Design of a MidO2PL Database Replication Protocol in the MADIS Middleware Architecture

J.E. Armendáriz*, F.D. Muñoz-Escoí†, J.R. Garitagoitia* and J.R. González de Mendívil*
* Universidad Pública de Navarra, Spain
{enrique.armendariz, joserra, mendivil}@unavarra.es
† Universidad Politécnica de Valencia, Spain
fmunyoz@iti.upv.es

(draft version)

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 [1] that orders transactions by way of a deadlock prevention schema, instead of using the total order transaction delivery obtained by Group Communication Systems (GCSs) techniques, and do not need the 2 Phase Commit (2PC) rule [3]. We formalize its definition as a state transition system [2] and show that it is 1-Copy-Serializable (1CS) [3].**

I. Introduction

Database replication is a way to increase scalability and fault-tolerance of a given system [3]. Although most commercially available solutions and the large majority of deployments use asynchronous updates in a shared nothing architecture, there is an increasing demand for additional guarantees. This demand has been addressed by a set of proposals for eager update replication [5]. Many research works manage the eager update replication by alternative approaches to provide data consistency: by way of distributed lock management as in [3], [1], [6]; or, by means of group communication systems (GCS) [7], [8], [9], [10], [4], [11], [12]. Database replication ranges from middleware-based approaches [13], [8], [9], [4], [11] where replication is controlled in a layer between clients and database replicas, to integrated solutions as in [3], [10], [12] which integrate replica control into the kernel of a DBMS. The advantage of the latter approach is that it is integrated in the same software as the centralized solution and it increases the throughput. On the other hand, middleware-based replication simplifies and restrains the development due to the fact that most database internals remain inaccessible. Furthermore, middleware solutions can be maintained independently of the DBMS and may be used in heterogeneous systems. Middleware replication is useful to integrate new replication functionalities (availability, fault-tolerance, etc.) for applications dealing with database systems that do not provide database replication [13], [8], [11]. In addition, it needs additional support (metadata) for the replica control management performed by the replication protocol; e.g., in the writeset collection. Writeset extraction is a standard mechanism in many commercial replication solutions implemented via triggers. This introduces an additional overhead in the system that affects the transaction response time. Nevertheless, the main goal is to coordinate replica control with concurrency control. Most of the recent solutions must re-implement database features [11]. In a recent approach [4], applications do not have to be modified, they maintain the same interface, like JDBC. Concurrency control is taken at two levels, the underlying database at a given node keeps the transaction isolation level for all active transactions executing on it [14], while the middleware is in charge of conflicts among different replicas. Thus, giving a global transaction isolation level.

In this paper, we present an eager update everywhere replication protocol adapted to our architecture. the idea of the atomic commitment protocol, more precisely the 2 Phase Commit (2PC) protocol, and it is an adaptation of the O2PL protocol proposed by Carey et al. [1]. It is a read-one-write-all-available (ROWAA) [3] protocol, where a transaction is firstly executed on a node and, afterwards, the updates are applied at the rest of nodes, once they are applied they send a message to the node where the transaction was originally executed saying they are willing to commit, and then the transaction is committed via a commit message. Hence, following a 2PC policy. This protocol has been modified, since we do not need to wait for applying the updates at all nodes to say a node is willing to commit. The order imposed by the protocol and a unique priority value assigned to each transaction determine whether a transaction may be ready to commit or not once it is delivered at a site. Underlying this idea, the current version of our protocol assumes that unilateral aborts [15] may not arise. Other extended versions of our protocol[1] were able to manage such kind of aborts. We need no lock management at the middleware level since we rely on the serializable behavior of the underlying DBMS. Besides, it uses basic features present in common DBMS (e.g. triggers, procedures, etc.) to generate the set of metadata needed to maintain each data

[1]Self-reference to be included in the camera-ready version of the paper.

item and conflict detection among transactions. This allows the underlying database to perform more efficiently the task needed to support the replication protocol, and simplifies its implementation. The main contributions of this paper are the following: (a) It provides a formal correctness proof of a variation of the O2PL protocol [1]. (b) Our replication protocol does not rely on strong group communication primitives, provided by GCS [7], in order to determine the order in which transactions are applied.(c) We present an example of a lock-based replication protocol that delegates such a lock management to the underlying DBMS, simplifying the development of the replication protocol in the middleware layer. (d) Some minimal modifications are presented in Section V for providing a snapshot isolation level, instead of the default serializable level assumed in the rest of this document. The paper is organized as follows: Section II introduces the system model, the communication and database module as well as the transaction and execution model. A formal description of this replication protocol, in a failure-free environment is given in Section III. Its correctness proof is shown in Section IV. Finally, conclusions end the paper.

