Proof and Evaluation of a 1CS Middleware Data Replication Protocol Based on O2PL *

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

[1] Instituto Tecnológico de Informática
Camino de Vera s/n
46022 Valencia, Spain
{armendariz, fmunyoz}@iti.upv.es

[2] Universidad Pública de Navarra
Campus de Arrosadía s/n
31006 Pamplona, Spain
{joserra, jr.juarez, mendivil}@unavarra.es

Abstract. Middleware data replication techniques are a way to increase performance and fault tolerance without modifying the internals of a DBMS. However, they introduce overheads that may lead to poor response times. In this paper a modification of the O2PL protocol is introduced. It orders conflicting transactions by using their priority, instead of the total order obtained by an atomic multicast. Priorities are also used to avoid deadlocks. For improving its performance, it does not use the strict 2PC rule as O2PL does. We provide a formal correctness proof of its 1-Copy Serializability (1CS). This protocol has been implemented, and a comparison with other already implemented protocols is also given.

1 Introduction

Database replication is an attractive way for increasing the performance and fault tolerance of applications, but they pay a price for maintaining data consistency. Traditionally, replication has been achieved modifying the Database Management System (DBMS) internals, such as [1,2,3] but this solution is not portable among different DBMS vendors. The alternative approach is to deploy a middleware architecture that creates an intermediate layer that features data consistency, being transparent to the final users. However, one drawback of the middleware approach is that the replication module usually reimplements many features provided by the DBMS. Besides, the database schema has to be extended with standard database features, such as functions, triggers, stored procedures, etc. [4], in order to manage additional metadata that eases replication. This alternative introduces an overhead that penalizes performance but permits to get rid of DBMSs' dependencies. Hence, the goal is to design a system that penalizes performance as less as possible, and that becomes portable to different DBMSs.

The strongest correctness criterion for database replication is 1CS [1] that implies a serial execution over a logical data unit although there are many physical copies. In [4] a middleware architecture is introduced providing 1CS by way of the total order multicast featured by a Group Communication System (GCS) [5,6]. The total order multicast is used so as to determine the order in which transactions are executed on the system. This is an interesting approach since transactions do not have to wait for applying the

* Work supported by the Spanish MEC grants TIC2003-09420-C02 and TIN2006-14738-C02.

updates at the rest of nodes in order to commit, as the 2PC rule states, increasing its performance. However, to rely on these strong GCS primitives is a high price to pay in environments where conflicts are rare, due to the latency and extra message rounds introduced by the total order multicast [6,3].

O2PL [2] was one of the first replication protocols that followed the Read One Write All Available (ROWAA) approach [1]. Transactions are firstly executed at their closest node (or *master node*, hereafter) and $updates$ are propagated to the rest of nodes without any ordering assumption. Updates reception at the rest of nodes starts a remote transaction requesting a *copy-lock* for applying the updates. This lock behaves like a write lock does, but it is used to prevent deadlocks with local transactions. Despite of this, a *snoop process* is still needed to detect and resolve distributed deadlocks. Once updates are applied at a replica, it sends a message to the master node saying it is $ready$ to commit. Meanwhile, the master node collects all $ready$ messages coming from the rest of replicas. At the time when all nodes have answered, the master node commits and multicasts a $commit$ message to all replicas. Therefore, O2PL is a 2PC protocol since it waits for the updates application at all available replicas before committing a transaction.

Here, we propose an evolution of O2PL [2] adapted to our MADIS architecture [7] called Enhanced Replication Protocol (ERP). Using MADIS, the concurrency and replica control may be split into two levels: the underlying DBMS at each node providing serializable isolation level whilst the middleware layer is in charge of data consistency. Hence, no specific DBMS feature has to be re-implemented at the middleware layer. ERP only needs a reliable multicast as the communication mechanism among replicas [5]. We have changed the 2PC philosophy of O2PL: a transaction does not wait for applying the updates at the rest of nodes in order to commit. Hence, all the advantages of total order based replication are obtained without their associated communication costs. Besides, all non-conflicting transactions do not need to be totally ordered as such protocols do. This is obtained using a dynamic priority function that guarantees the atomic commitment of transactions. The priority associated to a transaction is derived from a unique weight associated to the transaction, and the state of the transaction, which varies throughout its lifetime. This imposes the order on which conflicting transactions are applied at all nodes. Therefore, the success of a transaction that broadcasts its updates can be guessed before its submission to the underlying DBMS. Moreover, this dynamic priority function serves as a deadlock prevention function.

