $\equiv status_i(t) =$ pre_commit $\wedge\ m = \langle commit, t\rangle \in channel_i$.
eff $\equiv receive_i(m)$; $DB_i.commit(t)$; $status_i(t) \leftarrow$ committed;
if $\neg empty(queue_i)$ **then** $remove_i \leftarrow$ true.

$\mathbf{receive\_remote}_i(t, m)$ // $t \in T \wedge node(t) \neq i$ //
pre $\equiv status_i(t) =$ idle $\wedge\ m = \langle remote, t, WS\rangle \in channel_i$.
eff $\equiv receive_i(m)$;
$insert\_with\_priority(queue_i, \langle t, WS\rangle)$; $remove_i \leftarrow$ true.

$\mathbf{execute\_remote}_i$
pre $\equiv \neg empty(queue_i) \wedge remove_i$.
eff $\equiv aux\_queue \leftarrow \emptyset$;
while $\neg empty(queue_i)$ **do**
$\langle t, WS\rangle \leftarrow first(queue_i)$; $queue_i \leftarrow remainder(queue_i)$;
$conflictSet \leftarrow DB_i.getConflicts(WS)$;
if $\exists\, t' \in conflictSet$: $\neg higher\_priority(t, t')$ **then**
$insert\_with\_priority(aux\_queue, \langle t, WS\rangle)$;
else
$\forall\, t' \in conflictSet$:
if $status_i(t') =$ pre_commit $\wedge\ node(t') = i$ **then**
$sendRMulticast(\langle abort, t'\rangle, \mathcal{V}_i.availableNodes \setminus \{i\})$;
$DB_i.abort(t')$; $status_i(t') \leftarrow$ aborted;
$sendRUnicast(\langle ready, t, i\rangle)$ to $node(t)$;
$DB_i.begin(t)$; $DB_i.submit(t, WS)$; $status_i(t) \leftarrow$ blocked;
$queue_i \leftarrow aux\_queue$; $remove_i \leftarrow$ false.

$\mathbf{receive\_abort}_i(t, m)$ // $t \in T \wedge node(t) \neq i$ //
pre $\equiv status_i(t) \notin \{$aborted, committed$\} \wedge\ m = \langle abort, t\rangle \in channel_i$.
eff $\equiv receive_i(m)$; $status_i(t) \leftarrow$ aborted;
if $\langle t, \perp\rangle \in queue_i$ **then** $queue_i \leftarrow queue_i \setminus \{\langle t, \perp\rangle\}$
else $DB_i.abort(t)$; **if** $\neg empty(queue_i)$ **then** $remove_i \leftarrow$ true.

$\diamond$ **function** $higher\_priority(t, t') \equiv node(t) = j \neq i \wedge (a \vee b)$
(a) $node(t') = i \wedge status_i(t') \in \{$active, blocked$\}$
(b) $node(t') = i \wedge status_i(t') =$ pre_commit $\wedge\ t.priority > t'.priority$

Figure 2: State transition system for the MidO2PL automaton that optimizes the 2PC and allows remote transactions to wait. *pre* indicates precondition and *eff* effects respectively.

local conflictive transactions, whose priority is lower than t, then the remote transaction will never be aborted by the DBMS during its execution. Besides, we have modified the $higher\_priority(t, t')$ function so that any remote transaction executing at a given site will never be rolled back. If a delivered remote transaction has lower priority than any other transaction executing at its delivered site, then it will be inserted in $queue_i$ (sorted by $t.priority$). Hence, this is potential deadlock source, as the protocol permits remote transactions to wait. However, as channels are reliable, all $remote$ messages of enqueued transactions will reach the transaction master sites of all of them.

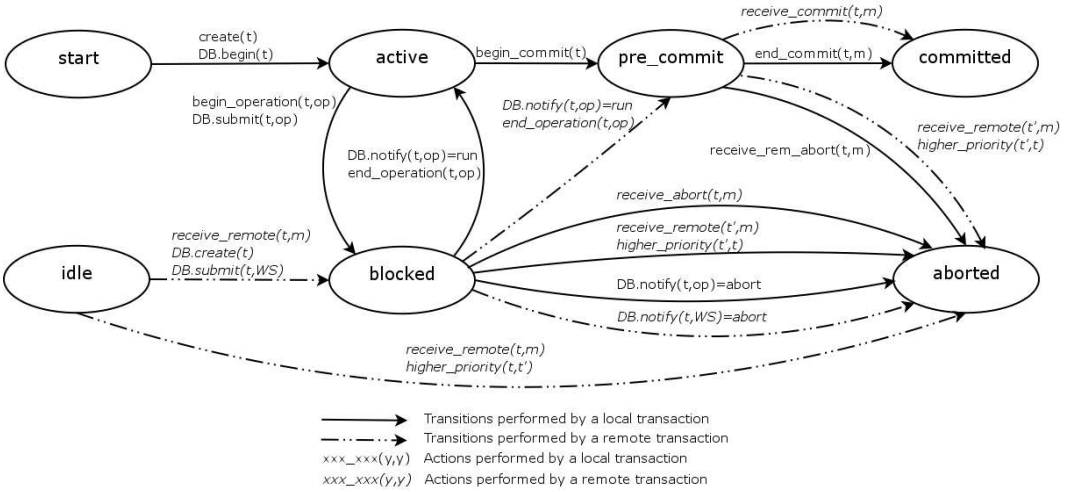

Figure 3: Valid transitions for a given $status_i(t)$ of a transaction $t \in T$.

The abortion decision is performed only at the enqueued transaction master site. Then if a transaction is aborted then all sites will receive the $abort$ message, as channels are reliable, and execute the $receive\_abort_i(t, \langle abort, t \rangle)$.

The MidO2PL must guarantee the atomicity of a transaction, that is, the transaction is either committed at all available sites or is aborted at all sites. If a transaction, t, is in pre_commit state then it is committable from the local DBMS point of view. Therefore, if a local transaction commits at its master site ($node(t) = i$) (i.e. it executes the $end\_commit_i(t)$ action); it multicasts a $commit$ message to each site where its respective remote transaction has been executed. Priority rules (see $higher\_priority(t, t')$ function in Figure 2) ensure that remote transactions are never aborted by a local transaction nor a remote one, provided that there are no unilateral aborts, and eventually reaches the pre_commit state. Thus, by the reliable communication channels the $commit$ message will be eventually delivered; every remote transaction of t will be committed via the execution of the $receive\_commit_j(t, \langle commit, t \rangle)$ action with $j \neq i$. On the contrary, if a transaction t aborts, every remote transaction previously created for t will be aborted. Unilateral aborts are not considered.

