A Semi-Optimistic Concurrency and Recovery Algorithm Design for the COPLA Software Architecture

J. E. Armendáriz, J. R. González de Mendivil, F. D. Muñoz-Escoí
{enrique.armendariz, mendivil}@unavarra.es, fmunyoz@iti.upv.es

Technical Report ITI-ITE-04/03
A Semi-Optimistic Concurrency and Recovery Algorithm Design for the COPLA Software Architecture

J.E. Armendáriz, J.R. González de Mendivil
Grupo de Sistemas Distribuidos
Universidad Pública de Navarra
Campus de Arrosadía s/n
31006 - Pamplona; España
e-mail: {enrique.armendariz, mendivil}@unavarra.es

F.D. Muñoz-Escó
Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Camino de Vera s/n
46022 - Valencia; España
e-mail: 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, in fact 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. Since this algorithm is a lock-based one, it is liable to suffer global deadlock cycles formation, thus we present a deadlock prevention algorithm fully integrated with the recovery algorithm.

*A reduced version of this report has been published as a regular paper in the 38th Hawai’i International Conference on System Sciences*
1 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 these 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, 8, 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 this technical report is organized as follows: Section 2 gives a brief review of works dealing with O2PL
and recovery; Section 3 is devoted to give an outline of COPLA architecture; Section 4 outlines the communication
model used in this algorithm; respectively, the transaction model is sketched in Section 5; and, the concurrency
control and recovery algorithm description is depicted in Section 6. Finally, we include some proofs about the
correct behavior of our algorithm proposal.

2 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 communica-
tion 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, 8, 9] and
Bernstein et al. solutions to the recovery problem in distributed databases [21].
3 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 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 [22, 12] 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, 8, 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.

Figure 1: COPLA Architecture.
4 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 uniform reliable FIFO multicast [4] as information exchange channels between nodes. FIFO multicast consists of broadcasting a message to all available nodes.

![Figure 2: View change in our system.](image)

If a reliable multicast is used, each site can deliver a message to the application immediately after its reception or after all pending messages from the same sender have been delivered. Otherwise, if a uniform reliable multicast is used all messages delivered at any node will be delivered to all available nodes in that view. This is more clearly depicted in Figure 2, as an example of the difference between reliable and uniform reliable, we have to consider message $m$ sent by node $N_1$, if we use a reliable multicast, the communication group will immediately deliver back $m$ to $N_1$, although none of the remainder sites has received the message. Using uniform reliable delivery, $N_1$ will not immediately deliver the message but would only do it when it knew that the rest of available nodes have received it.

In order to handle message loss and failures, the communication module on each node acknowledges the reception of a multicast message from the network. It keeps a message until it has received the acknowledgments for the message from all sites. Hence, all available sites have exactly the same messages and will deliver the same set of messages before delivering the view change message. Virtual synchronization usage allows nodes to see a sequence of communication events, where an event is, for example: sending a message, delivering a message or a view change.

Our algorithm assumes that we use virtual synchronous communication using a FIFO reliable multicast, always working in a primary view. Each view change event may easily determine nodes crashed and rejoining the system, since the difference between available nodes at each view may give nodes rejoining and crashing. The notification of the set of nodes currently available is done by firing a $view\_change$ action. Therefore, nodes rejoining the system may be easily determined if the system stored the current view ($\mathcal{V}$) and the previous view of the node ($pre\_\mathcal{V}$).

5 Transaction Model

A final user accesses an object repository by means of sessions. A session may be viewed as a set of sequential transactions. A session $S_i$ is a partial order, $\prec_i$, of read ($r_i(X)$) and write operations ($w_i(X)$) on logical objects
Each read operation is performed in the local physical copy of $X$ ($X_i$), write operations are firstly performed on the local copy. When the end of transaction is reached, all write operations are transformed into copy operations ($c_i(X)$) on the remainder available nodes, these are equivalent to write operations on remote nodes [13]. This situation may be best viewed as executing special remote sessions at all available sites that exclusively perform write operations. Therefore we follow a read one/write all available (ROWAA) policy.

There are also special transactions fired each time a node rejoins from a crash failure. These transactions are called recovery sessions, which request for reading all object updates missed by a recovering node. It is composed by a set of recovery operations ($recov_i(X)$). This operation allows a recoverer node to read locally the value of $X$ ($X_i$), and write it in the recovering site ($X_j$, where $j$ is the recovering node).

![Diagram](image)

**Figure 3:** Execution model of a given session with no failures.

### 6 Algorithm Description

#### 6.1 Concurrency Algorithm

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. As we have previously cited our concurrency algorithm is an adaptation of [13] to COPLA, on which all transactions are locally executed. Thus, all reads are locally performed, 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, i.e. when the user has finished requesting locks and performed all operations. 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, see Table 1) 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 it is introduced in Figure 3. As in 2PL [14], possible global deadlocks may occur, which we prevent by the proper deadlock prevention technique.
Definition 1. A local session is a system session still on its local phase, i.e. still acquiring read-locks and write-locks.