The rest of this paper is organized as follows: The system model is introduced in Section 2. The formalization of our protocol is presented in Section 3. Section 4 shows its correctness proof. ERP has been implemented and some experimental results as well as its comparison with other replication protocols are shown in Section 5. A brief outline of related works is given in Section 6. Finally, conclusions end the paper.

2 System Model and Definitions

For the sake of the explanation of ERP and its correctness proof, an abstraction of MADIS in a failure free environment is presented in this Section. Details about failures and the recovery process are given in [8] and the interested reader should refer to that complementary work. The system considered in this paper (Figure 1) is composed by

N nodes which communicate among them using reliable multicast [5]. We assume a fully replicated system. An application submits transactions for its execution over its local DBMS via the middleware. The replication protocol coordinates the execution of transactions among different nodes to ensure 1CS [1]. 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 DBMS operations.

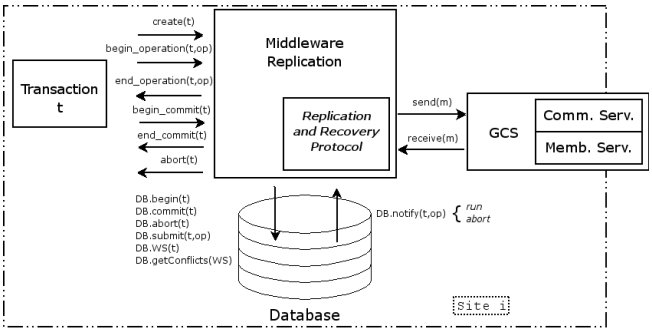

Fig. 1. Main components of the system

Database. It is assumed a DBMS ensuring ACID properties of transactions and satisfying the serializable transaction isolation level. After a SQL statement submission (denoted op) in the context of a transaction t, the $DB.notify(t, op)$ informs about the successful completion of an operation (run); or, its rollback ($abort$) due to DBMS internals. It is assumed that a transaction will only be unilaterally aborted if it is involved in a local deadlock. 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 be built by standard database functions [4]. $DB.WS(t)$ retrieves the set of objects written by t and the respective SQL update statements. In the same way, the set of conflicting transactions between a write set and current active transactions is given by $getConflicts(WS(t)) = \{t' \in T\colon (WS(t') \cup RS(t')) \cap WS(t) \neq \emptyset\}$, where T is the set of system active transactions.

Transactions. Each transaction t has an identifier including the information about its *transaction master node* ($node(t)$), in order to know if it is a local or a remote transaction. It also contains information so as to obtain the weight associated to it ($weight(t)$). This value is based on its own information, such as: number of restarts, size of readset, size of writeset, node identifier and so on. All these parameters are defined by ERP at the system startup with different influence in the final weight, although ensuring its uniqueness. A transaction t created at node i ($node(t) = i$) is locally executed and starts the interaction with the rest of nodes when the application wishes to commit the transaction with the execution of *remote transactions*. Finally, ERP reports on the transaction fate to the application. For simplicity, we do not consider an application abort.