In this Section we use the following notation and definitions [Sha93]. For each action in the MidO2PL automaton it is defined an enabled condition (precondition, *pre* in Figure 2), a predicate over the state variables. An action is enabled if its predicate is evaluated to true on the current state. For each action, π, the enabling condition defines a set of state transitions, that is: $\{(p, \pi, q)$, p, q are states; π is an action; p satisfies $pre(\pi)$; and q is the result of executing *eff* (π) in $p\}$. An execution, α, is a sequence of the form $s_0 \pi_1 s_1 \ldots \pi_z s_z \ldots$ where s_z is a state, π_z is an action and every (s_{z-1}, π_z, s_z) is a transition of π_z. A finite execution always finishes in a state, or infinite. Every finite prefix of an infinite execution is a finite execution. A state is reachable if it is the end of a finite execution. All possible executions are sufficient for defining safety properties. Liveness properties require the notion of fair execution. We assume that each MidO2PL action requires weak fairness. Informally, a fair execution will satisfy weak fairness for π, if π is continuously enabled then it will be eventually executed.

The following property formalizes the status transition for a given transaction $t \in T$. It indicates that some $status$ transitions are unreachable, i.e., if $s_k.status_j(t) =$ pre_commit and $s_{k'}.status_j(t) =$ committed with $k' > k$. There is no action in α such that $s_{k''}.status_j(t) =$ aborted with $k' > k'' > k$.

Property 1. *Let $\alpha = s_0 \pi_1 s_1 \ldots \pi_z s_z \ldots$ be an arbitrary execution of the MidO2PL automaton and $t \in T$. Let $\beta = s_0.status_j(t)\, s_1.status_j(t) \ldots s_{z'}.status_j(t)$ be the sequence of status values of t at site $j \in N$, obtained from α by removing the consecutive repetitions of the same $status_j(t)$ value and maintaining the same order apparition in α. The following property holds:*

1. *If* $node(t) = j$ *then* β *is a prefix of the regular expression:* $start \cdot active \cdot (blocked \cdot active)^* \cdot pre\_commit \cdot (committed|aborted)|start \cdot (active \cdot blocked)^+ \cdot aborted$

2. *If* $node(t) \neq j$ *then* β *is a prefix of the regular expression* $idle \cdot blocked \cdot pre\_commit \cdot (committed|aborted)|idle \cdot (blocked|\epsilon) \cdot aborted$*; where* ϵ *denotes the empty string.*

The property is simply proved by induction over the length of α following the preconditions and effects of the MidO2PL actions in Figure 2. A $status$ transition for a t transaction in Property 1 is associated with an operation on the DB module where the transaction was created, i.e. pre_commit to $committed$ involves the $DB.commit(t)$ operation. These aspects are straightforward from the MidO2PL automaton inspection in Figure 2.

The following technical property is needed to prove Lemma 1.

Property 2. *Let* $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ *be an arbitrary execution of the MidO2PL automaton and* $t \in T$*, with* $node(t) = i$*.*

1. *If* $\exists\, j \in N \setminus \{i\}\colon s_z.status_j(t) =$ *committed then* $s_z.status_i(t) =$ *committed.*

2. *If* $\exists\, z' < z\colon s_{z'}.status_j(t) = s_z.status_j(t) =$ *blocked for any* $j \in N$ *then* $\forall\, z''\colon z' < z'' \leq z\colon \pi_{z''} \notin \{receive\_abort_j(t, \langle abort, t\rangle), end\_operation_j(t, WS(t))\}$.

3. *If* $\exists\, z' < z\colon s_{z'}.status_j(t) = s_z.status_j(t) =$ *pre_commit for any* $j \in N$ *then* $\forall\, z''\colon z' < z'' \leq z\colon \pi_{z''} \notin \{receive\_commit_j(t, \langle commit, t\rangle), receive\_abort_j(t, \langle abort, t\rangle)\}$.

4. *If* $s_z.status_i(t) =$ *committed then* $\forall\, j \in N\colon s_z.status_j(t) \in \{blocked, pre\_commit, committed\}$.

Proof. 1. By induction over the length of α. The property holds for the initial state $s_0\colon s_0.status_j(t) =$ idle for all $j \in N$. By hypothesis, assume the property holds at s_{z-1}, the induction is proved for each $(s_{z-1}\pi_z s_z)$ transition of the MidO2PL automaton. By Property 1.2, if $s_z status_j(t) =$ committed, the $status$ does not change. A commit message must be received so that j reaches $s_z.status_j(t) =$ committed. At some prior state to s_z the $receive\_commit_j(t, \langle commit, t\rangle)$ action has been executed. The only action that sent such a message is $end\_commit_i(t)$. By its effects $status_i(t) =$ committed and by Property 1.1, the $status_i(t)$ never changes. Hence, the property holds.

2. We proof this property by contradiction. Let us assume that the $\pi_{z''} = receive\_abort_j(t, \langle abort, t\rangle)$ action. By its effects $s_{z''}.status_j(t) =$ aborted and by Property 1.2 its $status$ never changes in contradiction with our initial assumption. If we suppose that $\pi_{z''} = end\_operation_j(t, WS(t))$ action is executed then, by its effects, $s_{z''}.status_j(t) =$ pre_commit. By Property 1.2 it never goes to $blocked$ again in contradiction with our assumption.

3. We proof this by contradiction. If we assume that $\pi_{z''} = receive\_commit_j(t, \langle commit, t\rangle)$ by this action effects $s_{z''}.status_j(t) =$ committed and by Property 1.2 its $status$ never changes it contradiction with our initial assumption. Besides, if we assume that the executed action is $\pi_{z''} = receive\_abort_j(t, \langle abort, t\rangle)$ then $s_{z''}.status_j(t) =$ aborted and again, by Property 1.2, its $status$ never changes in contradiction with our assumption.