II. System Model and Definitions

We assume a partially synchronous distributed system composed by N sites (or nodes) which communicate among them using reliable multicast featured by a group communication system [7]. It is 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 [3]. We do not consider failures in this paper; a recovery subprotocol has already been described in [2].

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 isolation level [14]. The DBMS 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; or $abort$ due to DBMS internals (e.g. enforcing serialization). 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: $DB.WS(t)$ 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 that site.

Transactions. Users access the system through their closest site to perform transactions by way of several actions. 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, that is used to prevent distributed deadlocks. A transaction t created at site i ($t.site = i$) follows a sequence initiated by $create(t)$ and continued by multiple $begin\_operation(t, op)$, $end\_operation(t, op)$ action pairs 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.

III. Replication Protocol Description

Our protocol is a minimal variation of the O2PL protocol described in [1] and its actions are shown in Figure 1. The local execution of a transaction is shown by a sequence of the following events: $create \cdot (begin\_operation \cdot end\_operation)^+ \cdot begin\_commit$. The latter event multicasts the writeset of the transaction, that is received in the $receive\_remote$ event, and applied later in the $execute\_remote$ event. In the original O2PL some distributed lock management was needed in order to decide whether conflicts arise between transactions. In our case, the local DBMS provides support for detecting conflicts using the *getConflicts* function and an answer can be sent to the transaction master site. This answer may only be a positive acknowledgment (READY message), since in case of conflict detection no reply message is needed. If the answer is sent, the master site will transit to the $receive\_ready$ event. Take into account that if some node does not provide its READY answer, the transaction will be aborted by its master site once it receives the writeset of some conflicting transaction. To this end, it will multicast an ABORT message in its $receive\_remote$ event, that leads all other replicas to the $receive\_abort$ event. If this does not happen, the master site will eventually receive all READY answers and will transit then to the end_commit event, multicasting a COMMIT message that leads all other replicas to the $receive\_commit$ event, committing thus the transaction in all sites.

IV. Correctness Proof

This section contains the proofs (atomicity, in Lemmas 1 and 2, and 1CS, in Theorem 1) of the basic replication protocol (BRP automaton), introduced in Figure 1, in a failure free environment. We only give an outline of some of them due to lack of space.

[2]Self-reference to be provided in camera-ready version, if accepted.

```
Signature:
  {$\forall\, i \in N, t \in T, m \in M, op \in OP$: $\mathbf{create}_i(t), \mathbf{begin\_operation}_i(t, op), \mathbf{end\_operation}_i(t, op), \mathbf{begin\_commit}_i(t), \mathbf{end\_commit}_i(t, m),$
    $\mathbf{local\_abort}_i(t), \mathbf{receive\_remote}_i(t, m), \mathbf{receive\_ready}_i(t, m), \mathbf{receive\_commit}_i(t, m), \mathbf{receive\_abort}_i(t, m), \mathbf{execute\_remote}_i,$
    $\mathbf{discard}_i(t, m)$}.
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 ≡ $status_i(t) =$ start.
eff ≡ $DB_i.begin(t)$; $status_i(t) \leftarrow$ active.

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

$\mathbf{end\_operation}_i(t, op)$
pre ≡ $status_i(t) =$ blocked $\wedge\ DB_i.notify(t, op) =$ run.
eff ≡ 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 ≡ $status_i(t) =$ active.
eff ≡ $status_i(t) \leftarrow$ pre_commit;
        $participants_i(t) \leftarrow \mathcal{V}_i.availableNodes \setminus \{i\}$;
        $sendRMulticast(\langle remote, t, DB_i.WS(t)\rangle, participants_i(t))$.

$\mathbf{end\_commit}_i(t, m)$ // $t \in T \wedge node(t) = i$ //
pre ≡ $status_i(t) =$ pre_commit $\wedge\ participants_i(t) = \{source\} \wedge$
        $m = \langle ready, t, source\rangle \in channel_i$.
eff ≡ $receive_i(m)$; // Remove m from channel
        $participants_i(t) \leftarrow participants_i(t) \setminus \{source\}$;
        $sendRMulticast(\langle commit, t\rangle, \mathcal{V}_i.availableNodes \setminus \{i\})$;
        $DB_i.commit(t)$; $status_i(t) \leftarrow$ 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 ≡ $status_i(t) =$ pre_commit $\wedge\ \|participants_i(t)\| > 1 \wedge$
        $m = \langle ready, t, source\rangle \in channel_i$.
eff ≡ $receive_i(m)$; $participants_i(t) \leftarrow participants_i(t) \setminus \{source\}$.

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

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

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

$\mathbf{receive\_remote}_i(t, m)$ // $t \in T \wedge node(t) \neq i$ //
pre ≡ $status_i(t) =$ idle $\wedge\ m = \langle remote, t, WS\rangle \in channel_i$.
eff ≡ $receive_i(m)$; $conflictSet \leftarrow DB_i.getConflicts(WS)$;
        if $\exists\, t' \in conflictSet$: $\neg higher\_priority(t, t')$ then
          $insert\_with\_priority(queue_i, \langle t, WS\rangle)$;
        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.

$\mathbf{execute\_remote}_i$
pre ≡ $\neg empty(queue_i) \wedge remove_i$.
eff ≡ $aux\_queue \leftarrow \emptyset$;
        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 ≡ $status_i(t) \notin$ {aborted, committed} $\wedge\ m = \langle abort, t\rangle \in channel_i$.
eff ≡ $receive_i(m)$; $status_i(t) \leftarrow$ 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.

◇ 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 $\wedge\ t.priority > t'.priority$
```

