A Lock Based Algorithm for Concurrency Control and Recovery in a Middleware Replication Software Architecture

J.E. Armendáriz and J.R. González de Mendívil
Dpto. de Matemática e Informática
Universidad Pública de Navarra
Campus Arrosadía s/n, 31006 Pamplona, Spain
Email: {enrique.armendariz, mendivil}@unavarra.es

F.D. Muñoz-Escoí
Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, Spain
Email: fmunyoz@iti.upv.es

***Abstract*— Data replication among different sites is viewed as a way to increase application performance and its data availability. In this paper, we propose an algorithm design for concurrency control and recovery in a middleware architecture called COPLA (Common Object Programmer Library Access). This architecture provides persistent object state replication. The algorithm is based on locks, it is an adaptation of the Optimistic Two Phase Locking (O2PL) protocol to this architecture. The recovery process of this algorithm allows applications to continue (or start) executing transactions at all nodes, even in the node being recovered.**

I. Introduction

Replication is mainly used to store some (or all) data items redundantly at multiple sites. Its goal is to increase system reliability and application performance. Databases are widely used by enterprises as the preferred storage media for their data and their management. Thus, storing data at multiple sites allows the system to continue working even though some sites may have failed. Besides, it also increases its throughput by means of performing local reads where in a non-replicated architecture implied the start of a new remote transaction. A replicated middleware architecture providing object state persistence may be viewed as a replicated object database providing persistence.

All these advantages are not for free, replication has some problems, such as data consistency and fault-tolerance. The system must introduce an additional overhead for maintaining replicated data consistency. Applications must introduce additional software in order to access distributed resources, thus increasing application development complexity.

Data consistency is granted by a particular consistency protocol. These protocols have been widely discussed in the literature [1]. They vary from optimistic, where no (or low) data contention occurs, and pessimistic. And also they may be eager, if update propagation takes place at commit time (to all alive nodes or the primary copy), or lazy, if it happens on demand of a node requesting data, using pull and push strategies [2]. All these combinations provide a set of consistency protocols that features a set of advantages and drawbacks that greatly depends on the kind of application used. Besides data access is performed concurrently among several users, usually in a transactional manner, thus this consistency protocols must guarantee the transaction isolation rules proposed by ANSI [3].

One of the key issues of replicated architectures, as it has been previously highlighted, is data availability. The system must continue accomplishing its tasks, even though a node fails. Group membership monitors [4] are used to detect node failures or network partitions. Steps to be done when a node fails vary from system reconfiguration after the failure, passing by partition merge to bring data "*up-to-date*" after it recovers from a crash by a previously alive node. We do not consider our data replication proposal complete unless we do not provide a recovery protocol for this concurrency protocol. Reconfiguration needed when the number of sites increases is a far more complex task than that necessary when the number of nodes decreases. In particular, before a node can execute transactions, an "*up-to-date*" node has to provide the current data state to the joining node [5].

Our collaboration experience in the development of the COPLA (Common Object Programmer Library Access) architecture with enterprises [6] has shown that they are very interested in a serializable transactional behavior that guarantees an eager replication to all nodes, as well as in the development of recovery techniques so no back-up copies are necessary and alive nodes may continue working independently of a node failure. COPLA consists of a middleware architecture providing transparency for persistent object state replication while guarantees several consistency levels [7]: $transactional$ (serializable), $checkout$ (similar to the concurrent version system guarantees) and $plain$ (read-only). Several optimistic consistency protocols (eager and lazy), along with their respective recovery protocols implemented in this architecture, allow applications to switch to the one that best suits to them and to maintain data coherence [7]–[9]. Object state is persistently stored in a Relational Database Management System (RDBMS), more precisely, PostgreSQL [10]. The necessity of storing objects in an RDBMS permits the coexistence of old information with new data. This was a requirement of our industrial partners [6] in the development of COPLA. This

kind of storage leads to an additional problem. There is an imbalance between the entity relational model and the object oriented paradigm that must be solved. We have defined a translation pattern for a schema (or application), defined in COPLA, to its equivalent relational model, performed by its respective compiler [11], [12].

The work presented in this paper makes several contributions. First, it proposes an adaptation of the Optimistic Two Phase Locking (O2PL) consistency protocol, proposed by Carey et al. [13], to the COPLA architecture, granting a $transactional$ consistency level. The basic idea underlying O2PL is, thus, to set locks locally, where doing so is cheap, while taking a more optimistic, less message-intensive approach across node boundaries. Since O2PL is liable to suffer global deadlocks, this first draft includes a deadlock prevention technique. Second, we introduce a recovery protocol for this concurrency control algorithm. This algorithm will allow sessions at all nodes (even the recovering one) to continue working as long as they do not interfere with objects currently being recovered, thus benefitting system performance. Up to our knowledge, this is something missed in related works that introduced O2PL usage. This recovery algorithm follows a lock policy very similar to O2PL since objects being recovered have a special lock set on it. This special lock controls accesses to these recovering objects. The key idea behind this protocol is to transfer the latest object state of an object to the recovering nodes [11].

The rest of the paper is organized as follows: Section II gives a brief review of works dealing with O2PL and recovery; Section III is devoted to give an outline of COPLA architecture; the concurrency control and recovery algorithm description is depicted in Section IV; and, finally, we summarize the main insights from the paper and the future research lines derived from this work.

II. Related Works

We will base our algorithm design on the Optimistic Two Phase Locking (O2PL) specified in [13]. It focuses on data contention for distributed database systems. Several algorithms are introduced and studied, based on timestamp ordering, optimistic approaches and several for strict Two Phase Locking (2PL), as introduced by Bernstein et al. in [14]. These algorithms are compared varying the number of replicas, data contention level and communication costs. Results obtained highlight the benefits of O2PL against all of them, under all circumstances.

