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

J.E. Armendáriz-Iñigo*, F.D. Muñoz-Escof†, J.R. Garitagoitia* and J.R. González de Mendivil*

* Universidad Pública de Navarra, Spain
† Universidad Politécnica de Valencia, Spain
{enrique.armendariz, joserra, mendivil}@unavarra.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 [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) [2] techniques, and do not need the 2 Phase Commit (2PC) rule [3]. We formalize its definition as a state transition system [4] and show that it is 1-Copy-Serializable (1CS) [3].

I. 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 [1], [3]. 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. [5], [6] 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 [3] that implies a serial execution over the logical data unit although there are many physical copies. In [6] a middleware architecture is introduced providing 1CS by way of a GCS [2]. The total order delivery of messages establishes the order in which transactions are executed at all available sites [2]. 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 [3], 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 [6]. Besides, the order of committed transactions has nothing to do with the information associated to them, 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 [1] adapted to our middleware architecture, that only needs a reliable multicast to the rest of sites [2]. 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 [4]. Hence, the formal description of the algorithm and the sketch of its correctness proof is the main focus of the paper. The rest of this paper is organized as follows: The system model is introduced in Section II. The formalization of our MidO2PL protocol is presented in Section III. Section IV shows its correctness proof. Finally, conclusions end the paper.

II. SYSTEM MODEL AND DEFINITIONS

As we want to emphasize the explanation of MidO2PL and its correctness proof, an abstraction of the MADIS middleware architecture [5] in a failure free environment is presented as the system model. Details about failures and the recovery process are given in [7]. The system (see Figure 1) is composed by \( N \) sites (or nodes) which communicate by means of unicast and multicast primitives provided by a GCS [2]. As we are assuming a failure free model, it guarantees that all messages are uniformly and reliably multicast to all available sites. We assume a fully replicated system. Each site contains a copy of the entire database and executes transactions on its data copies. Our middleware provides a JDBC interface to applications [5]. An application submits transactions 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]. Actions in Figure 1 are shown with arrows, they describe how components interact...
with each other. Actions may easily be ported to the particular GCS primitives and JDBC methods.

**Database.** We assume that the DBMS ensures ACID properties of transactions and satisfies the serializable transaction isolation level. The DBMS, as it is depicted in Figure 1 offers an abstraction of a JDBC interface to the middleware [5]. After an operation submission in the context of the given transaction, the `DB.notify(t, op)` informs about the successful completion of an operation (run); or, abort due to DBMS internals, a transaction will only be unilaterally aborted if it is involved in a local deadlock. As a remark, we also assume that after the successful completion of a submitted operation, a transaction may commit at any time. We have added two functions which are not provided by DBMSs, but may easily be built by standard database 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 given by `getConflicts(WS(t)) = \{ t' \in T: (WS(t') \cup RS(t')) \cap WS(t) \neq \emptyset \}`, where `T` is the set of transactions in the system.

**Transactions.** Each transaction has an identifier including the information about the site where it was first created (`t.node`), called its transaction master site, in order to know if it is a local or a remote transaction. It also contains a global unique priority value (`t.priority`) based on transaction information (e.g. number of restarts, the node identifier, etc.). A transaction `t` created at site `i` (`t.node = 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. 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

The MidO2PL is a Read One Write All Available (ROWAA) [3] 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, hence there is no need to implement any database specific data structure at the middleware. 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 [8], 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 at 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 [4]. 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 (pre) is enabled. The effects (eff) modify the state of the system as stated by the sequence of instructions included in the action effects, enabling and disabling actions. Actions are atomically executed and weak fairness for each action execution is assumed. We will start with the states defined for this replication protocol, each one is initialized with a given value. Each state and action is subscribed 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(t)` variable indicates the execution state of a given transaction. The `participants(t)` variable keeps track of the sites that have not yet sent the ready message to the master site of `t`. `V_i` is the system current view, with a failure-free assumption, is \(\{0, N\}\). 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 their associated priorities are 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 actions are 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)`

![Fig. 1. Main components of the system.](image-url)
starts the interaction of the rest of nodes. This set of actions are entirely self-explanatory from inspection of Figure 2.