4. If $s_z.status_i(t) =$ committed, by Property 1.1 we have that $\forall\, z' > z\colon s_{z'}.status_i(t) \neq$ aborted and $\langle abort, t\rangle \notin s_{z'}.channel_j$ for all $j \in N$. Thus, the $receive\_abort_j(t, m)$ action is disabled at any state of α. As $s_z.status_i(t) =$ committed then all $j \in N \setminus \{i\}$ has sent the $ready$ message to i which implies, by the $execute\_remote_j$ action effects, that $status_j(t) =$ blocked and it has been submitted to the

$DB_j module$ or $status_j(t) =$ pre_commit if the $end\_operationj(t, WS(t))$ has been already executed. Thus by Property 1.2 $status_j(t) \in \{blocked, pre\_commit, committed, aborted\}$. We must show that $status_j(t) =$ aborted will never be achieved. This is easy to proof since it will need an $abort$ message coming from the transaction master site, and this is not possible since $status_i(t) =$ committed. On the other hand, as we assume that the DB_j module does not abort a submitted remote transaction then it will never execute the $local\_abort_j(t, WS(t))$ and $status_j(t) =$ aborted will never be possible. Hence, the property holds. □

The following lemma, liveness property, states the atomicity of committed transactions.

Lemma 1. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the MidO2PL automaton and $t \in T$ with $node(t) = i$. If $\exists\, j \in N\colon s_z.status_j(t) =$ committed then $\exists\, z' > z\colon s_{z'}.status_j(t) =$ committed for all $j \in N$.*

Proof. If $j \neq i$ by Property 2.1 (or $j = i$) $s_z.status_i(t) =$ committed. By Property 2.4, $\forall\, j \in N \setminus \{i\}\colon s_z.status_j(t) \in \{\text{blocked}, \text{pre\_commit}, \text{committed}\}$. Without loss of generality, assume that s_z is the first state where $s_z.status_i(t) =$ committed and $s_z.status_j(t) =$ pre_commit (if $s_z.status_j(t) =$ blocked it is because of its submission to the DB_j module, by weak fairness of action execution, the $end\_operation_j(t, WS)$ will be eventually invoked and $status_j(t) =$ pre_commit). By the effects of $\pi_z = end\_commit_i(t)$, we have that $\langle commit, t\rangle \in s_z.channel_j$. By Property 2.4 invariance either $s_z.status_j(t) =$ committed or $s_z.status_j(t) =$ pre_commit and $\langle commit, t\rangle \in s_z.channel_j$. In the latter case the $receive\_commit(t, \langle commit, t\rangle)$ action is enabled. By weak fairness assumption, it will be eventually delivered, thus $\exists\, z' > z\colon \pi_{z'} = receive\_commit_j(t, \langle commit, t\rangle)$. By its effects, $s_{z'}.status_j(t) =$ committed. □

We may formally verify that if a transaction is aborted then it will be aborted at all nodes in a similar way. This is stated in the next lemma.

Lemma 2. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the MidO2PL automaton and $t \in T$ with $node(t) = i$. If $s_z.status_i(t) =$ aborted then $\exists\, z' \geq z\colon s_{z'}.status_j(t) =$ idle for all $j \in N \setminus \{i\}$ or $s_{z'}.status_j(t) =$ aborted for all $j \in N$.*

Proof. By inspection of the possible actions that may lead to $status_i(t) =$ aborted, we have the following two cases:

(I) Let us assume, without loss of generality, that s_z is the first state such that $s_z.status_k(t) =$ aborted. Assume that it has been reached by the execution of the $\pi_z = local\_abort_k(t)$ action. This action was enabled because $s_{z-1}.status_i(t) =$ blocked. By Property 1.1 $status_i(t) =$ pre_commit. Therefore, the $begin\_commit_i(t)$ action has never been executed so $\forall j \in N \setminus \{i\}\colon s_z.status_j(t) =$ idle. It is easy to show that this situation will remain the same due to the fact that $status_i(t) =$ aborted is a final state and it will never send anything to any node so as to change $status_j(t) =$ idle.

(II) Let us assume, without loss of generality, that s_z is the first state such that $s_z.status_i(t) =$ aborted. We also assume that this state has been reached by the invocation of the $pi_z = execute\_remote_i$ action being t one of the aborted transactions by this action execution, due to the higher priority of another conflicting transaction, t'. Thus, an $\langle abort, t\rangle$ message will be sent to the rest of nodes, excluding i.

Every $j \neq i$ nodes such as $status_j(t) \notin \{\text{aborted}, \text{committed}\}$ will have the $receive\_abort_j(t, \langle abort, t\rangle)$ action enabled and finally it will be executed, obtaining $status_j(t) =$ aborted. Let us assume that there exists $j \neq i$ nodes such as $status_j(t) \in \{\text{aborted}, \text{committed}\}$. If $status_j(t) =$ aborted then by Property 1.2 this is a final state and the Lemma holds. If we assume that $status_j(t) =$ committed then by Property 2.1 we will have that $status_i(t) =$ committed. The Lemma states that $status_i(t) =$ aborted, and by Property 1.1 this is a stable situation and the Lemma holds.

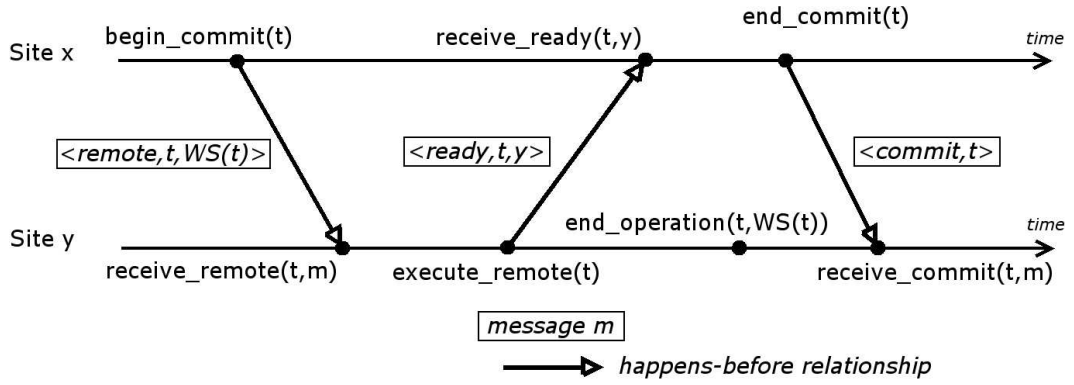

Figure 4: Happens before relationship for a given transaction t between its execution at the master site and the rest of nodes.