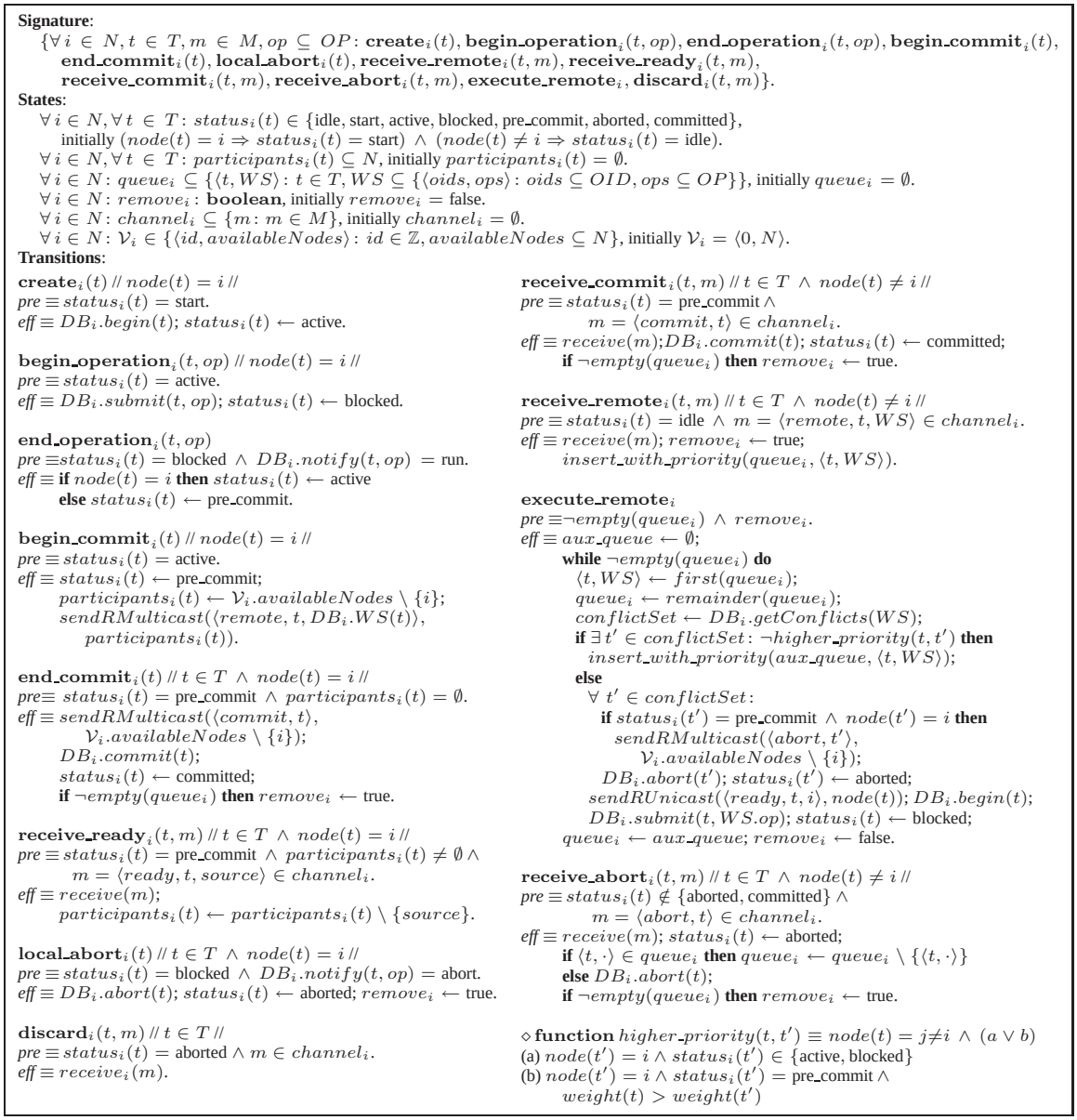

Fig. 2. State transition system for the Enhanced Replication Protocol

3 ERP Description

In this Section we use a state transition system [9] for describing ERP (introduced in Figure 2). It includes a set of state variables and actions, each one of them subscripted

with the node identifier where they are considered. State variables include their domains and an initial value. Each action in the state transition system has an enabling 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. The effects of an action (*eff* in Figure 2) is a sequential program that atomically modifies the state variables; hence, new actions may become enabled while others disabled. Weak fairness is assumed for actions, i.e. if an action is continuously enabled then it will be eventually executed. Although the state transition system seems a static structure, it defines the algorithm's execution flow. We explain such algorithm on the sequel.

A transaction t starts the execution at its master node since $status_i(t) =$ start as $node(t) = i$. It invokes the $create_i(t)$ action followed by a sequence of pairs of the form $begin\_operation_i(t, op)$ and $end\_operation_i(t, op)$. The $begin\_operation_i(t, op)$ invocation submits the SQL statement to the database ($DB_i.submit(t, op)$) and $status_i(t) =$ blocked. The transaction may be aborted due to a local deadlock resolution by the DBMS replica, as long as $status_i(t) =$ blocked, or an ERP decision (that will be discussed afterwards). The $end\_operation_i(t, op)$ action will be eventually invoked after the operation is completed in the database and the local transaction may submit a new statement. Once the transaction is done it requests its commitment, by means of the $begin\_commit_i(t)$ action, as $status_i(t) =$ active. This action initializes the variable $participants_i(t)$ to the set of reachable nodes excluding itself. ERP starts to work. Until now only the underlying DBMS was managing the concurrency control. ERP collects the writeset of the transaction and multicasts a $remote$ message to the rest of available nodes and changes its $status_i(t)$ to pre_commit. This emphasizes that it is a local transaction that has propagated its updates to the rest of available nodes.