The key action of our replication protocol is the execute_remote, one. It is invoked, each time a transaction is delivered as well as when a transaction is committed or aborted given that queue, is non-empty. 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 function, it depends on the state of the transaction (status(t)) and its priority. A new delivered conflicting remote transaction whose priority is lower than any other executing transaction, will be inserted again in queue. 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 conflicting transaction and submits t to DB_i. Aborted local transactions in pre_commit state will multicast an abort message to the rest of sites. The finalization of the remote transaction (end_operation(t, op)) changes its status(t) = pre_commit, i ≠ t.node. MidO2PL 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(t)).

IV. CORRECTNESS PROOF

This section contains a sketch of the correctness proofs (atomicity and 1CS) of the MidO2PL automaton (Figure 2). A more detailed description is given in [9]. In this Section we continue using the notation and definitions used in [4]. For each action, π, the enabling condition defines a set of state transitions, that is: \{ (p, π, q), p, q are states; π is an action; p satisfies pre(π); and q is the result of executing eff(π) in p \}. An execution, α, is a sequence of the form \( s_0 \pi_1 s_1 \cdots \pi_2 s_2 \cdots \) where \( s_0 \) is a state, \( \pi_i \) is an action and every \( (s_{i-1}, \pi_i, s_i) \) is a transition of \( \pi_i \). 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 finite 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 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. The following technical property is needed to prove the atomicity.

**Property 1:** Let \( \pi = s_0 \pi_1 s_1 \cdots \pi_2 s_2 \cdots \) be an arbitrary execution of the MidO2PL state transition system and \( t \in T \), with t.node = i.

1) If \( \exists j \in N \setminus \{i\}: \text{commit}(s_j(t)) = \text{committed} \) then \( \text{commit}(s_j(t)) = \text{committed} \).

2) If \( \exists j < z: \text{commit}(s_j(t)) = \text{committed} \) for any \( j \in N \setminus \{i\} \) then \( \forall z'' : z'' < z'' < z : \pi_z'' \notin \{\text{receive_abort}(t, m), \text{end_operation}(t, W.S.ops)\} \).

3) If \( \exists z' < z: \text{commit}(s_z(t)) = \text{committed} \) for every \( j \in N \setminus \{i\} \) then \( \forall z'' : z'' < z'' < z : \pi_z'' \notin \{\text{commit}(t, m), \text{receive_abort}(t, m)\} \).

4) If \( \text{commit}(s_j(t)) = \text{committed} \) then \( \forall j \in N: s_j \text{status}(t) \in \{\text{committed, pre_commit, committed}\} \).

The following Lemma, liveness property, states the atomicity of committed transactions. In a similar way, it may formally be verified that if a transaction is aborted then it will be aborted at all nodes.

**Lemma 1:** Let \( \alpha = s_0 \pi_1 s_1 \cdots \pi_2 s_2 \cdots \) be a fair execution of MidO2PL and \( t \in T \) with t.node = i. If \( \exists j \in N: \text{commit}(s_j(t)) = \text{committed} \) then \( \exists z' > z: \text{commit}(s_j(t)) = \text{committed} \) for all \( j \in N \).

**Proof:** If \( j \neq i \) by Property 1.1 (or \( j = i \)) \text{commit}(s_j(t)) = \text{committed}. By Property 1.4, \( \forall j \in N \setminus \{i\}: s_j \text{status}(t) \in \{\text{committed, pre_commit, committed}\} \) without loss of generality, assume that \( s_0 \) is the first state where \( s_j \text{status}(t) = \text{committed} \) and \( s_0 \text{status}(t) = \text{pre_commit} \) (if \( s_0 \text{status}(t) = \text{block} \) it is because of its submission to the DB module, due to the execute_remote for t). By weak fairness of action execution, the end_operation(t, W.S.ops) will be eventually invoked and \( s_j \text{status}(t) = \text{pre_commit} \).

By the effects of \( \pi_z = \text{end_commit}(t) \), we have that \( \text{commit}(t) \in s_z \text{channel} \). By Property 1.4 invariance either \( s_j \text{status}(t) = \text{committed} \) or \( s_j \text{status}(t) = \text{pre_commit} \) and \( \text{commit}(t) \in s_z \text{channel} \). In the latter case the receive_commit(t, m) action is enabled. By weak fairness assumption, it will be eventually delivered, thus \( \exists z' > z: \text{commit}(s_j(t)) = \text{committed} \).