□

Before continuing with the correctness proof we have to add a definition dealing with causality between actions. Some set of actions may only be viewed as causally related to another action in any execution α. We denote this fact by $\pi \prec_\alpha \pi'$ (*happens-before* relations [Lam78]). For example, see Figure 4, with $node(t) = i \neq j$, $begin\_commit_i(t) \prec_\alpha receive\_remote_j(t, \langle remote, t, DB_i.WS(t)\rangle)$. This is clearly seen by the effects of the $begin\_commit_i(t)$ action, it sends a $\langle remote, t, DB_i.WS(t)\rangle$ to all $j \in N \setminus \{i\}$. This message will be eventually received by j that enables the $receive\_remote_j(t, \langle remote, t, DB_i.WS(t)\rangle)$ action, since $status_j(t) =$ idle, and, by weak fairness of actions, it will be eventually executed. The following Lemma indicates that a transaction is $committed$ if it has received every $ready$ message from its remote transaction ones. These remote transactions have been created as a consequence of the $execute\_remote_j$ action execution. We are going to add t as a parameter to the $execute\_remote_j$ action provided that t is one of the submitted transactions to the DB_j module by its execution ($execute\_remote_j(t)$).

Lemma 3. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the MidO2PL automaton and $t \in T$ be a committed transaction, $node(t) = i$, then the following happens-before relations hold: $\forall j \in N \setminus \{i\}$: $begin\_commit_i(t) \prec_\alpha receive\_remote_j(t, \langle remote, t, WS(t)\rangle) \prec_\alpha execute\_remote_j(t) \prec_\alpha receive\_ready_i(t, \langle ready, j\rangle) \prec_\alpha end\_commit_i(t) \prec_\alpha receive\_commit_j(t, \langle commit, t\rangle)$.*

Proof. Let $t \in T$, $node(t) = i$, be a committed transaction. By Property 1.1, it has previously been with $status_i(t) =$ active. As $status_i(t) =$ pre_commit has been also achieved, the $begin\_commit_i(t)$ action has been executed. It multicasts to the rest of nodes the $\langle remote, t, DB_i.WS(t)\rangle$ message. $\forall j \in N, j \neq i$ the message is in $channel_j$ and the $receive\_remote_j(t, \langle remote, t, WS(t))$ action will be invoked that inserts the delivered transaction in $queue_j$. When t becomes the transaction with the highest priority among all in $queue_j$ the $execute\_remote_j$ action (this action will be enabled each time a transaction is committed, rolled back or a remote transaction is delivered) will be invoked for t ($execute\_remote_j(t)$), then by its effects it will send the $ready$ message to i and the operation is submitted to the DB_j module. It is important to note that this transaction will not be rolled back by DB_j, recall we are assuming there are not unilateral aborts for remote transactions. By channel reliability it will eventually invoke the $receive\_ready_i(t, \langle ready, j\rangle)$ action at the transaction master site. Respectively, the only action enabled at site i (when $participants_i(t) = \emptyset$) will be the $end\_commit_i(t)$ action. This action will commit the transaction at i and multicast the $\langle commit, t\rangle$ message to the rest of nodes that leads to transaction commitment at the rest of sites. The only actions enabled

for t at j (being $j \in N, j \neq i$) are the $end\_operation_j(t, WS(t))$ or $receive\_commit_j(t, \langle commit, t\rangle)$ actions depending whether $status_j(t) =$ pre_commit or $status_j(t) =$ blocked respectively. If the transaction is still $blocked$, assuming weak fairness for action execution and due to the fact that there are no unilateral aborts, the $end\_operation_j(t, WS(t))$ will be eventually enabled. By its effects, $status_j(t) =$ pre_commit. As the $\langle commit, t\rangle \in channel_j$ then the $receive\_commit_j(t, \langle commit, t\rangle)$ Hence, $\forall\, j \in N \setminus \{i\}$, the Lemma holds following the causal chain. □

The following lemma emphasizes the *happens-before* relationship for remote transactions. It is based on Property 1.2 which establishes the relationship between status transitions for remote transaction to their respective algorithm actions. This will serve in order to set up the relationship for a transaction t, $node(t) = i \neq j$ between the $execute\_remote_j$ that submits t to the DB_j module and the pair $end\_operation_j(t, \langle remote, t, WS(t)\rangle)$ and $receive\_commit_j(t, \langle commit, t\rangle)$ actions. This is needed due to the fact that with the previous lemma there is no point where this causal relationship may be put in.

Lemma 4. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the MidO2PL automaton and $t \in T$ be a committed transaction, $node(t) = i$, then the following happens-before relations hold: $\forall j \in N \setminus \{i\}$: $receive\_remote_j(t, \langle remote, t, WS(t)\rangle) \prec_\alpha execute\_remote_j(t) \prec_\alpha end\_operation_j(t, WS(t)) \prec_\alpha receive\_commit_j(t, \langle commit, t\rangle)$.*

Proof. As t is a committed transaction. By Property 1.1, it has previously been with $status_i(t) =$ active. As $status_i(t) =$ pre_commit has been also achieved, the $begin\_commit_i(t)$ action has been executed. It multicasts to the rest of nodes the $\langle remote, t, DB_i.WS(t)\rangle$ message. The reception of this message will invoke the $receive\_remote_j(t, \langle remote, t, WS(t)\rangle)$ that inserts the t in $queue_j$. When t reaches the highest priority among all delivered transactions at j, it will be submitted to DB_j and the $ready$ message will be sent to i. The collection of all $ready$ messages at i will invoke the $end\_commit_i(t)$ action that multicasts the $\langle commit, t\rangle$ message to all nodes excluding i. The remote transaction will eventually finish ($DB_j.notify(t, WS(t)) =$ run), either before or after the $end\_commit_i(t)$ action, that executes the $end\_operation_j(t, WS(t))$ action. By its effects $status_j(t) =$ pre_commit and by weak fairness action execution the $receive\_commit_j(t, \langle commit, t\rangle)$, as $\langle commit, t\rangle \in channel_j$, will be executed. Then this lemma holds for all remote transactions that finally commit. □