ERP includes a $queue$ so that each time a transaction t is delivered at a node via a $remote$ message (the $receive\_remote_j(t, \langle remote, t, WS\rangle)$ action, with $j \neq i \in N$), it is firstly enqueued (arranged by $weight(t)$) and the $remove_j$ state variable is set to true. This last variable governs the time when the $queue_j$ has to be inspected, i.e. when the $execute\_remote_j$ action is called. The straightforward points of checking $queue_j$ are: when database resources are released, such as a transaction commit or rollback, and when a new $remote$ message arrives, see Figure 2. The execution of $execute\_remote_j$ disables $remove_j$ and iterates through all the transactions contained in $queue_j$ in order to check if any of these transactions has more priority than any other conflicting transaction currently submitted to the database, this is done to prevent distributed deadlock cycles between nodes. If so, the delivered transaction will send the $ready$ message to the master node and all updates will be executed in the context of another local transaction, called *remote transaction*. Before executing this remote transaction all local conflicting transactions must be firstly aborted, as this transaction has sent the $ready$ message, it may not be involved in any local deadlock. Otherwise, when the enqueued transaction has not enough priority, it will remain in $queue_j$ until it reaches the highest priority or its master node decides to abort it. One can note that several transactions (that do not conflict and have enough priority) can be submitted in one execution of $execute\_remote_j$. In either case, $ready$ messages are collected at the master node, via the $receive\_ready_i(t, \langle ready, t, j\rangle)$ action. Once all of them have been received ($participants_i(t) = \emptyset$), the $end\_commit_i(t)$ is enabled and the master node

commits and multicasts a $commit$ message. The $receive\_commit_j(t, \langle commit, t\rangle)$ action will not be invoked in the rest of replicas until the updates have not been done ($status_j(t) =$ pre_commit) as it can be seen in the enabling condition of this action.

The priority between transactions is defined by the $higher\_priority(t, t')$ function where t and t' play the role of an enqueued and an executing transaction respectively. Recall that ERP does not abort a *remote transaction* already submitted to the database unless its master node decides to do so. Hence, an enqueued transaction will have more priority than those local transactions, $t'.node = i$, still executing SQL statements ($status_i(t') \in \{$active, blocked$\}$) or local transactions in pre_commit whose $weight(t')$ is lower than $weight(t)$. Otherwise, an enqueued transaction will remain enqueued. Therefore, the transaction master node exclusively decides the outcome of the transaction.

Hence, ERP has modified the 2PC rule of O2PL by the use of this function. It allows a *remote transaction* to send the $ready$ message to the master node before its completion in the database. Thus, the response time $\theta_{r_{O2PL}}(t)$ of a transaction t ($node(t) = i$) with O2PL in the middleware architecture is determined by the sum of the following times: the transaction processing at the master node, $\theta_{DB_i}(t)$; multicasting the $remote$ message to the rest of nodes, $\theta_{MC}(t)$; transaction updates processing at the rest of available nodes $\theta_{DB_j}(t)$, with $j \in N \setminus \{i\}$; and, finally, each remote node sending the $ready$ message back to the master node, $\theta_{UC_j}(t)$. Therefore, we have $\theta_{r_{O2PL}}(t) \approx \theta_{DB_i}(t) + \theta_{comm}(t) + \max_j(\theta_{DB_j}(t))$, with $\theta_{comm}(t)$ grouping all communication costs. This response time is a consequence of the 2PC origin of the O2PL, and it is limited by the slowest remote transaction execution, since it waits for applying the updates at all nodes before committing. Thus, if we send the $ready$ message back once the transaction has overcome the deadlock prevention function and before it has been submitted to the database, we reduce the transaction response time to the following: $\theta_{r_{ERP}}(t) \approx \theta_{DB_i}(t) + \theta_{comm}(t)$. This time is decreased because it does not need to wait for the execution of the remote transactions, therefore we get rid of $\max_j(\theta_{DB_j}(t))$.

A *local transaction* is the single kind of transaction that may be aborted by the DBMS while it is executing SQL statements. Hence, $local\_abort_i(t, op)$ may be invoked if the DBMS may not execute the op statement contained in the $begin\_operation_i(t, op)$ due to an internal deadlock resolution; thus, $DB_i.notify(t, op) =$ abort. It is important to note that ERP aborts all local conflicting transactions before the execution of a *remote transaction*; hence, it may never be involved in a local deadlock at the time it is submitted to the database. An aborted local transaction may be in the pre_commit state, in that case it will multicast an $abort$ message that will enable the $receive\_abort_j(t, \langle abort, t\rangle)$ action. In order to simplify the algorithm's presentation, we assume that the writeset of a remote transaction is atomically executed in order to avoid its concurrent execution with local transaction operations that may lead to an abortion of the former. Therefore a remote transaction may only be aborted by its master node.