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 \( \alpha \). We denote this fact by \( \pi_{\alpha} \prec_{\alpha} \pi' \) (happens-before relation [10]). MidO2PL has some set of actions that happens-before other actions, i.e. they are causally related. For example, assuming \( t \) is a committed transaction with t.node = i ≠ j, the following happens-before relationship \( \text{begin_commit}(t) \prec_{\alpha} \text{receive_remote}(t, m) \) is held. This is clearly seen by the effects of the begin_commit(t) action: it sends a \( \langle\text{remote}, t, DB_i, WS(t)\rangle \) to all \( j \in N \setminus \{i\} \). This message will be eventually received by \( j \) that enables the receive_remote(t, m) action, since status(t) = idle, and, by weak fairness of actions, it will be eventually executed. The next two Lemmas indicate that a transaction is committed if it has received every ready message from its remote transaction ones and the actions invoked by a remote transaction at a remote site respectively.

**Lemma 2:** Let \( \alpha = s_0 \pi_1 s_1 \cdots \pi_2 s_2 \cdots \) be a fair execution of the MidO2PL state transition system and \( t \in T \) be a committed transaction, t.node = i, then the following happens-before relations hold: \( \forall j \in N \setminus \{i\} : \) (1) begin_commit(t) \( \prec_{\alpha} \) receive_remote(t, m) \( \prec_{\alpha} \)
Fig. 2. State transition system for the MidO2PL automaton that optimizes the 2PC rule and allows remote transactions to wait. pre indicates precondition and eff effects respectively.

execute_remote_i(t) <_a receive_ready_i(t, m) <_a end_commit_i(t) <_a receive_commit_i(t, m); and,
(2) receive_remote_i(t, m) <_a execute_remote_i(t) <_a end_operation_i(t, WS.ops) <_a receive_commit_i(t, m)

As we have mentioned, the strongest correctness criterion for replicated data is 1CS. Thus, for any execution resulting in local histories \( H_1(\alpha), H_2(\alpha), \ldots, H_N(\alpha) \) at all sites its serialization graph, \( \bigcup_k SG(H_k(\alpha)) \), must be acyclic so that conflicting transactions are equally ordered in all local histories [3]. An arc and a path in \( SG(H_i(\alpha)) \) are denoted as \( t \rightarrow t' \) and \( t \rightsquigarrow t' \) respectively. 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 \( t \rightarrow t' \) [3] then the \( status_i(t') = pre\text{-commit} \) will be subsequent to \( status_i(t) = committed \) in the execution. The following property establishes a property about local executions of committed transactions.

Property 2: Let \( \alpha = \pi_0 \pi_1 \pi_2 \ldots \pi_s \ldots \) be a fair execution of the MidO2PL state transition system and \( i \in N \). If there exist two transactions \( t, t' \in T \) such that \( t \rightarrow t' \) in \( SG(H_i(\alpha)) \) then the following happens-before relations, with the appropriate parameters, hold:

1) \( t\text{-node} = t', t\text{-node} = i \): begin_commit_i(t) <_a end_commit_i(t) <_a begin_commit_i(t') <_a end_commit_i(t', m').
2) \( t\text{-node} = i \land t'\text{node} \neq i \): begin_commit_i(t) <_a end_commit_i(t) <_a operation_i(t', WS.ops) <_a receive_commit_i(t', m).
3) \( t\text{-node} \neq i \land t'\text{node} = i \): end_operation_i(t, WS.ops) <_a receive_commit_i(t, m) <_a begin_commit_i(t) <_a end_commit_i(t).
4) \( t\text{-node} \neq i \land t'\text{node} \neq i \): end_operation_i(t, WS.ops) <_a receive_commit_i(t, m) <_a operation_i(t', WS.ops) <_a receive_commit_i(t', m').

If we have two conflicting remote transactions, such that \( t \rightarrow t' \) in \( SG(H_i(\alpha)) \), then the execute_remote_i(t') action that submits \( t' \) to the database must be executed after \( t's
commitment, via the receive_commit(t, m) action. We may follow a similar reasoning if one of them is a local transaction, the execute_remote1 action that submits t′ to the database must be executed after the commitment of t, via the end_commit(t) action.