In order to define the correctness of our replication protocol we have to study the global history (H) of committed transactions($C(H)$) [BHG87]. We may easily adapt this concept to our BRP automaton. Therefore, a new auxiliary state variable, H_i, is defined in order to keep track of all the DB_i operations performed on the local DBMS at the i site. For a given α execution of the BRP automaton, $H_i(\alpha)$ plays a similar role as the local history at site i, H_i, as introduced in [BHG87] for the DBMS. In the following, only committed transactions are part of the history, deleting all operations that do not belong to transactions committed in $H_i(\alpha)$. The serialization graph for $H_i(\alpha)$, $SG(H_i(\alpha))$, is defined as in [BHG87]. An arc and a path in $SG(H_i(\alpha))$ are denoted as $t \rightarrow t'$ and $t \xrightarrow{*} t'$ respectively. Our local DBMS produces serializable histories as stated in [BBG$^+$95]. Thus, $SG(H_i(\alpha))$ is acyclic and the history is strict. Thus, for any execution resulting in local histories $H_1(\alpha), H_2(\alpha), \ldots, H_N(\alpha)$ at all sites its serialization graph, $\cup_k\, SG(H_k(\alpha))$, must be acyclic so that conflicting transactions are equally ordered in all local histories. The correctness criterion for replicated data is 1CS, which stands for a serial execution over the logical data unit (although there are several copies of this data among all sites) [BHG87]. Before showing the correctness proof, we need an additional property relating transaction isolation level of the underlying DB modules to the automaton execution event ordering. Let us see first this with an example, assume we have a strict-2PL scheduler as the underlying DB_i, hence a transaction must acquire all its locks before committing. In our case, if we have two conflictive transactions, $t, t' \in T$, such that $t \rightarrow t'$ then the $status_i(t') =$ pre_commit will be subsequent to $status_i(t) =$ committed in the execution. The following property and corollary establish a property about local executions of committed transactions.

Property 3. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be an arbitrary execution of the MidO2PL automaton and $i \in N$. If there exist two transactions $t, t' \in T$ such that $t \xrightarrow{*} t'$ in $SG(H_i(\alpha))$ then $\exists\, z_1 < z_2 < z_3 < z_4$: $s_{z_1}.status_i(t) =$ pre_commit $\wedge$ $s_{z_2}.status_i(t) =$ committed $\wedge$ $s_{z_3}.status_i(t') =$ pre_commit $\wedge$ $s_{z_4}.status_i(t') =$ committed.*

Proof. We firstly consider $t \rightarrow t'$. Thus, exists an operation, op issued by t and another operation, op', issued by t' such that op conflicts with op' and op executes before op'. Hence, by $H_i(\alpha)$ construction we have that $DB_i.notify(t, op) =$ run is prior to $DB_i.notify(t', op') =$ run. However, we have assumed that the DB_i is serializable as shown in [BBG+95]. In such a case, $H_i(\alpha)$ is strict serializable for write and read operations. Therefore, it is required that $DB_i.notify(t, op) =$ run must occur before $DB_i.commit(t)$ and the latter must be prior to $DB_i.notify(t', op') =$ run. The $DB_i.commit(t)$ operation is associated with $status_i(t) =$ committed. Considering t', $DB_i.notify(t', op') =$ run is associated with $status_i(t) \in \{\text{active}, \text{pre\_commit}\}$. Therefore, $\exists\, z_2 < z_3'$ in α such that $s_{z_2}.status_i(t) =$ committed and $s_{z_3'}.status_i(t') \in \{$active, pre_commit$\}$. By Property 1 and by the fact that both transactions commit, $\exists\, z_1 < z_2 < z_3' \leq z_3 < z_4$ in α such that $s_{z_1}.status_i(t) =$ pre_commit $\wedge$ $s_{z_2}.status_i(t) =$ committed $\wedge$ $s_{z_3}.status_i(t') =$ pre_commit $\wedge$ $s_{z_4}.status_i(t') =$ committed. Thus, the property holds for $t \rightarrow t'$. The case $t \xrightarrow{*} t'$ is proved by transitivity. □

The latter property reflects the *happens-before* relationship between the different $status$ of conflictive transactions. The same order must hold for the actions generating the mentioned status. The next corollary expresses this property.

Corollary 1. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the MidO2PL automaton and $i \in N$. If there exist two transactions $t, t' \in T$ such that $t \xrightarrow{*} t'$ in $SG(H_i(\alpha))$ then the following happens-before relations, with the appropriate parameters, hold:*

1. $node(t) = node(t') = i$: $begin\_commit_i(t) \prec_\alpha end\_commit_i(t) \prec_\alpha begin\_commit_i(t') \prec_\alpha end\_commit_i(t', \langle commit, t' \rangle)$.
2. $node(t) = i \wedge node(t') \neq i$: $begin\_commit_i(t) \prec_\alpha end\_commit_i(t) \prec_\alpha end\_operation_i(t', WS(t')) \prec_\alpha receive\_commit_i(t', \langle commit, t' \rangle)$.
3. $node(t) \neq i \wedge node(t') = i$: $end\_operation_i(t, WS(t)) \prec_\alpha receive\_commit_i(t, \langle commit, t \rangle) \prec_\alpha begin\_commit_i(t') \prec_\alpha end\_commit_i(t')$.
4. $node(t) \neq i \wedge node(t') \neq i$: $end\_operation_i(t, WS(t)) \prec_\alpha receive\_commit_i(t, \langle commit, t' \rangle) \prec_\alpha end\_operation_i(t', WS(t')) \prec_\alpha receive\_commit_i(t', \langle commit, t' \rangle)$.

Proof. By Property 3, $\exists\, z_1 < z_2 < z_3 < z_4$: $s_{z_1}.status_i(t) =$ pre_commit $\wedge$ $s_{z_2}.status_i(t) =$ committed $\wedge$ $s_{z_3}.status_i(t') =$ pre_commit $\wedge$ $s_{z_4}.status =$ committed. Depending on $node(t)$ and $node(t')$ values the unique actions that modify their associated $status$ to the given values, by Property 3, are the ones indicated in the Corollary. □

If we have two conflictive transactions, $t, t' \in T$ with $node(t) \neq i$ and $node(t') \neq i$, such that $t \rightarrow t'$ then the $execute\_remote_i(t')$ action that submits t' to the database must be executed after the commitment of t, via the $receive\_commit_i(t, \langle commit, t \rangle)$ action. The next Lemma states how the *happens-before* relationship affects to two committed transactions executing at a remote node.