4 Correctness Proof

This Section contains the most important proofs (atomicity and 1CS) of ERP in a failure free environment. For a more detailed description of the correctness proof the reader is referred to [8]. We continue using the notation and definitions of a state transition sys-

tem [9]. For each ERP's 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. An execution is finite if it always finishes in a state, or infinite if not. 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 ERP action requires weak fairness.

We firstly start proving that ERP is deadlock free. Local deadlocks are handled by the $local\_abort_i(t, op)$ action. Remote transactions are only involved in distributed deadlocks. The use of priorities avoids distributed deadlocks in the system. A transaction t_i ($node(t_i) = i$) *waits for* another transaction t_j ($node(t_j) = j$) if and only if t_i is remote at node j such that $weight(t_i) < weight(t_j)$, thus $t_i \in queue_j$. Assume there exist a cycle in the system ($t_0 \rightarrow t_1 \rightarrow \cdots \rightarrow t_{N-1} \rightarrow t_0$). In such a case $t_{i \bmod N}$ waits for $t_{(i+1) \bmod N}$ for i: $0 \ldots N-1$. Hence, we have that $weight(t_0) < weight(t_1) \cdots < weight(t_{N-2}) < weight(t_{N-1})$ and, if we continue with the cycle, $weight(t_{N-1}) < weight(t_0)$ which is a contradiction. Thus, the system is deadlock free.

The next Property formalizes the $status$ transition for a given transaction $t \in T$ in ERP. It points out 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 $k' > k'' > k$, as it can be deduced from Figure 2.

Property 1 *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be an arbitrary execution of the ERP 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 node $j \in N$, 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:*
   $start \cdot active \cdot (blocked \cdot active)^* \cdot pre\_commit \cdot committed$
   $start \cdot active \cdot (blocked \cdot active)^* \cdot pre\_commit \cdot aborted$
   $start \cdot active \cdot (blocked \cdot active)^* \cdot aborted$
   $start \cdot (active \cdot blocked)^+ \cdot aborted$
2. *If $node(t) \neq j$ then β is a prefix of the regular expression:*
   $idle$ $\qquad$ $idle \cdot blocked \cdot pre\_commit \cdot committed$
   $idle \cdot blocked \cdot aborted$ $\qquad$ $idle \cdot blocked \cdot pre\_commit \cdot aborted$
   $idle \cdot aborted$

This Property is simply proved by induction over the length of α following the preconditions and effects of the ERP actions. A $status$ transition for a transaction t 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 inspection of Figure 2.

The following Property is needed to prove the atomicity of a transaction; that is, the transaction is either committed, or aborted, at all available nodes. It states the invariant properties of ERP. If a transaction t with $node(t) = i$ is committed at $j \neq i$, it is

because it was already committed at its master node. A remote transaction currently being executed at its DB_j module ($status_j(t) =$ blocked) may only change its $status$ if its execution is completed or by an $abort$ message coming from its master node. In other words, it will never be aborted by the DB_j module. In the same way, a remote transaction in the pre_commit state may only change its $status$ if it receives a $commit$ or an $abort$ message from its master node. Finally, if a transaction is $committed$ at its master node then at the rest of available nodes it will be either $committed$ (it has already received the $commit$ message), pre_commit (it is waiting to receive the $commit$ message) or $blocked$ (it is still applying the updates at that node).

Property 2 *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be an arbitrary execution of the ERP 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 $j \in N \setminus \{i\}$ then $\forall\, z''\colon z' < z'' \leq z\colon \pi_{z''} \notin \{receive\_abort_j(t, \langle abort, t\rangle), end\_operation_j(t, WS.op)\}$.*
3. *If $\exists\, z' < z\colon s_{z'}.status_j(t) = s_z.status_j(t) =$ pre_commit for any $j \in N \setminus \{i\}$ 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 $\forall\, j \in N\colon s_z.status_j(t) \in \{blocked, pre\_commit, committed\}$.*

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 ERP 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 with $j = i$) $s_z.status_i(t) =$ committed. By Property 2.4, $\forall\, j \in N \setminus \{i\}\colon s_z.status_j(t) \in \{$blocked, pre_commit, 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, due to $execute\_remote_j$ for t). By weak fairness of action execution, the $end\_operation_j(t, WS.op)$ action will be eventually invoked and $s_z.status_j(t) =$ 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 $receive\_commit_j(t, \langle commit, t\rangle)$ action is enabled. By weak fairness assumption, the action will be eventually executed, thus $\exists\, z' > z\colon \pi_{z'} = receive\_commit_j(t, \langle commit, t\rangle)$. Thus, by its effects, $s_{z'}.status_j(t) =$ committed.