O2PL has been implemented in several architectures as in MIRROR [15] where O2PL is enhanced with a novel-state-based real-time conflict. The O2PL algorithm has been extended to comprise object based-locking as in [16], because object supports more abstract operations than the low-level read and write operations. O2PL has been extended to the mobile environment as it is depicted in [17]. The algorithm introduced there is called O2PL-MT (O2PL for Mobile Transactions) which allows a read unlock for an item to be executed at any copy site of the item.

Similar approaches to ours have been proposed in the literature. Kemme et al. [18] have developed eager replication protocols oriented to the internal structure of the RDBMS. There is an initial local reading phase where all operations are performed locally. All write operations are performed once the latter phase is done, they are bundled and sent using a total order multicast. Thus, deadlocks are avoided and serialization is granted thanks to the total order provided by the group communication. The main differences with the approach followed in our work are the following: we develop it for a middleware architecture, we do not rely on total order multicast to avoid deadlocks, we use a deadlock prevention technique based on information associated to a session, such as restart ratio, transaction execution time, the session identifier, the number of updates done, etc. Usage of total order multicast primitives is costly in communication terms, we do not need these strong group communication primitives. Consistency protocols developed in COPLA by [8], [9] share some characteristics with those described in [18].

Another consistency protocol that has been implemented in COPLA is the Full Object Broadcast (FOB) [7], which uses the object ownership concept, i.e. the node where the object was created. As a brief outline, the idea is to perform object updates locally at the node where the transaction was originated, at transaction commit time, the site owning the object is asked whether it can grant the request to update the object, if this grant is denied the transaction is automatically aborted, otherwise is granted. Therefore, deadlock is avoided because only one transaction is allowed to proceed. We do not rely on object ownership in order to grant an access to an object, we base our update policy in acquiring locks at all available sites.

Our main differences with protocols currently developed for COPLA are: first, it utilizes simple group communication primitives such as reliable FIFO multicast and unicast; second, it provides a deadlock prevention technique which is flexible enough in order to be based on inherent transaction characteristics [19]; and, third, we do not need to persistently store any metadata associated to our concurrency protocol.

Jiménez-Peris et al. [20] propose a concurrency control and a recovery algorithm based on logs and partitions which avoids deadlocks in the same way as [18] does, based on group communication guarantees, i.e. total order. The main difference with our approach is that we do not propose neither stored procedures nor partitions that may lead to a flexibility application loss.

Up to our knowledge, we are not aware of recovery protocols developed for O2PL. Nevertheless, we have studied solutions proposed by Kemme et al. in [5], Jiménez-Peris et al. in [20], COPLA existing solutions [7]–[9] and Bernstein et al. solutions to the recovery problem in distributed databases [21].

III. COPLA Software Architecture

COPLA architecture is split into three different layers as it is shown in Figure 1. These layers are implemented in Java and may reside in different machines since they use

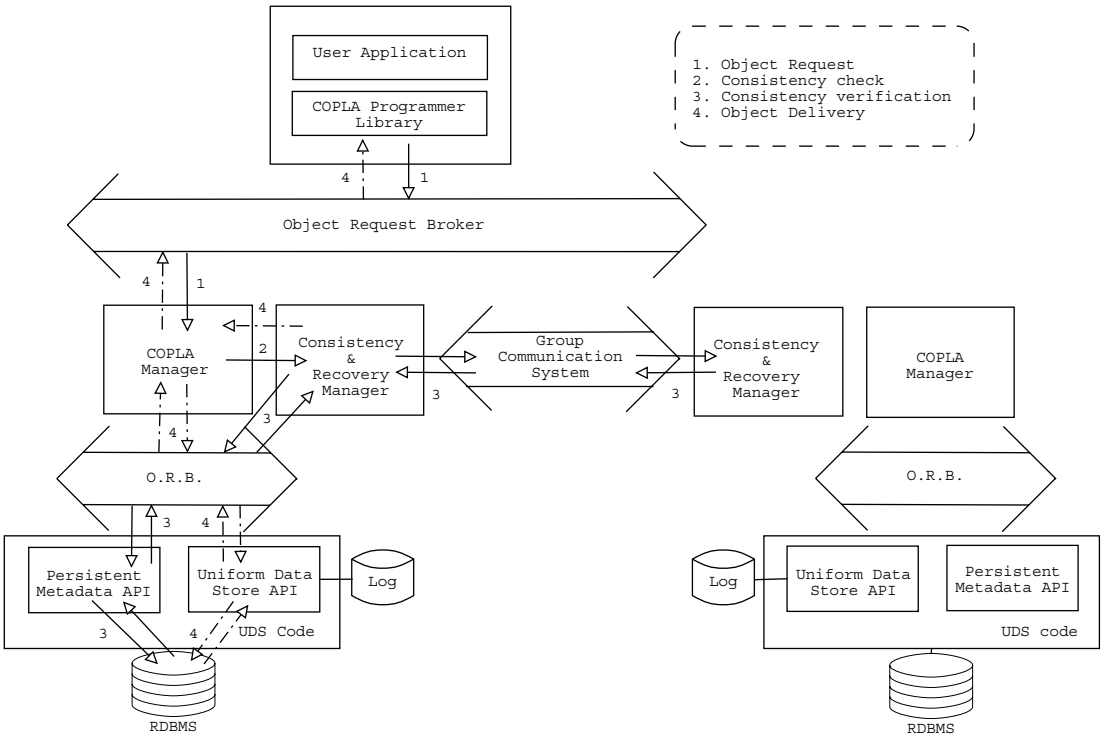

Fig. 1. COPLA Architecture.

an ORB to interact among them. Following a top to bottom approach the first layer is the library. It provides an object oriented view following the ODMG standard [12], [22] to the application programmers. Classes are defined inside a schema, which defines an object repository. Objects can be concurrently accessed in the context of a distributed transaction [7]. Application programmers use a subset of the Object Query Language [12] to obtain references to distributed objects. Once these references are obtained, the application may modify data or obtain new objects through relationships. When the application has finished, it requests for committing the current session.