Lemma 5. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the MidO2PL automaton and $i \in N$. If there exist two committed transactions $t, t' \in T$ with $node(t) = j \neq i$ and $node(t') = k \neq i$ such that $t \xrightarrow{*} t'$ in $SG(H_i(\alpha))$ then the following happens-before relations hold: $\forall\, i \in N \setminus \{k, j\}$: $execute\_remote_i(t) \prec_\alpha receive\_commit_i(t, \langle commit, t \rangle) \prec_\alpha execute\_remote_i(t') \prec_\alpha receive\_commit_i(t', \langle commit, t' \rangle)$.*

Proof. Let $t \in T$ be a committed transaction, with $node(t) \neq i$. By Property 1.2, it has previously been $status_i(t) =$ blocked which has been caused by the $execute\_remote_i$ action, although this action may have been called several times we consider the one that was successfully completed for t ($execute\_remote_i(t)$). This action submits the transaction to the DB_i module and sends the $ready$ message to the transaction master site. Provided that t is a remote transaction, all local conflicting transactions have been rolled back earlier. Moreover, this remote transaction will never be aborted by the underlying database (recall that we do not consider unilateral aborts for remote transaction), only the protocol itself may consider whether to abort or not a remote transaction. As t has been committed, the $end\_operation(t, WS(t))$ had been invoked and finally, with the $commit$ message coming from the master site, the $receive\_commit(t, \langle commit, t\rangle)$ action would have been invoked. Let us assume another committed transaction $t' \in T$, with $node(t') \neq i$, such that $t \xrightarrow{*} t'$. This implies that t' committed after t. By Corollary 1.4 $end\_operation_i(t, WS(t)) \prec_\alpha receive\_commit_i(t, \langle commit, t\rangle) \prec_\alpha end\_operation_i(t', WS(t')) \prec_\alpha receive\_commit_i(t', \langle commit, t'\rangle)$. Now we have to set up the *happens-before* relationship between the $execute\_remote_i(t')$ and the $receive\_commit_i(t, \langle commit, t\rangle)$. If t' is delivered to i after t has committed, via the $receive\_remote_i(t', \langle remote, t'\rangle)$ action, then the lemma will trivially hold for this case. Otherwise, the message is delivered after the $receive\_remote_i(t, \langle remote, t\rangle)$ and before the $receive\_commit_i(t, \langle commit, t\rangle)$ actions execution, then more cases will be taken into account. As $receive\_remote_i(t', \langle remote, t'\rangle)$ will insert into $queue_i$ the delivered transaction. By the effects of this action then the $execute\_remote_i$ action will be invoked. This action will check all the set of conflicting transactions currently executing at DB_i. There will be at least one, t, that conflicts with t'. By the invocation of the $higher\_priority$ function for t and t', it results that t has higher priority than t'. This fact will not change, even though several invocations of the $execute\_remote_i$ action will take place, as long as t does not perform the commit, or, in other words, the execution of the $receive\_commit_i(t, \langle commit, t\rangle)$ action. This can be derived by inspection of the $higher\_priority$ function that returns false if the compared transaction is a non-committed remote transaction currently being executed at the DB_i module. Hence, the lemma holds. □

The same may be applied to two conflictive transactions, $t, t' \in T$ with $node(t) \neq i$ and $node(t') \neq i$, such that $t \rightarrow t'$ then the $execute\_remote$ action that submits t' to the database must be executed after the commitment of t, via the end_commit action. The next lemma states how the *happens-before* relationship affects to a committed transaction executing at a remote node.

Lemma 6. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the MidO2PL automaton and $i \in N$. If there exist two transactions $t, t' \in T$ with $node(t) = i$ and $node(t') \neq i$ such that $t \xrightarrow{*} t'$ in $SG(H_i(\alpha))$ then the following happens-before relations hold: $\forall\, i\colon i \in N : begin\_commit_i(t) \prec_\alpha end\_commit_i(t) \prec_\alpha execute\_remote_i(t') \prec_\alpha end\_operation_i(t', WS(t')) \prec_\alpha receive\_commit_i(t', \langle commit, t'\rangle)$.*

Proof. Let us assume two committed transactions $t, t' \in T$ with $node(t) = i \neq node(t')$ such that $t \xrightarrow{*} t'$. By Corollary 1.2 we have: $begin\_commit_i(t) \prec_\alpha end\_commit_i(t) \prec_\alpha end\_operation_i(t', WS(t')) \prec_\alpha receive\_commit_i(t', \langle commit, t'\rangle)$. As t' is committed remote transaction at i, via Lemma 4, it has executed the following actions: $receive\_remote_i(t', \langle remote, t', WS(t')\rangle) \prec_\alpha execute\_remote_i(t') \prec_\alpha end\_operation_i(t', WS(t')) \prec_\alpha receive\_commit_i(t', \langle commit, t'\rangle)$. Hence, we have to establish the *happens-before* relation between the $execute\_remote_i(t')$ and the $end\_commit_i(t)$ actions. Again, we have two options: if the $receive\_remote_i(t', \langle remote, t', WS(t')\rangle)$ action is executed after the $end\_commit_i(t)$ action then the lemma holds. The remainder case is when the $receive\_remote_i(t', \langle remote, t', WS(t')\rangle)$ and the successful completion of the $execute\_remote_i$ action for t' happens between the $begin\_commit_i(t)$ and the $end\_commit_i(t)$ actions. The successful completion of $execute\_remote_i$ for t' will never happen under this interval. This is easily shown by inspection of the $execute\_remote_i$ action. As $t \xrightarrow{*} t'$, we have that the $getConflicts$ function will return at least t as a conflictive transaction. However, t has $status_i(t) =$ pre_commit and by hypothesis it has been executed before t', this means that $t.priority > t'.priority$ for read-write conflicts and it must wait enqueued. □

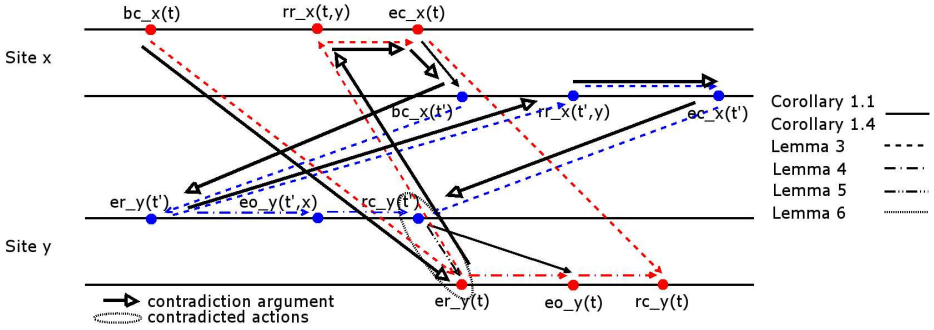

Figure 5: CASE (I): $node(t) = node(t') = x$.