Fig. 1. State transition system for the Basic Replication Protocol (BRP) enhanced to optimize the 2PC and allowing remote transactions to wait.

In this Section we use the following notation and definitions [2]. For each action in the BRP automaton it is defined an enabled condition (precondition, *pre* in Figure 1), 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: $\{(p, q)$, p, q are states; p satisfies $pre(\pi)$; and q is the result of executing $(\mathit{eff}(\pi))$ 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}, s_z) is a transition of π_z, also denoted (s_{z-1}, π_z, s_z). An execution may be finite, if it ends 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. Progress properties require the notion of fair execution. We assume that each BRP action requires weak fairness. Informally, a fair execution satisfies weak fairness for π if π is continuously enabled then it will be eventually executed.

The following property formalizes the $status$ transition of transactions. 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 $\alpha = s_o\pi_1 s_1 \ldots \pi_z s_z \ldots$ be an arbitrary execution of the BRP 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 $status$ transitions 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 finite prefix of the regular expression: $start \cdot active \cdot (blocked \cdot active)^* \cdot pre\_commit \cdot (committed|aborted)|start \cdot (active.blocked)^+ \cdot aborted$.
2) If $node(t) \neq j$ then β is a finite 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 BRP actions in Figure 1. 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 BRP automaton inspection in Figure 1 and the $status$ transition of transactions. The following technical property is needed to prove Lemma 1.

Property 2: Let $\alpha = s_0\pi_1s_1\ldots\pi_zs_z\ldots$ be an arbitrary execution of the BRP 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 $s_z.status_i(t) =$ committed then $\forall j \in N\colon \exists z' < z\colon s'_z.status_j(t) =$ pre_commit.
3) If $\exists z' < z\colon s_{z'}.status_j(t) = s_z.status_j(t) =$ pre_commit for any $j \in N$ then $\forall z' < z'' \leq z\colon \pi_{z''} \notin \{receive\_commit_j(t, m), receive\_abort_j(t, m)\}$.
4) If $\exists z' < z\colon s_{z'}.status_j(t) = s_z.status_j(t) =$ blocked for any $j \in N$ then $\forall z' < z'' \leq z\colon \pi_{z''} \notin \{receive\_abort_j(t, m), end\_operation_j(t, op)\}$.
5) If $s_z.status_i(t) =$ committed then $\forall j \in N \setminus \{i\}\colon s_z.status_j(t) \in \{$blocked,pre_commit, committed$\}$.

Proof: This property is shown by induction over the length of α. We only show the last one in order to highlight its importance. If $s_z.status_i(t) =$ committed, by Property 1.1 we have that $\forall s_{z'} \in \alpha\colon s_{z'}.status_i(t) \neq$ aborted $\wedge \langle abort, t\rangle \notin s_{z'}.channel_j$ for all $j \in N$. Thus, the $receive\_abort_j(t, m)$ action is disabled at any state of α. By Property 2.2, we have that $\exists z' < z\colon s_{z'}.status_j(t) =$ pre_commit for all $j \in N$. By Property 2.3, either $s_z.status_j(t) =$ pre_commit or $\exists z' < z'' < z\colon \pi_{z''} = receive\_commit_j(t)$. In the latter case, $s_{z''}.status(t) =$ committed and by Property 1.2 its associated $status$ never changes. Therefore, $s_z.status_j(t) \in \{$pre_commit, committed$\}$.