Definition 2. A remote session is a system session that has finished its local phase and has propagated its updates to the rest of available nodes. Equivalently, a remote session is a local session that has reached the pre-commit state.

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.

<table>
<thead>
<tr>
<th></th>
<th>read-lock</th>
<th>write-lock</th>
<th>copy-lock</th>
<th>recover-lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>read-lock</td>
<td>y</td>
<td>n</td>
<td>y</td>
<td>y/n</td>
</tr>
<tr>
<td>write-lock</td>
<td>n</td>
<td>n</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>copy-lock</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>y</td>
</tr>
<tr>
<td>recover-lock</td>
<td>y/n</td>
<td>n</td>
<td>n</td>
<td>y</td>
</tr>
</tbody>
</table>

Table 1: The compatibility matrix.

6.2 Recovery Algorithm
6.2.1 Introduction

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 reliable delivery [4]; 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.

6.2.2 Algorithm Behavior

Each site stores in a a node state variables (node-state) an array containing the state of the node, that will be later on explained, and the age of a node. The age refers to the last time a site joined the system, thus the site with the lowest age is the oldest node of the system. Therefore, once a node has failed in our architecture, the system stores for the new view installed on the remainder available sites, inside another state variable called objects-to-recover, the set of node identifiers (NIDs) crashed and the object identifiers (OIDs) of the updates missed by them.

There are three possible situations of node failure: the first one includes a node that is currently running local transactions, its failure does not affect the state of the remainder available nodes. The second one arises when a node is executing a remote session, see on the left part of Figure 4, and it has not answered to the master site about the copy-lock acquisition. Upon firing the view-change action the session master site removes the failed node identifier from the set of nodes where it was waiting for the failed node response. The third situation occurs when the master site of a session that reached the pre-commit state, the rest of available sites aborts all its remote sessions, as it is sketched in the right hand side of Figure 4.

Whenever a node rejoins the system, the view-change action is fired and the recovery part of our algorithm proposal is started up. Every previously alive node will detect that a new node has joined and one of them will be in charge of the recovery process, more precisely the oldest one. This node determines the view (or views) missed by the joining site and multicasts these views (along with the OIDs and nodes) missed by the joining node to all available sites. The reception of this message starts a special session, called recovery-session, that requests for a new kind of lock to be set on all these objects that we introduce here. Before introducing it, it is important to note that if there are still some failed nodes, this action creates a new entry in the objects-to-recover variable containing for the new view the set of unreachable nodes.
Figure 4: Different kind of failures considered in our model when a site executing a session.

Definition 3. A recovery session is a system session in charge of recovering a given view for a recovering node. Hence, there will be as many sessions of this kind as views missed by a recovering node multiplied by the number of nodes being currently recovered.

As it was previously stated, a new lock is set at every available node, called recover-lock and its compatibility is shown in Table 1, on all missed objects by the joining site. As it is depicted in Table 1, its behavior is different depending on the kind of node where it has been set. When we refer to the kind of node, there are two kinds of nodes: recovering and previously alive nodes. In a recovering site this lock establishes that this object is outdated, thus it makes no sense to read or write on it. Respectively, in a previously alive node (including the one in charge of the recovering of the joining site) does make sense to read an object but not its modification.

Once these recover-locks are set at the node in charge of the recovery process, that we will denote from now on as an alive-recoverer node, will send the state of objects missed to the joining node, denoted as recovering-recoverer node, since it has been assigned a recoverer to the node. Respectively, the finalization of the recovery session on the recovering site will switch it to the recovering-recoverer state and local transactions may be started on this node as long as they do not interfere with OIDs being currently recovered. This process is introduced in Figure 5.

It is important to note that we do not interfere with transactions currently executed on previously alive nodes, unless they attempt to modify an object being recovered, and the recovering node starts executing local transactions once all the recover-locks are set, again sessions trying to access (in read or write mode) an outdated object will be aborted. Finally, all available sites, even those being recovered, participate in the pre-commit action at all times.

Once a recovering session assesses a view has been recovered, its node identifier (NID) is removed at all available sites from the objects_to_recover variable associated to that view. If a given view on that variable has no node associated then it is removed from the variables. Thus, it is assumed that the view has been entirely installed on the system.

The recovery process depicted until now corresponds to a failure free environment, neither the recovering nor the
recovery nodes fail. Thus, the recovery process is done without interruptions of any kind. If we go one step further and accept failures inside the recovery process, we can assume failures of a recovering, a recovering-recovering or an alive-recovering node.