The COPLA manager layer is the key component of the architecture, it manages an object cache and the consistency among different replicas, located at different sites or nodes. Therefore, it needs a specific protocol (or consistency manager, as shown in Figure 1) that defines some specific rules so as to update replicas following the same order. It has to determine the existence of conflicts between different nodes trying to concurrently access or modify the same object. Several consistency protocols have been implemented in COPLA [7]–[9]. The best protocol's choice depends on the network topology and the application's workload. All these consistency protocols implement a common protocol interface. This allows COPLA to be configured according to the environment characteristics where it runs. This layer is also exclusively responsible for information exchange among different replicas (or nodes) which COPLA consists of.

The last layer, called Uniform Data Store (UDS) [11], is responsible for storing the state of persistent objects in an RDBMS. It has been defined an interface, which is exported by the UDS and called UDS-API, through which objects can be stored and retrieved, thus hiding all "relational issues" to the rest of the system. Since objects are created, accessed and modified inside a session context, this session is respectively mapped to a transaction in the RDBMS with the proper isolation level. This layer translates all queries performed by an application into normalized SQL queries. Finally, the UDS is used to store in a persistent way all the control information needed by consistency protocols. This control information is stored and accessed by means of the respective interface, the Persistent Metadata API.

IV. Algorithm Description

A. Introduction

COPLA is a session generator, a session may be viewed as a set of sequential transactions. In the following algorithm description the terms session and transaction may be used in an interchangeable manner. When a site updates an object replica, it requests the respective local lock, but the request of the remainder locks in the rest of copies is delayed until the initial commit phase is reached. The session master site sends its update information once the commit phase is reached (which we will refer as the pre_commit state). This request contains the list of all objects to be updated. Each remote updater is requested to acquire the *copy-locks* (similar to a *write-lock* in terms of its compatibility) of all of them during this phase. When all locks are acquired at all remote updaters, the session

is ended and changes are persistently stored (committed). As in 2PL [14], possible global deadlocks may occur, which we prevent by the proper deadlock prevention technique.

It is important to note that in the COPLA architecture, we do not allow *blind writes* in regular sessions. Therefore, a given session must acquire firstly a *read-lock* before requesting for a *write lock* on that object. A special case occurs when an update is performed by a remote session, i.e. when it requests a *copy-lock* on that object. These kind of sessions do perform *blind writes*.

	read-lock	*write-lock*	*copy-lock*	*recover-lock*
read-lock	y	n	n	y/n
write-lock	n	n	n	n
copy-lock	n	n	n	n
recover-lock	y/n	n	n	y

TABLE I
THE COMPATIBILITY MATRIX.

Current solutions of replicated systems are able to mask site failures efficiently, but many of them have not described their recovery of failed sites, merging of partitions, or joining of new sites. Reconfiguration that is necessary when the number of sites increases is a far more complex task than that necessary when the number of sites decreases. In particular, before a site can execute transactions, an "*up-to-date*" site has to provide the current data state to the joining site [5]. We use a membership protocol to provide support for failure detection [23]. Moreover, its services are used by our multicast services to ensure uniform delivery [4] and reliable communication; i.e., all sent messages are finally delivered to their destinations. Besides this, our consistency protocol uses all the failure and joining notifications to fire the recovery subprotocol start.

A recovering site joins the group of working sites, triggering a "*change-of-view*" action by the proper group membership monitor [23]. We consider a *primary partition* system model. In this context, we propose a recovery protocol sketch closely related to our concurrency algorithm. Therefore, once a node has failed in our architecture, the concurrency algorithm stores object identifiers (OIDs) modified by sessions on a given object repository. When a node recovers from its failure, a previously alive node with the lowest node identifier (NID) multicasts a message saying that a new node is rejoining the system. When the previous message is delivered, previously alive nodes request a special lock called *recover-lock*, whose compatibility is exactly the same as the *read-lock*, although it aborts all sessions currently updating the given objects instead of blocking itself, as it will be shortly explained. Current sessions reading objects to be recovered will not be affected by this new lock. After these special locks have been assigned, fore-coming sessions trying to update an object will be blocked following the ordinary policy established by the respective compatibility matrix depicted in Table I.

Concurrently to ordinary sessions, nodes that behave as $recoverer$, will determine the amount of OIDs to be transferred to the $recovering$ node. Thus, the recovery task is equally balanced among all available nodes. The $recovering$ node will hold *recover-locks* on all OIDs that will not allow local sessions to read or write on it. Nevertheless, local sessions in the $recovering$ node may start as soon as the site recovers, and they will behave as normal sessions, i.e. they will not be aborted unless they access an object whose OID has a *recover-lock* set on it. Therefore, this recovery algorithm does not need to block any replica, or even the recovering replica.

Object state transfer among available nodes, may be accomplished using different approaches. One approach is to send all objects grouped in a single message, which may be costly in terms of message size, or, alternatively, send one object state after the other, which may be costly in terms of number of messages too. We have determined to follow the latter approach. Once the $recovering$ node finishes applying all the updates, it sends an "*I am alive message*", which commits missed updates on the recovering site and releases all *recover-locks* held in the system.

B. Datatypes

Datatype	Content
NID	$\langle node \rangle$
SID	$\langle node\colon id \rangle$
OID	$\langle class\colon repository\colon node\colon id \rangle$
OBJ_STATE	*State of an object* [11]
$UPDATE$	{*SQL update statements*}
$LOCKS$	$\{read, write, copy, recover\}$
$SESSION\_STATES$	$\{run, blocked, pre\_commit, commit, abort\}$
$NODE\_STATES$	$\{alive, recoverer, recovering, crashed\}$

TABLE II
DATATYPES USED BY THE ALGORITHM.

There are eight datatypes defined in our algorithm, they are introduced in Table II. The NID is the node identifier; it is composed by a number, that identifies in a solely manner a site in the COPLA architecture. Applications access their information by means of sessions, which is equivalent to a set of transactions sequentially executed. The datatype used to define a session is SID, which is composed of the site where the session was started and a unique session identifier inside that node. These two fields guarantee uniqueness inside COPLA.