If $s_z.status_i(t) =$ committed, by Property 1.1 we have that $\forall s_{z'} \in \alpha\colon s_{z'}.status_i(t) \neq$ aborted $\wedge \langle abort, t\rangle \notin s_{z'}.channel_j$ for all $j \in N$. Thus, the $receive\_abort_j(t, m)$ action is disabled at any state of α. By Property 2.2, we have that $\exists z' < z\colon s_{z'}.status_j(t) =$ pre_commit for all $j \in N$. By Property 2.3, either $s_z.status_j(t) =$ pre_commit or $\exists z' < z'' < z\colon \pi_{z''} = receive\_commit_j(t)$. In the latter case, $s_{z''}.status_j(t) =$ committed and by Property 1.2 its associated $status$ never changes. By Property 2.4, either $s_z.status_j(t) =$ blocked or the $end\_operation_j(t, op)$ action, by weak fairness of actions, will be eventually executed. This causes that $status_j(t) =$ pre_commit and as $\langle commit, t\rangle \in channel_j$, the $receive\_commit_j(t, m)$ will be executed. Therefore, $s_z.status_j(t) \in \{$blocked, pre_commit, committed$\}$. ■

However, the invariant of Properties 2.3 and 2.4 establish one abortion point for remote transactions which is the transaction master site. Our modification establishes that only at the transaction master site is decided whether a transaction aborts or not. The following lemma, liveness property, states the atomicity of committed transactions.

Lemma 1: Let $\alpha = s_0\pi_1s_1\ldots\pi_zs_z\ldots$ be a fair execution of the BRP 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 2.5, $\forall j \in N \setminus \{i\}\colon s_z.status(t) \in \{$pre_commit, committed$\}$. Without loss of generality, assume that s_z is the first state where $s_z.status_i =$ committed and $s_z.status_j(t) =$ pre_commit. By the effects of $\pi_z = end\_commit_i(t, m)$, we have that $\langle commit, t\rangle \in s_z.channel_j$. By Property 2.5 invariance either $s_z.status_j(t) =$ committed or $s_z.status_j(t) =$ pre_commit $\wedge \langle commit, t\rangle \in s_z.channel_j$. In the latter case the $receive\_commit(t, m)$ action is enabled. By weak fairness assumption, it will be eventually delivered, thus $\exists z' > z\colon \pi_{z'} = receive\_commit_j(t, m)$. 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 following Lemma.

Lemma 2: Let $\alpha = s_0\pi_1s_1\ldots\pi_zs_z\ldots$ be a fair execution of the BRP automaton and $t \in T$ with $node(t) = i$. If $s_z.status_i(t) =$ aborted then $\exists z' \geq z\colon s_{z'}.status =$ idle for all $j \in N \setminus \{i\} \vee s_{z'}.status_j(t) =$ aborted for all $j \in N$.

Before continuing with the correctness proof we have to add a definition dealing with causality (*happens-before* relations [16]) 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 $\pi_i \prec_\alpha \pi_j$. For example, with $node(t) = i \neq j$, $end\_operation_j(t, WS(t)) \prec_\alpha receive\_ready_i(t, m)$. This is clearly seen by the effects of the $end\_operation_j(t, WS(t))$ action, it sends a $\langle ready, t, j\rangle$ to i. This message will be eventually received by the transaction master site that enables the $receive\_ready_i(t, m)$ action, since $status_i(t) =$ pre_commit, 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 $receive\_remote_j(t, m)$ action execution.

Lemma 3: Let $\alpha = s_0\pi_1s_1\ldots\pi_zs_z\ldots$ be an arbitrary execution of the BRP automaton and $t \in T$ be a committed transaction, $node(t) = i$, then the following happens-before relations hold for the appropriate parameters: $\forall j \in N \setminus \{i\}\colon begin\_commit_i(t) \prec_\alpha receive\_remote_j(t, m) \prec_\alpha$

$end\_operation_j(t, op) \prec_\alpha end\_commit_i(t, m) \prec_\alpha receive\_commit_j(t, m)$.