In a similar way, when a transaction is aborted, it is aborted at all nodes, as stated in the following Lemma. The proof is very simple, by inspection of the ERP actions.

Lemma 2 *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the ERP 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 $j \in N$.*

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 $\pi \prec_\alpha \pi'$. For example, assuming t is a committed transaction with $node(t) = i \neq j$, the following *happens-before* relation $begin\_commit_i(t) \prec_\alpha receive\_remote_j(t, \langle remote, t, WS\rangle)$ is held, see Figure 2. 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 $receive\_remote_j(t, \langle remote, t, WS\rangle)$ action. As $status_j(t) =$ idle, and by weak fairness of actions, it will be eventually executed. However, this fact is delegated to the $execute\_remote_j$ action. The following Lemma indicates that a transaction is $committed$ if it has received every $ready$ message from all available remote replicas.

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

The following Lemma emphasizes the *happens-before* relation for remote transactions. It is based on Property 1.2 which establishes the relation between $status$ transitions for remote transactions to their respective algorithm actions. This will serve in order to set up the relation for a transaction t, with $node(t) = i \neq j$, between the $execute\_remote_j$, that submits t to the DB_j module, and the pair $end\_operation_j(t, WS.op)$ and $receive\_commit_j(t, \langle commit, t\rangle)$ actions. This is needed due to the fact that with Lemma 3 there is no point where this causal relation 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 ERP automaton and $t \in T$ be a committed transaction, $node(t) = i$, then the following happens-before relations hold: $\forall j \in N \setminus \{i\}$: $receive\_remote_j(t, \langle remote, t, WS\rangle) \prec_\alpha execute\_remote_j(t) \prec_\alpha end\_operation_j(t, WS.op) \prec_\alpha receive\_commit_j(t, \langle commit, t\rangle)$.*

In order to define the correctness of our replication protocol we have to study the global history (H) of committed transactions ($C(H)$) [1]. We may easily adapt this concept to ERP. 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 node i. For a given α execution of ERP, $H_i(\alpha)$ plays a similar role as the local history at node i, H_i, as introduced in [1] 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 [1]. An arc and a path in $SG(H_i(\alpha))$ are denoted as $t \rightarrow t'$ and $t \stackrel{*}{\rightarrow} t'$ respectively. Recall that our local DBMS produces serializable histories; 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 nodes its serialization graph, $\cup_k SG(H_k(\alpha))$, must be acyclic so that conflicting transactions are equally ordered in all local histories. Recall that the correctness criterion is 1CS.

Before showing the correctness proof, we need an additional Property relating the transaction isolation level of the underlying DB modules to the automaton execution

event ordering. The following Property and Corollary establish a property about local executions of committed transactions and their respective ERP actions.

Property 3 *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be an arbitrary execution of the ERP 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.*

Corollary 1 *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the ERP 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 relation, with the appropriate parameters, holds:*

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

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 $\wedge$ $s_{z_4}.status_i(t') =$ 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 conflicting transactions $t, t' \in T$, with $node(t) \neq i$ and $node(t') \neq i$, such that $t \rightarrow t'$ in $SG(H_i(\alpha))$, then the $execute\_remote_i$ action that submits t' to the database must be executed after committing t, via the $receive\_commit_i(t, \langle commit, t\rangle)$ action. The next Lemma states how the *happens-before* relation 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 ERP 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 relation hold: $\forall\, i \in N \setminus \{k, j\} : execute\_remote_i(t) \prec_\alpha receive\_commit_i(t, \langle commit, t\rangle) \prec_\alpha execute\_remote_i(t') \prec_\alpha receive\_commit_i(t', \langle commit, t'\rangle)$.*

The same may be applied to two conflicting transactions $t, t' \in T$ with $node(t) = i$ and $node(t') \neq i$, such that $t \rightarrow t'$ in $SG(H_i(\alpha))$. The $execute\_remote_i$ action that submits t' to the database must be executed after the commitment of t by the $end\_commit_i(t)$ action. The next Lemma states how the *happens-before* relation 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 ERP 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 relation hold: $\forall\, i \in N$: $begin\_commit_i(t) \prec_\alpha end\_commit_i(t) \prec_\alpha execute\_remote_i(t') \prec_\alpha end\_operation_i(t', WS'.op) \prec_\alpha receive\_commit_i(t', \langle commit, t'\rangle)$.*

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

Theorem 1. *Let $\alpha = s_0\pi_1 s_1 \ldots \pi_z s_z \ldots$ be a fair execution of the ERP 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 nodes $x, y \in N$, $x \neq y$, 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 $execute\_remote_x$ 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 $rc_x(t)$.