Objects need to be solely identified in COPLA too. Thus, every object has a unique object identifier (OID) which consists of: its class name, the name of the application where it was defined, the node where it was created and a unique identifier inside that node where it was created. Again, all these fields uniquely identify an object in our architecture.

The next one, called OBJ_STATE, includes what is defined as object state in the COPLA architecture [11]. And, finally, the $UPDATE$ datatype, which is the log information of a given session. It mainly consists of a set of SQL sentences that have modified an object repository for a given session [11], since we are persistently storing objects in an RDBMS.

State variables kept by each site and repository	Content	Initial value
my_node_id	NID	NID_i
$number\_of\_nodes$	$Integer$	$MAX\_NUMBER\_OF\_NODES$
$node\_state$	$NODE\_STATES$	$alive$
$oids\_to\_transfer$	$\{o\colon o \in OID\}$	$\emptyset$
$oids\_to\_rec$	$\{o\colon o \in OID\}$	$\emptyset$
$prev\_view$	$\{n\colon n \in NID\}$	$\emptyset$
$current\_view$	$\{n\colon n \in NID\}$	$\{NID_1 \dots NID_{MAX\_NUMBER\_OF\_NODES}\}$
$lock\_table_{1..MAX\_OID}$	$\begin{cases} lock\_table_o.assigned = \{\langle SID, LOCKS\rangle\} \\ lock\_table_o.waiting = \{\langle SID, LOCKS\rangle\} \end{cases}$	$\forall i \in \{1..MAX\_OID\}\colon \begin{cases} lock\_table_i.assigned \leftarrow \emptyset \\ lock\_table_i.waiting \leftarrow \emptyset \end{cases}$
$session\_state_{1..MAX\_SID}$	$sesion\_state_s \in SESSION\_STATES$	$\forall i \in \{1..MAX\_SID\}\colon session\_state_i \leftarrow \emptyset$
$updates_{1..MAX\_SID}$	$updates_s \in UPDATES$	$\forall i \in \{1..MAX\_SID\}\colon updates_i \leftarrow \emptyset$
$wait\_response_{1..MAX\_SID}$	$wait\_response_s \in \{n\colon n \in NID\}$	$\forall i \in \{1..MAX\_SID\}\colon wait\_response_i \leftarrow \emptyset$

Procedures and functions used in the algorithm	Brief outline
$compatible(o : OID, \langle s, mode\rangle : \langle SID, LOCKS\rangle)$	It determines whether the lock request is compatible with the current lock assignment on o in the $lock\_table$. See Table I for the compatibility matrix.
$deadlock\_prev(o : OID, \langle s, mode\rangle : \langle SID, LOCKS\rangle)$	It checks, in the $lock\_table$, if the request must be blocked or if this session or another session assigned (or even blocked) must be aborted in order to avoid deadlock.
$local\_commit(s : SID)$	It commits the updates on the local RDBMS.
$local\_abort(s : SID)$	Rollbacks all updates performed on the RDBMS.
$release\_locks(s : SID)$	This method releases locks assigned (or waiting) to the given session.
$oids\_written(s : SID)$	It returns the set of OIDs written by s.
$update(s : SID)$	This returns the log containing all the updates done by s in the given object repository.
$apply\_updates(s : SID)$	It applies the given updates to the RDBMS.
$det\_oids(my\_node\_id : NID, oid : \{oids : oids \in OID\})$	It returns a set of OIDs to be transferred to a $recovering$ node.
$get\_state(o : OID)$	It returns the object state (OBJ_STATE) associated to the given OID passed as a parameter.
$apply(o : OID, state : OBJ\_STATE)$	It applies, in the $recovering$ node, all the updates missed for the given object in the RDBMS.
$min(view : \{nid : nid \in NID\}, node\_id \in NID)$	A boolean function returning `true` if the second parameter is the lowest among all those contained in $view$.
$new\_session()$	It creates a new session, returning a SID, to perform the recovery process each time a node fails.

Fig. 2. States kept by each site and procedures used by the algorithm.

Different modes of lock request by sessions are defined in $LOCKS$. Similarly, as it will be explained on the sequel, sessions and nodes may switch among several states, those included in $SESSION\_STATES$ and $NODE\_STATES$ of Table II.

C. State Variables

The first seven state variables deal with an object repository recovery process, see Figure 2. We consider the COPLA architecture to be composed by a fixed set of nodes ($MAX\_NUMBER\_OF\_NODES$). Each node is uniquely identified by an NID, this identifier allows nodes to identify which nodes are running and which nodes have failed. The variables containing this information are $prev\_view$ and $current\_view$. The key concept dealing with this group of variables is to properly manage the change of view action fired by the respective membership monitor [23], each time a node fails or rejoins the system. This action modifies the current set of reachable nodes.

The node state for a given object repository is contained in $node\_state$. Figure 3 shows all possible states and transitions among these states for a node in the COPLA architecture, under this recovery algorithm assumption. Nodes that are working, i.e. accepting and executing sessions, are said to be $alive$. Whenever a failure happens, the node has $crashed$. Once the node is restarted, the node is said to be in the $recovering$ state. If a site enters in the $recovering$ state, one or several nodes may be chosen to act as a recoverer of that site, thus they enter in the $recoverer$ state.

Information about objects missed by nodes crashed is stored in $oids\_to\_rec$, as well as the number of objects to be transferred by a previously alive node or to be received by a recovering node are stored in $oids\_to\_transfer$.

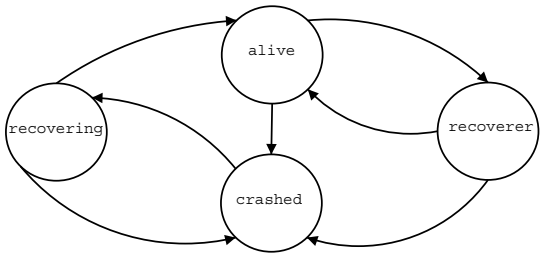

Fig. 3. Node state transitions diagram.