In order to define the correctness of our replication protocol we have to study the global history (H) of committed transactions($C(H)$) [3]. 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 [3] 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 [3]. 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 ANSI serializable histories [14]. Thus, $SG(H_i(\alpha))$ is acyclic and the history is strict. 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) [3]. 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.

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 a fair execution of the BRP 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 $t \rightarrow t'$. $\exists\, op_t < op'_{t'}$ and op_t conflicts with $op'_{t'}$. Hence, by construction of $H_i(\alpha)$: $DB_i.notify(t, op) =$ run $< DB_i.notify(t', op') =$ run. This fact makes true $op_t < op'_{t'}$. However, we have assumed that the DB_i is serializable and satisfies ANSI serializable transaction isolation [14]. 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 $< DB_i.commit(t) < DB_i.notify(t', op') =$ 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 $status_i(t) \in$ {active, 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 BRP 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) $node(t) = node(t') = i$: $begin\_commit_i(t) \prec_\alpha end\_commit_i(t, m) \prec_\alpha begin\_commit_i(t') \prec_\alpha end\_commit_i(t', m')$.
2) $node(t) = i \wedge node(t') \neq i$: $begin\_commit_i(t) \prec_\alpha end\_commit_i(t, m) \prec_\alpha end\_operation_i(t', WS') \prec_\alpha receive\_commit_i(t', m')$.
3) $node(t) \neq i \wedge node(t') = i$: $end\_operation_i(t, WS) \prec_\alpha receive\_commit_i(t, m) \prec_\alpha begin\_commit_i(t') \prec_\alpha end\_commit_i(t', m')$.
4) $node(t) \neq i \wedge node(t') \neq i$: $end\_operation_i(t, WS) \prec_\alpha receive\_commit_i(t, m) \prec_\alpha end\_operation_i(t', WS') \prec_\alpha receive\_commit_i(t', m')$.

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 whose effects modify their associated $status$ are the ones indicated in the Property 3. ■

In the following, we prove that the BRP protocol provides 1CS [3].

Theorem 1: Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the BRP 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) $i = j \wedge i \neq x \wedge i \neq y$; and, (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 $bc_x(t)$; $end\_commit_x(t, m)$ by $ec_x(t)$; $receive\_remote_x(t, m)$ by $rr_x(t)$; $end\_operation_x(t, op)$ by $eo_x(t)$; and $receive\_commit_x(t, m)$ 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): $eo_y(t') \prec_\alpha rc_y(t') \prec_\alpha eo_y(t) \prec_\alpha rc_y(t)$ (ii). By Lemma 3 for t: $bc_x(t) \prec_\alpha rr_y(t) \prec_\alpha eo_y(t) \prec_\alpha ec_x(t)$ followed by (i) $\prec_\alpha$ (via Lemma 3) $bc_x(t') \prec_\alpha rr_y(t') \prec_\alpha eo_y(t')$. Thus, $eo_y(t) \prec_\alpha eo_y(t')$ in contradiction with (ii).

CASE (II) By Corollary 1.2 for (a): $bc_x(t) \prec_\alpha ec_x(t) \prec_\alpha$

$eo_x(t') \prec_\alpha rc_x(t')$ (i). By Corollary 1.2 for (b): $bc_y(t') \prec_\alpha ec_y(t') \prec_\alpha eo_y(t) \prec_\alpha rc_y(t)$ (ii). By Lemma 3 for t: $bc_x(t) \prec_\alpha rr_y(t) \prec_\alpha eo_y(t) \prec_\alpha ec_x(t)$; by $(i) \prec_\alpha eo_x(t')$, and by Lemma 3 for t', $\prec_\alpha ec_y(t')$. Thus $eo_y(t) \prec_\alpha ec_y(t')$ in contradiction with (ii).