**Lemma 3:** Let \( \alpha = s_0 \pi_1 s_1 \ldots \pi_x s_x \) be a fair execution of MidO2PL and \( i \subseteq N \). If there exist two committed transactions \( t, t' \subseteq T \) such that \( t \prec t' \) in \( SG(H(\alpha)) \) then: (1) if \( t.\text{node} = j \neq i \) and \( t'.\text{node} = k \neq i \) then \( \forall i \in N \setminus \{k,j\} : \text{execute}_\text{remote}_1(t) \prec \text{receive}_\text{commit}(t, m) \prec \text{execute}_\text{remote}_1(t') \prec \text{receive}_\text{commit}(t', m') \); and, (2) if \( t.\text{node} = i \) and \( t'.\text{node} = i \) then \( \forall i \in N : \text{begin}_\text{commit}(t) \prec \text{end}_\text{commit}(t) \prec \text{execute}_\text{remote}_1(t') \prec \text{end}_\text{operation}_1(t', W'\text{ops}) \prec \text{receive}_\text{commit}(t', m') \).

In the following, we prove that MidO2PL provides 1CS [3].

**Theorem 1:** Let \( \alpha = s_0 \pi_1 s_1 \ldots \pi_x s_x \) be a fair execution of the MidO2PL state transition system. The graph \( \cup_{k \in E} SG(H(\alpha)) \) is acyclic.

**Proof:** (Outline) By contradiction. Assume there exists a cycle in \( \cup_{k \in E} SG(H(\alpha)) \). There are at least two different transactions \( t, t' \subseteq T \) and two different sites \( x, y \subseteq N \), such that those transactions are executed in different order at \( x \) and \( y \). Thus, we consider \( (a) t \prec t' \) in \( SG(H(\alpha)) \) and \( (b) t' \prec t \) in \( SG(H(\alpha)) \); being \( t.\text{node} = i \) and \( t'.\text{node} = j \).

There are four cases under study: (I) \( i = j = x \); (II) \( i = x \land j = y \); (III) \( i = j \land i \neq x \land i \neq y \); and, (IV) \( i \neq j \land i \neq x \land j \neq y \).

In the following, we only show cases (I) and (IV), as cases (II) and (III) are similar to (I), see [9] for more details.

The notation has been simplified since action names are shortened: begin_commit(t) by bc(t); end_commit(t) by ec(t); as each invocation of the execute_remote2 action may execute a set of transactions, \( K \subseteq T \), we denote it by \( er_x(k), \) with \( k \in K \); receive_ready(t, (ready, t, l)), with \( l \in N \), by \( rr_x(t, l) \); end_operation_x(t, op) by eo(t); and, receive_commit(t, m) by rc(t).

**I By Property 2.1** for (a): \( bc(t) \prec ec(t) \prec bc(t') \prec ec(t') \).

**By Property 2.4** for (b): \( eo_y(t) \prec rc_y(t) \prec eo_y(t) \prec rc_y(t) \).

Applying Lemmas 2.2 and 3.1 for \( t \) and \( t' \), \( er_y(t) \prec eo_y(t) \prec eo_y(t) \prec rc_y(t) \).

(For, 1) \( \cup_{k \in E} SG(H(\alpha)) \). Taking into account Lemma 2.1 for \( t \) and Lemma 3.1 for \( t \) and \( t' \); \( bc(t) \prec eo(t) \prec rr_y(t) \prec eo(t) \prec bc(t) \). Therefore, we have that \( er_y(t) \prec eo_y(t) \).

**IV By Property 2.4** for (a): \( eo_y(t) \prec eo_x(t) \prec eo_y(t) \).

Applying Lemmas 2.2 and 3.1 for \( t \) and \( t' \), \( er_x(t) \prec eo(t) \prec eo(t) \).

**By Property 2.4** for (b): \( eo_y(t) \prec eo_y(t) \).

If we apply Lemmas 2.2 and 3.1 for \( t \) and \( t' \), \( er_y(t) \prec eo_y(t) \).

**ACKNOWLEDGMENTS**

This work has been supported by the Spanish Government under research grant TIC2003-09420-C02.

**V. CONCLUSIONS**

In this paper, we present a middleware replication protocol, MidO2PL, providing database replication. The MidO2PL is 1CS, given that the underlying DBMSs feature 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). The underlying DBMS performs its own concurrency control and the replication protocol compliments this task with replica control. MidO2PL 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 DBMSs, 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.

**REFERENCES**