(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).

(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).

(III). As x and y are different nodes from the transaction master node, only one of them will be executed in the same order as in the master node. If we consider the different one with the master node we will be under assumptions considered in CASE (I).

(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).

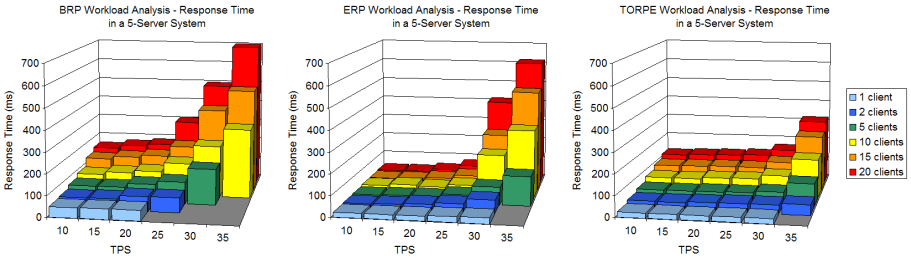

Fig. 3. Protocols Response Time Evaluation

5 Experimental results

We have implemented the ERP protocol in the MADIS architecture, in order to test the performance improvement of the ERP protocol against other approaches, such as BRP (based on a termination voting similar to a 2PC) and TORPE (based on total order delivery of write-sets, removing the voting phase). For all the experiments, we used a cluster of 5 workstations (Pentium IV 2.8GHz, 1GB main memory, 80GB IDE disk) connected by a full duplex Fast Ethernet network. We have implemented an *ad hoc* reliable multicast using TCP for BRP and ERP, whilst Spread 3.17 was in charge of the total order multicast needed by TORPE. PostgreSQL 7.4 was used as the underlying DBMS. The database consists of 30 tables each containing 1000 tuples. Each table has the same schema: two integers, one being the primary key. Transactions consist of a number of update operations each one modifying a given tuple randomly chosen from a table of the database. The interarrival time between the submission of two consecutive transactions is uniformly distributed. The workload is denoted by the number of transactions submitted per second (TPS). All tests were run until 2000 transactions were executed.

These experiments test how the three replication protocols cope with increasing number of users and workloads. Workload was increased steadily from 10 to 35 TPS. For each workload, several tests were executed varying the number of clients from 1 to 20 in the whole system. Figure 3 shows the response time obtained in this experiment for the three presented protocols. As a general rule, the maximum throughput is limited since a client can only submit one transaction at a time, and hence, the submission rate per client is limited by the response time. With one client TORPE and ERP's response times are below 25 and 28 ms respectively, but BRP ones are close to 50 ms and it is not possible to achieve the desired throughput. The abortion rates for this experiment are low, between 0% and 2%, due to the random nature of the generated numbers.

Results revealed that BRP presents the worst behavior of the presented protocols, due to its 2PC transaction termination. In ERP, the remote nodes send the $ready$ message once conflicts and priority rules are checked before executing the updates. In TORPE, the master node does not wait for any response from remote nodes, only waits for the delivery of the messages in total order. As seen in Figure 3, ERP response times keep between 20 and 70 ms for a given number of clients with workloads up to 25 TPS. TORPE response times, working with intermediate loads, remain between 20 ms with 1 client and 125 ms with 20 clients.

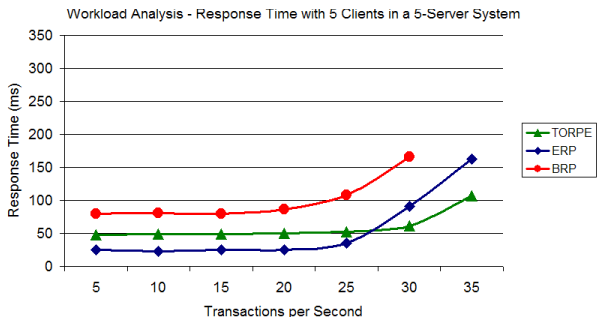

Fig. 4. Response time of the replication protocols varying the submission rate

Increasing the workload and the multiprogramming level only results in higher response times, due to the administration overhead (process switches, communication to/from the client) and contention at specific resources. As shown in Figure 3, BRP and ERP are more affected by this overhead than TORPE is. This happens since we have implemented the ad hoc reliable multicast inside the ERP and BRP protocols, hence they support the load of the protocol itself and the communication between nodes. On the other hand, TORPE uses an underlying GCS to manage communication among nodes that runs independently of the protocol. Thus, TORPE response times do not increase so much as in the other protocols at high workloads. That is the reason why there is a crossing point between ERP and TORPE in Figure 4, where only the configuration with 5 clients is analyzed.