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): $eo_x(t) \prec_\alpha rc_x(t) \prec_\alpha eo_x(t') \prec_\alpha rc_x(t')$ (i). By Corollary 1.4 for (b): $eo_y(t') \prec_\alpha rc_y(t') \prec_\alpha eo_y(t) \prec_\alpha rc_y(t)$ (ii). By Lemma 3 for t at site y: $bc_i(t) \prec_\alpha rr_y(t) \prec_\alpha eo_y(t) \prec_\alpha ec_i(t) \prec_\alpha rc_y(t)$. Applying Lemma 3 for t at x: $bc_i(t) \prec_\alpha rr_x(t) \prec_\alpha eo_x(t) \prec_\alpha ec_i(t) \prec_\alpha rc_x(t)$. Therefore, we have that $eo_y(t) \prec_\alpha rc_x(t)$. Let us apply Lemma 3 for t' at y: $bc_j(t') \prec_\alpha rr_y(t') \prec_\alpha eo_y(t') \prec_\alpha ec_j(t') \prec_\alpha rc_y(t')$ and for site x: $bc_j(t') \prec_\alpha rr_x(t') \prec_\alpha eo_x(t') \prec_\alpha ec_j(t') \prec_\alpha rc_x(t')$. Thus, we have $eo_x(t') \prec_\alpha rc_y(t')$. Taking into account Lemma 3 for t we have: $eo_y(t) \prec_\alpha rc_x(t)$ (via (i)) $\prec_\alpha eo_x(t')$ (via Lemma 3 for t') $\prec_\alpha rc_y(t')$, in contradiction with (ii). ■

V. Snapshot Isolation

Right now, we have only considered that the underlying DBMS provides ANSI serializable transaction isolation. However, Snapshot Isolation (SI) [14] is a very popular solution used by many DBMS vendors. There has been some recent research about this fact for database replication in order to provide something similar to 1CS [3] for SI. This approach is introduced in [4] where the 1CSI is presented.This transaction isolation level does not block read operations, so we only have to worry for write operations instead. Hence, we may use a DB module providing SI. We may achieve this functionality in our replication protocol. 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 that 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.

VI. Conclusions

In this paper, we present a Basic Replication Protocol (BRP) for the SampleOne middleware architecture, which provides a JDBC interface but enhanced to support replication by way of different replication protocols. This replication protocol is 1CS, given that the underlying DBMSs feature ANSI serializable transaction isolation. 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 BRP is an eager update everywhere replication protocol, based on the ideas introduced in [1]. 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, based on the transaction state and a given priority; besides, the information needed by the deadlock prevention schema is entirely local, i.e. no additional communication is needed among nodes. We have followed an optimistic approach, since we automatically abort transactions with lower priority because we suppose that we are working in a low conflict environment. Finally, we propose several modifications to this basic version so as to improve its response time and abortion rate.

References

[1] M. J. Carey and M. Livny, "Conflict detection tradeoffs for replicated data.," *ACM Trans. Database Syst.*, vol. 16, no. 4, pp. 703–746, 1991.

[2] A. U. Shankar, "An introduction to assertional reasoning for concurrent systems.," *ACM Comput. Surv.*, vol. 25, no. 3, pp. 225–262, 1993.

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

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

[5] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha, "The dangers of replication and a solution.," in *SIGMOD Conference*, H. V. Jagadish and I. S. Mumick, Eds. 1996, pp. 173–182, ACM Press.

[6] J. Holliday, R. C. Steinke, D. Agrawal, and A. El Abbadi, "Epidemic algorithms for replicated databases.," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 5, pp. 1218–1238, 2003.

[7] G. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: a comprehensive study.," *ACM Comput. Surv.*, vol. 33, no. 4, pp. 427–469, 2001.

[8] J. Esparza-Peidro, F.D. Muñoz-Escoí, L. Irún-Briz, and J.M. Bernabéu-Aubán, "RJDBC: a simple database replication engine," in *Proc. of the 6th Int'l Conf. Enterprise Information Systems (ICEIS'04)*, 2004.

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

[10] B. Kemme and G. Alonso, "A new approach to developing and implementing eager database replication protocols.," *ACM Trans. Database Syst.*, vol. 25, no. 3, pp. 333–379, 2000.

[11] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso, "Scalable replication in database clusters.," in *DISC*, M. Herlihy, Ed. 2000, vol. 1914 of *Lecture Notes in Computer Science*, pp. 315–329, Springer.

[12] S. Wu and B. Kemme, "Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation.," in *ICDE*. 2005, pp. 422–433, IEEE Computer Society.

[13] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "C-JDBC: Flexible database clustering middleware.," in *USENIX Annual Technical Conference, FREENIX Track*. 2004, pp. 9–18, USENIX.

[14] 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 *SIGMOD Conference*, M. J. Carey and D. A. Schneider, Eds. 1995, pp. 1–10, ACM Press.

[15] F. Pedone, *The database state machine and group communication issues (Thèse N. 2090)*, Ph.D. thesis, École Polytecnique Fédérale de Lausanne, Lausanne, Switzerland, 1999.

[16] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.