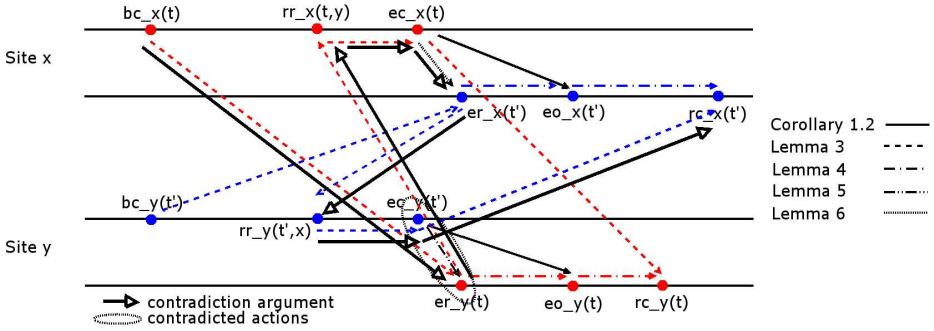

Figure 6: CASE (II): $node(t) = x$ and $node(t') = y$.

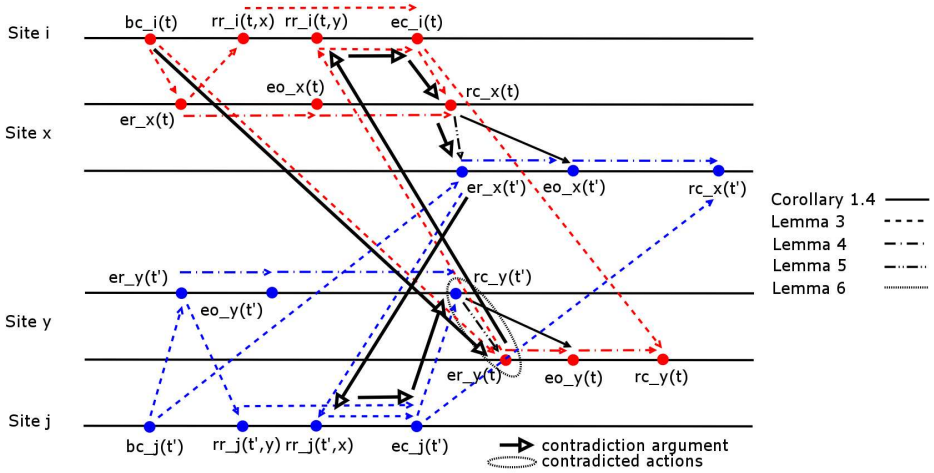

Figure 7: CASE (IV): $node(t) = i$ and $node(t') = j$.

In the following, we prove that the MidO2PL protocol provides 1CS [BHG87].

Theorem 1. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the MidO2PL automaton. The graph $\cup_{k\in N} SG(H_k(\alpha))$ is acyclic.*

Proof. By contradiction. Assume there exists a cycle in $\cup_{k\in N} SG(H_k(\alpha))$. There are at least two different transactions $t, t' \in T$ and two different sites $x, y \in N$, $x \neq y$, such that those transactions are executed in different order at x and y. Thus, we consider (a) $t \xrightarrow{*} t'$ in $SG(H_x(\alpha))$ and (b) $t' \xrightarrow{*} t$ in $SG(H_y(\alpha))$; being $node(t) = i$

and $node(t') = j$. There are four cases under study:

(I) $i = j = x$.

(II) $i = x \wedge j = y$.

(III) $i = j \wedge i \neq x \wedge i \neq y$.

(IV) $i \neq j \wedge i \neq x \wedge i \neq y \wedge j \neq x \wedge j \neq y$.

In the following, we simplify the notation. The action names are shortened, i.e. $begin\_commit_x(t)$ by $bc_x(t)$; $end\_commit_x(t)$ by $ec_x(t)$; as each invocation of the $execute\_remote_x$ action may execute a set of transactions, $K \subseteq T$, we denote it by $er_x(k)$, with $k \in K$; $receive\_ready_x(t, \langle ready, t, l\rangle)$, with $l \in N$, by $rr_x(t, l)$; $end\_operation_x(t, op)$ by $eo_x(t)$; and, $receive\_commit_x(t, \langle commit, t\rangle)$ by $rc_x(t)$.

CASE (I) By Corollary 1.1 for (a): $bc_x(t) \prec_\alpha ec_x(t) \prec_\alpha bc_x(t') \prec_\alpha ec_x(t')$. (i)

By Corollary 1.4 for (b): $eo_y(t') \prec_\alpha rc_y(t') \prec_\alpha eo_y(t) \prec_\alpha rc_y(t)$. Applying Lemmas 4 and 5 for t and t': $er_y(t') \prec_\alpha eo_y(t') \prec_\alpha rc_y(t') \prec_\alpha er_y(t) \prec_\alpha eo_y(t) \prec_\alpha rc_y(t)$. (ii)

For (i), via Lemma 3 for t, we have the following: $bc_x(t) \prec_\alpha er_y(t) \prec_\alpha rr_x(t, y) \prec_\alpha ec_x(t) \prec_\alpha bc_x(t') \prec_\alpha ec_x(t')$. Taking into account Lemma 3 for t' and Lemma 5 for t and t': $bc_x(t) \prec_\alpha er_y(t) \prec_\alpha rr_x(t, y) \prec_\alpha ec_x(t) \prec_\alpha bc_x(t') \prec_\alpha er_y(t') \prec_\alpha rr_x(t', y) \prec_\alpha ec_x(t') \prec_\alpha rc_y(t')$. Therefore, we have that $er_y(t) \prec_\alpha rc_y(t')$ in contradiction with (ii) as it can be seen in Figure 5.