The remainder state variables are needed to maintain consistency on each object repository opened on each node. They manage how sessions access objects and which is the current state of a given session, as it can be seen in Figure 2 along with its initial values.

The $lock\_table$ state variable is a hashtable where each key ($lock\_table_i$), i.e. the i-th object belonging to the given

object repository, two sets as its respective value: the first one contains sessions that acquired a lock on that object; and, the other one contains sessions requesting an incompatible lock on the given object. Initially, this variable contains all OIDs with their respective queues being empty.

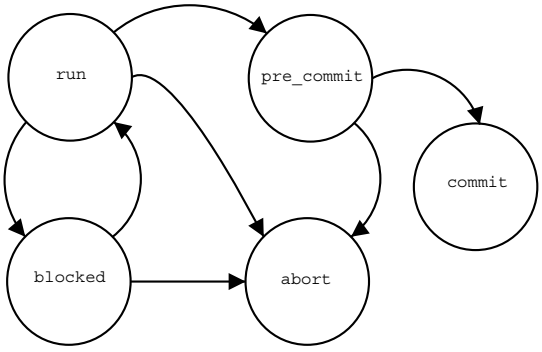

Fig. 4. Session state transitions diagram.

The $session\_state$ variable monitors the state of all sessions (local or remote) accessing a given object repository. There are defined several states that a session may pass during its life: run, $blocked$, pre_commit, $commit$, $abort$. This variable is a hashtable too. For each key entry has, as its associate value, one of all of the states shown in Figure 4.

A session starts in the run state, so it can start requesting locks on objects. It continues on this state until it performs an object lock request which is not compatible with the lock currently assigned on that object. Once the session reaches this situation, two possible state transitions may occur. Since the deadlock prevention function is invoked for that request, it will determine if the given session will be $blocked$ (if it is deadlock free), or switched to the $abort$ state. Otherwise, another session owning the lock, or waiting on that lock, may become aborted too.

If the session successfully obtains all the locks it requested (it may, or not, have switched several times from the run to $blocked$ state), it reaches the end of transaction. At this point it has decided whether to commit or abort. If it commits it moves to another state (pre_commit). At that time it summarizes all objects it has updated, and requests the locks for those objects in all available sites at that moment (following a ROWAA policy). If this session takes on all the locks at all available systems then it will be ready to commit, so it switches to the $commit$ state. Otherwise, if it decides to abort then it will change its state to $abort$, this session will be rollbacked on the underlying RDBMS and all locks will be released; no communication is necessary since the session is entirely local.

The following two state variables are closely related to the pre_commit state. The first one ($updates$) contains the log for each session running in COPLA. The other one ($wait\_response$), stores all available nodes for each session that has entered the pre_commit state, and whose answers have not yet been received –about *copy-lock* acquisition for that session on each of these nodes– by the master site of such session.

D. Procedures Used in the Algorithm

The algorithm, apart from its specified actions, will need helper functions and procedures in order to work properly. These special procedures are introduced in Figure 2. The first one ($compatible$) is a boolean function, it determines if a lock assigned on an object conflicts with the current lock request. Two locks conflict if they are on the same data item, they are issued by different sessions [14], and one, or both of them, are *write-lock*, *copy-lock* or *recover-lock* requests, as it is determined by the respective compatibility matrix shown in Table I.

If the lock requested is incompatible with the current lock assigned on an object, a deadlock situation may occur [13], [14]. In this first version of the algorithm we will employ a deadlock prevention technique. The function responsible for this task is $deadlock\_prev$. It returns a data structure consisting of a boolean and a SID. The boolean field assesses whether a deadlock situation may occur. If it is liable to suffer deadlock the SID returned is the session that must be aborted in order to prevent it, otherwise the lock request is queued.

Deadlock prevention techniques are based on setting up an order in lock assignment so cycles are prevented. The first deadlock prevention technique used is, as suggested by [13] giving a *copy-lock* request precedence over a *write-lock* request, provided that the session owning the latter has not reached the pre_commit state. In such a case, and invoked each time a session becomes blocked, we have defined a deadlock prevention technique for setting up an order in lock assignment, following a *wait-die* scheme [24]. The ordering factor is based on parameters of the given session, like: session restarting ratio, session execution time, number of objects read, number of objects written, log size, etc. In this first version we propose to use the SID as the ordering factor; it uses the id field as the first ordering factor, and, if they are equal, the $node$ field acts as the remaining ordering factor.

Two procedures are also closely tight, the first one called $update$ returns the log associated to the current session. A log in COPLA consists of all the SQL statements that have modified the given object repository for the given session. The second one ($apply\_updates$), applies all the SQL statements contained in a log into the DBMS, in our case an RDBMS. Details of its implementation are given in [11], [12]. The rest of the procedures are briefly sketched on Table 2.

The next two procedures are closely related ($local\_commit$, $local\_abort$), they are RDBMS direct invocations. The first one makes all changes done to the object repository persistent and the other one rollbacks all changes done in the RDBMS [11].

The $oids\_written$ function returns a set of OIDs that the current local session has modified along its execution life. This function is called when the pre_commit state is reached so as to acquire *copy-locks* on these objects in the remaining alive nodes.

The next two variables are also intimately tight. The first one, called $update$, returns the log associated to the current session. The second one applies the session log into the RDBMS [11], [12].

The following group of procedures and functions are related to the recovery process. Therefore, the det_oids function returns the amount of objects to be transferred to a $recovering$ node by the group of nodes inside the $recoverer$ state, once the former rejoins the system. Closely related to the previous one is the get_state function which returns, as it was previously depicted, the state of an object in the COPLA architecture [11]. The procedure utilized to apply object state updates in the $recovering$ node is $apply$; it performs the update process of the object state in the RDBMS.

One of the key processes used to fire the recovery is sending a message to the available nodes pointing out that a recovery process is about to start. This event is fired by the membership monitor [23], although the specific recovery task is initiated by the node whose NID is the lowest among all available nodes, which is fixed by the min function. Each time a recovery process takes place inside COPLA, a new session must be created so as to proceed, which is returned by the $new\_session$ function.