6 Related Works

There are a lot of replication protocols defined in the literature, modifying the DBMS core [1,2,3]. However, we will highlight those developed for middleware solutions. For instance, with optimistic atomic broadcast, as described in [3], messages are delivered as they are received, making possible a fast remote writeset application, although waiting for the final ordered message delivery in order to commit the transaction. Thus, only those remote transactions whose writeset did not follow the total order are rolled back, reapplying them in the correct order. This idea has been applied in [10] for WAN networks in the GlobData project. Respectively, a more aggressive version of the optimistic atomic broadcast [3] in a middleware architecture is presented in [4]. Database stored procedures are executed as transactions defining a conflict class. Transactions issued by users are delivered using the optimistic atomic broadcast to all nodes but the outcome of transactions is only decided at the master node of the respective conflict class. Hence, remote nodes do not even have to wait for the definitive total order to execute a transaction. It additionally provides good scalability results. The BRP and ERP may accept any SQL statement, hence they are more flexible; however, ours present a higher overhead since they propagate the SQL statements and we do not try to balance the workload.

7 Conclusions

In this paper, we present a middleware database replication protocol, called ERP. It is an adaptation of O2PL where the 2PC rule has been modified in order to improve O2PL performance. We have proved that ERP is 1CS –and this has been, up to our knowledge, the first prove of this kind for any O2PL-based replication protocol–, given that the underlying DBMSs feature a serializable transaction isolation level. This replication protocol has the advantage that no specific DBMS tasks have to be re-implemented. The underlying DBMS performs its own concurrency control and the replication protocol complements this task with replica control.

ERP is an eager update everywhere replication protocol. All transaction operations are firstly performed on its master node, and then all updates are grouped and sent to the rest of nodes using a reliable multicast. However, our algorithm is liable to suffer distributed deadlock. Hence, we have defined a deadlock prevention schema that orders transactions, which is based on the transaction state and associated weight. Besides, as it totally orders transactions, ERP will know if a transaction may proceed or not before its submission to the DBMS. This allows us to get rid of the of the strict 2PC rule.

ERP has been implemented in MADIS, as well as an adaptation of O2PL (BRP) and a total order based (TORPE) protocols. They have been compared against each other with a specific benchmark. Results highlight the benefits of ERP when compared to BRP. However, TORPE shows better performance in heavy loaded environments, due to our reliable multicast implementation, whilst ERP is the best in the rest of tests, due to the overhead introduced by Spread to provide total order. Thus, ERP may be considered as an intermediate solution between 2PC and total order based protocols.

References

1. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison Wesley (1987)
2. Carey, M.J., Livny, M.: Conflict detection tradeoffs for replicated data. ACM Trans. Database Syst. **16** (1991) 703–746
3. Kemme, B., Pedone, F., Alonso, G., Schiper, A., Wiesmann, M.: Using optimistic atomic broadcast in transaction processing systems. Trans. Knowl. Data Eng. **15** (2003) 1018–1032
4. Jiménez-Peris, R., Patiño-Martínez, M., Kemme, B., Alonso, G.: Improving the scalability of fault-tolerant database clusters. In: ICDCS. (2002) 477–484
5. Chockler, G., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Comput. Surv. **33** (2001) 427–469
6. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Comput. Surv. **36** (2004) 372–421
7. Irún, L., Decker, H., de Juan, R., Castro, F., Armendáriz, J.E., Muñoz, F.D.: MADIS: A slim middleware for database replication. Springer LNCS **3648** (2005) 349–359
8. Armendáriz, J.E.: Design and Implementation of Database Replication Protocols in the MADIS Architecture. PhD thesis, Universidad Pública de Navarra, Pamplona, Spain (2006)
9. Shankar, A.U.: An introduction to assertional reasoning for concurrent systems. ACM Comput. Surv. **25** (1993) 225–262
10. Rodrigues, L., Miranda, H., Almeida, R., Martins, J., Vicente, P.: The GlobData fault-tolerant replicated distributed object database. Springer LNCS **2510** (2002) 426–433