CASE (II) By Corollary 1.2 for (a): $bc_x(t) \prec_\alpha ec_x(t) \prec_\alpha eo_x(t') \prec_\alpha rc_x(t')$. By Lemma 6 for t and t': $bc_x(t) \prec_\alpha ec_x(t) \prec_\alpha er_x(t') \prec_\alpha rc_x(t')$. (i)

By Corollary 1.2 for (b): $bc_y(t') \prec_\alpha ec_y(t') \prec_\alpha eo_y(t) \prec_\alpha rc_y(t)$. Applying Lemma 6 for t' and t: $bc_y(t') \prec_\alpha ec_y(t') \prec_\alpha er_y(t) \prec_\alpha eo_y(t) \prec_\alpha rc_y(t)$. (ii)

By Lemma 3 for t: $bc_x(t) \prec_\alpha er_y(t) \prec_\alpha rr_x(t, y) \prec_\alpha ec_x(t)$, via (i), $\prec_\alpha er_x(t') \prec_\alpha rr_y(t', x) \prec_\alpha ec_y(t') \prec_\alpha rc_x(t')$. Thus $er_y(t) \prec_\alpha ec_y(t')$ in contradiction with (ii), see Figure 6.

CASE (III) As x and y are different sites from the transaction master site, only one of them will be executed in the same order as in the master site. If we take into account the different one with the master site then we will be under assumptions considered in CASE (I).

CASE (IV) By Corollary 1.4 for (a): $eo_x(t) \prec_\alpha rc_x(t) \prec_\alpha eo_x(t') \prec_\alpha rc_x(t')$. Applying Lemmas 4 and 5 for t and t' at x: $er_x(t) \prec_\alpha eo_x(t) \prec_\alpha rc_x(t) \prec_\alpha er_x(t') \prec_\alpha eo_x(t') \prec_\alpha rc_x(t')$. (i)

By Corollary 1.4 for (b): $eo_y(t') \prec_\alpha rc_y(t') \prec_\alpha eo_y(t) \prec_\alpha rc_y(t)$. If we apply Lemmas 4 and 5 for t' and t at y: $er_y(t') \prec_\alpha eo_y(t') \prec_\alpha rc_y(t') \prec_\alpha er_y(t) \prec_\alpha eo_y(t) \prec_\alpha rc_y(t)$. (ii)

By Lemma 3 for t at x and y: $bc_i(t) \prec_\alpha er_y(t) \prec_\alpha rr_i(t, y) \prec_\alpha ec_i(t) \prec_\alpha rc_x(t)$. Via Corollary 1.4 for (a): $bc_i(t) \prec_\alpha er_y(t) \prec_\alpha rr_i(t, y) \prec_\alpha ec_i(t) \prec_\alpha rc_x(t) \prec_\alpha er_x(t') \prec_\alpha rr_j(t', x) \prec_\alpha ec_j(t') \prec_\alpha rc_y(t')$. Therefore, we have that $er_y(t) \prec_\alpha rc_y(t')$ in contradiction with (ii), as it is depicted in Figure 7.

□

5 Snapshot Isolation

Right now, we have only considered that the underlying DBMS provides serializable transaction isolation as pointed out in [BBG+95]. However, SI [BBG+95] is used by most of popular DBMS vendors, like PostgreSQL or Oracle. There have been some recent research dealing with correctness criteria for database replication with SI DBMSs, such as in [LKPMJP05] (1CSI) and GSI [EPZ05]. We will focus on the latter, that states, in few words, that transactions executing at a given site gets the latest snapshot of that site that may not coincide with the latest system snapshot. As SI does not block read operations, we only have to worry about write operations instead. Hence, we may use a DB module providing SI. We may achieve this functionality in the replication protocol presented in the paper, all we have to do is to perform the $getConflicts(WS)$ function over write sets exclusively. Lemma 3 states that the protocol behavior is not influenced by the underlying database. On the other hand, Property 3 asserts what the execution depends on the transaction isolation level that imposes a determined order. Let $t, t' \in T$ be two conflictive committed transactions as $WS(t) \cap WS(t') \neq \emptyset$ the Property 3 holds. It can be shown that with Lemma 3 and Property 3 all writesets are applied at all sites following the same order. This fact does not assure, due to network latency, that a read or write transaction may obtain a different snapshot from the current snapshot, this leads to a similar behavior to GSI [EPZ05].

6 Conclusions

In this paper, we present a middlware replication protocol, MidO2PL, providing database replication. The MidO2PL is 1CS, given that the underlying DBMSs feature serializable transaction isolationas in [BBG+95]. We have formally described and verified its correctness using a formal transition system. This replication protocol has the advantage that no specific DBMS tasks have to be re-implemented (e.g. lock tables, "*a priori*" transaction knowledge). The underlying DBMS performs its own concurrency control and the replication protocol compliments this task with replica control.

The MidO2PL is an eager update everywhere replication protocol, based on the ideas introduced in [CL91]. All transaction operations are firstly performed on its master site, more precisely on its underlying DBMS, and then all updates are grouped and sent to the rest of sites using a reliable multicast. However, our algorithm is liable to suffer distributed deadlock. We have defined a deadlock prevention schema that orders transactions; it is based on the transaction state and a given priority. Besides, as it totally orders transactions, the MidO2PL will know if a transaction may proceed or not. This allows us to get rid of the waiting for applying updates at the rest of nodes. Finally, we propose several modifications to adapt MidO2PL to DBMS providing SI.

References

[BBG+95] H. Berenson, P.A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P.E. O'Neil. A critique of ANSI SQL isolation levels. In Michael J. Carey and Donovan A. Schneider, editors, *SIGMOD Conference*, pages 1–10. ACM Press, 1995.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[CKV01] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[CL91] M.J. Carey and M. Livny. Conflict detection tradeoffs for replicated data. *ACM Trans. Database Syst.*, 16(4):703–746, 1991.

[DSU04] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[EPZ05] S. Elnikety, F. Pedone, and W. Zwaenopoel. Database replication using generalized snapshot isolation. In *SRDS*. IEEE Computer Society, 2005.

[GHOS96] J. Gray, P. Helland, P.E. O'Neil, and D. Shasha. The dangers of replication and a solution. In H. V. Jagadish and Inderpal Singh Mumick, editors, *SIGMOD Conference*, pages 173–182. ACM Press, 1996.

[HT94] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, 1994.

[JPPMKA02] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *ICDCS*, pages 477–484, 2002.

[KPA+03] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE Trans. Knowl. Data Eng.*, 15(4):1018–1032, 2003.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[LKPMJP05] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD Conference*, 2005.

[Lyn96] N.A. Lynch. *Distributed Systems*. Morgan Kaufmann Publishers, 1996.

[Sha93] A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Comput. Surv.*, 25(3):225–262, 1993.