E. Communication Model

This protocol does not rely on strong group communication primitives to maintain data consistency, such as [18], [20] do. We assume that the communication channel is reliable without losses. We consider a fixed set of nodes composing the system. Nodes only fail by crashing, we do not consider byzantine failures. We use a group communication system providing group membership monitoring (nodes currently reachable) and a unicast and a reliable FIFO multicast [4] as information exchange channels between nodes. The notification of a node failure is done by firing a $change\_of\_view$ action. Therefore, nodes rejoining the system may be easily determined if the system stored the $current\_view$ and the $prev\_view$ of the node, which are included in Figure 2 as node state variables of our concurrency algorithm.

F. Actions Performed by the Algorithm

Actions described are needed for each object repository on a given node. As it has been highlighted in the introduction of the algorithm, sessions are started at a given node, and begin to request all locks locally. It has also been pointed out that COPLA does not allow *blind-writes* to be locally performed, therefore a *read-lock* must be requested before acquiring a *write-lock* on an object.

Once the session is done reading and writing objects, it reaches the pre_commit state. At this state, the concurrency control at the master site of the given session multicasts changes done as well as the OIDs where these changes were performed to the $current\_view$. Reception of this message at remote nodes starts a special session trying to acquire *copy-locks* on those objects. If these locks are granted at all remote nodes, then the session will switch to the $commit$ state and changes will be persistently stored at all sites.

Since this algorithm includes concurrency control and recovery tasks in behalf of cleanliness, it may be better explained, grouping both tasks in different parts: first, the concurrency control accomplished by our algorithm proposal, and, next, the recovery process.

1) Concurrency Control: Each time a session requests a lock, no matter what kind of lock is requested, a $lock\_request$ action is executed. This action checks whether the lock assignment is compatible with the current sessions holding the lock. In such a case the lock is granted and the session may continue requesting new locks (*running* state), otherwise the session must be *blocked* or *abort*ed. This is determined by the deadlock prevention function.

Whenever a session becomes *blocked*, it cannot request any lock until the session is woken up again. This event may occur whenever a release lock operation is performed in the object where the session is waiting. At this point we will briefly outline the release lock policy followed in our algorithm.

The only case where there are several sessions assigned on an object is when they request a *read-lock* on it. Thus, no waiting session will access that object until all read locks are released. Special cases arise when the session assigned to an object is a *write-lock* or *copy-lock*. Quite often there will be sessions waiting to acquire a *read-lock* or a *copy-lock* on that object. It will not happen that a *write-lock* is waiting to be assigned, since we do not allow *blind-writes*. Due to our deadlock prevention policy, the *copy-lock*, if it exists, will be located at the last position of the queue.

A session may be aborted due to two reasons: the final user decides to abort the current session; or, the deadlock prevention function determines that the given session must be aborted in order to prevent a deadlock. In the first situation, the session is always local and the tasks to be done are: aborting changes in the RDBMS, releasing all locks held by the session and switching the session state to $abort$.

Aborts induced by the deadlock prevention technique are treated quite different, since they usually involve message exchange with other COPLA sites. This case corresponds to a session in the pre_commit state. As it is depicted in the $abort$ action of Figure 5, it sends a msg_abort multicast message, due to the fact that it is currently requesting *copy-lock*s to all available nodes. Receiving a msg_abort means that a remote session, currently executed in the node where the message is received, has been aborted. This message is generated by the master site where the session originally started and implies, as it is depicted in Figure 5, the session abortion on the site where it was delivered.

The other message is sent when the abort is generated by the algorithm and the resulting session aborted is a remote session. This message will be mainly sent due to conflicts with local reads, which can be a potential global deadlock source since they are not propagated.

The remainder situation occurs when the deadlock prevention function aborts a local session that has not yet reached the

```
□ lock_request(o, ⟨s, mode⟩): o ∈ OID,
    ⟨s, mode⟩ ∈ {s, m: s ∈ SID, m ∈ LOCKS}
if mode = recover then
    ∀⟨sid, m⟩ ∈ lock_table_o.assigned :
        if (m = write) ∨ (m = copy) then abort(sid)
        else if m = read ∧ node_state = recovering then
            abort(sid)
        else break;
if compatible(o, ⟨s, mode⟩) then
    lock_table_o.assigned ← ++{⟨s, mode⟩};
    session_state_s ← run
else
    ⟨sid, result⟩ ← deadlock_prev(o, ⟨s, mode⟩);
    if result then abort(sid)
    else
        session_state_s ← blocked;
        lock_table_o.waiting ← ++{⟨s, mode⟩};

□ abort(s): s ∈ SID
if s.node = my_node_id then
    if session_state_s = pre_commit then
        multicast(current_view, msg_abort(s))
    else
        session_state_s ← abort;
        local_abort(s); release_locks(s);
else send(s.node, msg_remote_abort(s));

□ receive_msg_remote_abort(s): s ∈ SID
if session_state_s ≠ abort then
    session_state_s ← abort;
    local_abort(s); release_locks(s);
    multicast(current_view, msg_abort(s));

□ receive_msg_abort(s) : s ∈ SID
if session_state_s ≠ abort then
    session_state_s ← abort;
    local_abort(s); release_locks(s);

□ pre_commit(s) : s ∈ SID
session_state_s ← pre_commit;
multicast(current_view,
    msg_update_session(s, oids_written(s), update(s)));

□ receive_msg_update_session(s, oids, update) :
    s ∈ SID, oids ∈ P(OID), update ∈ UPDATE
if s.node = my_node_id then
    wait_response_s ← current_view;
else
    update_s ← update;
    copy_locks_request(s, oids);

□ copy_locks_request(s, oids): s ∈ SID, oids ∈ P(OID)
∀o ∈ oids :
    if session_state_s ≠ abort then lock_request(o, ⟨s, copy⟩);
    else break;
send(s.node, msg_ready(my_node_id, s));

□ receive_msg_ready(node, s): node ∈ NID, s ∈ SID
wait_response_s ← −{node}
if wait_response_s = ∅ then
    multicast(current_view, msg_commit(s));

□ receive_msg_commit(s): s ∈ SID
if s.node ≠ my_node_id then
    apply_updates(updates_s);
if #current_view ≠ number_of_nodes then
    oids_to_rec ← ++{oids_written(s)};
session_state_s ← commit;
local_commit(s); release_locks(s);

□ change_of_view(nodes): nodes ∈ P(NID)
prev_view ← current_view; current_view ← nodes;
if #current_view > #prev_view then
    if min(prev_view, my_node_id) then
        multicast(current_view,
            msg_recov_obj(new_session(), oids_to_rec, prev_view));
    else break;
else
    failed_nodes ← prev_view \ current_view;
    ∀s ∈ SID:
        if s.node ∉ current_view then abort(s)
        else if session_state_s = pre_commit then
            wait_response_s ← wait_response_s \ failed_nodes

□ receive_msg_recov_obj(s, oids, view) :
    s ∈ SID, oids ∈ P(OID), view ∈ P(NID)
session_state_s ← run; recov_lock_request(s, oids);
if #current_view = number_of_nodes then
    oids_to_rec ← NIL
if my_node_id ∈ current_view \ prev_view then
    prev_view ← view; node_state ← recovering;
    if #current_view ≠ number_of_nodes then
        oids_to_rec ← oids;
    oids_to_transfer ← oids;
else
    node_state ← recoverer;
    oids_to_transfer ← det_oids(my_node_id, oids);
∀o ∈ oids_to_transfer :
    send(current_view \ prev_view,
        msg_obj_update(s, o, get_state(o)));
oids_to_transfer ← NIL

□ receive_msg_obj_update(s, o, state): s ∈ SID, o ∈ OID,
    state ∈ OBJ_STATE
oids_to_transfer ← −{o};
apply(o, state);
if oids_to_transfer = ∅ then
    local_commit(s);
    multicast(current_view, msg_alive(s));

□ receive_msg_alive(s): s ∈ SID
node_state ← alive;
session_state_s ← commit; release_locks(s);
```

Fig. 5. Actions performed by the algorithm.

pre_commit state; in such a case no messages are generated since the session is entirely local. Thus all locks are released and the session state is changed to $abort$.

O2PL philosophy consists of performing all operations locally until the final user commits. This is performed by the pre_commit action. At that time (session moves to the pre_commit state), we have to update the rest of sites. During this update propagation, the session may be aborted due to existing conflicts with other transactions.

Once the user decides to commit the current session, the algorithm asks for all objects written in the session. It also requests for all SQL statements that have modified the given object repository where the session has been executed. All this information is used to build a $msg\_update\_session$ message that is multicast to all available nodes in the current view.

When the $msg\_update\_session$ is received by the master site of the session it builds the data structure that contains a queue of all the available node identifiers. At the remote sites execute the $copy\_locks\_request$ action. It starts, for the given session, requesting *copy-locks* as many as objects modified by the given session. This is a special action, since it is not atomic, and the session will become blocked or, even worse, aborted. Each time a node is finished requesting all *copy-locks*, it generates a msg_ready message which is sent to the master site.

Receiving a msg_ready will remove the message source node from the $wait\_response$, and once this queue is empty, it sends a msg_commit message to all nodes so changes are

persistently applied throughout all the COPLA architecture and the session state is changed to $commit$.

2) Recovery Process: The recovery process is started by the group membership protocol [23], invoking a $change\_of\_view$ action. This action has as a parameter the current set of available nodes. If its size is less than the $number\_of\_nodes$, then the msg_commit action will store the set of OIDs modified by subsequent sessions. If this action is a result of a rejoining node, then the node with the minimum NID will multicast a message containing the OIDs updates missed while the rejoining node has been $crashed$, as well as the previous view of the system, so that the $recovering$ node stores it. It also creates a new SID to manage the recovery process throughout all the COPLA architecture, thus the recovery process is considered as a new session is in charge of transferring the last state of missed object updates.

The reception of a msg_recov_obj implies, firstly, a common task which consists of requesting for *recover-locks* on all OIDs missed by the recovering node. As it was previously stated, this will lead to a set of session abortions trying to update the objects missed by the $recovering$ node. Continuing with the $recovering$ node, no new session is allowed to proceed until all *recover-locks* are assigned. This action will generate the appropriate data structure to process all updates transferred by the $recoverer$ nodes. Thus, the $recoverer$ nodes estimate the number of objects each one is responsible for transferring to the $recovering$ node. As it was previously explained, we have selected the option of sending a separate message for each object to be transferred. This action, i.e. receiving a msg_obj_update, applies the update for the received OID in the underlying RDBMS inside the context of the recovering session. Once it is done receiving all updates, it sends a msg_alive multicast to all available nodes that will commit the given session on the $recovering$ node, release all *recover-locks* and switch all available sites to the $alive$ state. Besides, if every node is not yet recovered, OIDs modified are stored at all nodes, this OIDs log will be erased once the last node of the system has finally been recovered.

During this recovery process, new sessions may be started at all nodes, enhancing system availability. Nevertheless, there is an extra cost of restarting transactions, provided that they are trying to update OIDs involved in the recovery process on nodes in the $recoverer$ state. Moreover, a node in the $recovering$ state may start new sessions too, after it sets all *recover-locks*, but at the expense of no accessing on OIDs with a *recover-lock* set on it, all these sessions will be automatically aborted.

V. Conclusions

This paper introduces the design of an Optimistic Two Phase Locking (O2PL) consistency protocol to be implemented in a middleware architecture called COPLA [7]–[9], [11], [12]. This architecture supports transactional access to persistent transparently replicated objects. This protocol is mainly based on the ideas proposed in [13], but adapted to the COPLA architecture. Since our protocol is liable to suffer global deadlocks, we have defined a global deadlock prevention technique, which is based on flexible prevention techniques, such as session information like the following: number of objects read or written, number of session restarts, log size, etc. Right now, we are only considering the session identifier (SID) as the ordering factor on lock requesting.

Besides, this paper introduces a preliminary design of a recovery protocol to be used by COPLA along with the O2PL consistency protocol. We have introduced a new kind of lock, called *recover-lock*, which is used to enhance system availability whenever there is a node joining COPLA after its failure. The main goal pursued with this recovery protocol is to continue executing transactions at all nodes, even in the node being recovered. This new lock is set on all objects updates missed by a failed node so as to perform the object state transfer of these missed objects [11]. This lock has a different behavior depending on the node where it has been assigned. If it is in a $recoverer$ node then it will allow any other transactions to acquire a *read-lock* on it, but no any other lock. Otherwise, it is assigned on a $recovering$ node and no other transaction may access that object.

Presently, we are implementing this protocol in COPLA so as to compare its behavior with currently implemented protocols [7]–[9] and different application workloads. We are also very interested in comparing this algorithm with actual solutions proposed in the literature such as [18], where deadlock is prevented based on the total order of the group communication used to propagate the updates performed on a repository, against our approach. Since it utilizes different deadlock prevention techniques and with several deadlock detection and resolution algorithms.

Acknowledgment

This project is supported by the Spanish Government as a CICYT project under research grant TIC2003-09420-CO2.

References

[1] A.A. Helal, A.A. Heddaya, and B.B. Bhargava. *Replication techniques in distributed systems*. Kluwer Academic Publishers, USA, 1996.

[2] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proc. of the 20th International Conference on Distributed Computing Systems (ICDCS)*, pages 464–474, Taipei, Taiwan, April 2000. IEEE-CS Press.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. of the 1995 ACM SIGMOD international conference on management of data*, pages 1–10, San Jose, USA, May 1995.

[4] V. Hadzilacos, and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. *Distributed Systems, Second Edition*. S. Mullender, ACM Press/Addison-Wesley, USA, 1993.

[5] B. Kemme, A. Bartoli, and Ö. Babaoğlu. Online reconfiguration in replicated databases based on group communication. In *Proc. of the International Conference on Dependable Systems and Networks (DSN 2001), Goteborg, Sweden*, pages 117–127, June 2001.

[6] J.E. Armendáriz, J.R. González de Mendívil, and F.D. Muñoz-Escoí. Working on GlobData: An efficient software tool for global data access. In *Proc. of Workshop on Research and Education in Control and Signal Processing (REDISCOVER 2004), Accepted*, Cavtat, Croatia, June 2004. IEEE Press.

[7] F.D. Muñoz-Escoí, L. Irún-Briz, P. Galdámez, J.M. Bernabéu-Aubán, J. Bataller, and M.C. Bañuls. GlobData: Consistency protocols for replicated databases. In *Proc. of the IEEE-YUFORIC'2001, Valencia, Spain*, pages 97–104, November 2001.

[8] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. Strong replication in the globdata middleware. In *Proc. Workshop on Dependable Middleware-Based Systems (Supplemental Volume of the 2002 Dependable Systems and Networks Conference), Washington D.C., USA*, pages G96–G104, June 2002.

[9] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. The GlobData fault-tolerant replicated distributed object database. In *Proc. of the First Eurasian Conference on Advances in Information and Communication Technology, Teheran, Iran*, October 2002.

[10] PostgreSQL Home Page. `http://www.postgresql.org` June 2004.

[11] J.E. Armendáriz, J.J. Astrain, A. Córdoba, J. Villadangos, and J.R. González de Mendívil. A persistent object storage service on replicated architectures. In *Proc. of VI Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software (IDEAS 03), Asunción, Paraguay*, pages 133–144, April 2003.

[12] J.E. Armendáriz, J.J. Astrain, A. Córdoba, and J. Villadangos. Implementation of an object query language for replicated architectures. In *Proc. of VIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 03), Alicante, Spain*, pages 441–450, November 2003.

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

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

[15] M. Xiong, K. Ramamritham, J. Haritsa, and J.A. Stankovic. MIRROR: A state-conscious concurrency control protocol for replicated real-time databases. In *Proc. of 5th IEEE Real-Time Technology and Applications Symposium*, pages 100–110, Vancouver, Canada, June 1999.

[16] K. Hasegawa, and M. Takizawa. Object-based locking protocol for replicated objects. In *Proc. of 13th International Conference on Information Networking (ICOIN '98)*, pages 398–402, Tokyo, Japan, January 1998. IEEE-CS Press.

[17] J. Jing, O. Bukhres, and A. Elmagarmid. Distributed lock management for mobile transactions. In *Proc. of 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 118–126, Vancouver, Canada, June 1995. IEEE-CS Press.

[18] B. Kemme, and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Trans. on Database Sys.*, 25(3):333–379, September 2000.

[19] A. Silberschatz, H.F. Korth, and S. Sudarshan. *Database Systems Concepts, Fourth Edition*. McGraw-Hill Science/Engineering/Math, USA, October 2001.

[20] R. Jiménez-Peris, M. Patiño-Martínez, and G. Alonso. Non-intrusive, parallel recovery of replicated data. In *Proc. of 21st Symposium on Reliable Distributed Systems*, pages 150–159, Osaka Univ., Suita, Japan, October 2002. IEEE-CS Press.

[21] P.A. Bernstein, and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. on Database Sys.*, 9(4):596-615, December 1984.

[22] ODMG home page. `http://www.odmg.org` June 2004.

[23] F.D. Muñoz, O. Gomis, P. Galdámez, and J.M. Bernabéu. HMM: a cluster membership service. In *Proc. of the 7th International Euro-Par Conference*, Volume 2150 of LNCS, pages 773–782, Manchester, United Kingdom, August 2001.

[24] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems, Third Edition*. Addison-Wesley, USA, 2000.