Let us start with a recovering node failure, this corresponds to a node that has not yet received its missed views (along with the updated OIDs and the crashed nodes), but it has participated on sessions entering in the pre-commit state. Therefore, its NID is removed from the wait_response variable. According to our model description, this situation implies the abortion of the recovery sessions established in the remainder nodes, although we assume a uniform reliable channel and the start recovery message will not be received by any node, otherwise the joining node will be in the recovering-recovering state.

The failure of a recovering-recovering node implies the abortion of all the ordinary (as it is depicted in Figure 4) and recovery sessions associated to this node, as it is sketched in the left side of Figure 6.

The management of an alive-recovering failure requires more steps than the previous failure situations. First all remote sessions from that node will be aborted at all available sites, as it happen with transactions executed in the system. Besides, it is in charged of transferring all pending updates to all recovering nodes. Therefore, the new oldest node alive will take care of new recovery sessions and will check all recovery sessions pending on its site (those managed by the failed alive-recovering), and start the object state transfer to all recovering sites, see the right side of Figure 6.

6.3 Deadlock Prevention

Since this algorithm (both the consistency and recovery parts) is based on locks it is liable to suffer global deadlock situations. In order to avoid these situations we have to establish an order for lock acquisition between transactions. We introduce here another novelty of our algorithm proposal, our deadlock prevention algorithm is closely related to the recovery algorithm, since it guarantees the recovery of a site.
Our approach uses the kind of lock request, the state of a given session and, finally, the pair timestamp and NID in order to prevent deadlock cycles. There are several ways to prevent deadlock. In our algorithm, we follow the appropriation and rollback approach. The appropriation consists of whenever a transaction $T_2$ requests a lock on an object that another transaction $T_1$ already holds, $T_1$ is rolled back and the lock is granted to $T_2$. The management of these appropriation is managed by a unique identifier that is generated by the system so as to decide which session is going to be rolled back. We have decided to use wound-wait policy is used, inside this schema whenever a transaction $T_i$ requests for a data element already held by a transaction $T_j$, $T_i$ only waits if its order (i.e. the unique identifier) is less than $T_j$. Otherwise, $T_i$ is rolled back ($T_j$ wounds $T_i$).

We must be very careful about establishing in our algorithm proposal the order on which transactions are classified. As a first approach, we will only consider the elements highlighted at the beginning of the last paragraph. Following that order stated in the last paragraph, we will begin with the different kind of sessions considered in our algorithm from a top to bottom approach: recovery sessions are the most important ones, these sessions will abort sessions already holding an update lock on that object (write-lock or copy-lock), remember that no local session on a recovering node may access these objects. The second ones are: remote sessions already holding a copy-lock on an object, and local sessions that have been restarted $n$ times due to the deadlock prevention algorithm; a local session belonging to this last group is called elderly. The next kind of sessions are the remote ones (sessions that have reached pre-commit state), these sessions will abort all local sessions, not belonging to the previous level,
at all available sites holding a lock on objects updated by them. The last group is the set of local transactions currently on their acquisition phase and not being restarted for \( n \) times. Sessions belonging to the same level are ordered by the timestamp and NID pair.

**Definition 4.** A local session is called elderly whenever it is on its acquisition phase and it has been restarted by the deadlock prevention algorithm, due to recovery or remote sessions, for \( n \) times. The value of \( n \) should not be excessively great.

These elderly sessions are introduced to circumvent the problem of continuous restart of long-lived transactions, due to the presence of remote sessions conflicting with them. Let us show this with an example, consider one long session accessing an object that conflicts with a short periodic one at a remote node. Since the latter will easily reach the pre-commit state, it will abort locks already held by the former. This long session will be restarted countless number of times and it will never be committed. If we put a threshold of restarting times (say two or three times), this session will become soon an elderly one. The short session is put to sleep whenever a conflicting copy-lock request arrives from it. Therefore, the short session will not commit and the long session may reach the pre-commit state and multicast its updates to the rest of available sites. At that time, the timestamp and NID pair will prevail, and the oldest one will survive, assuring the completion of both, since there will be a time when the timestamp of the long session will be the lowest among all sessions running in the system.

![Figure 7: Establishing the order of sessions for deadlock prevention.](image)

The definition of the ordering session proposed here is like a regular plane (see Figure 7). The first ordering factor used is the \( y \) axis, it has four different possible values, ordered in an ascending manner, i.e. a recover session has more precedence than any other session, as it was earlier on explained. The second ordering factor is the \( x \) axis which is ordered by the timestamp and NID pair in an ascending fashion, the older a session is the greater value the session has on this axis.

Following the example shown in Figure 7, there are two sessions (\( S_1 \) and \( S_2 \)) that, at different nodes, have both reached the pre-commit state, if they try to access the same object, only one of both will survive. As they belong to the same kind of sessions, the only factor they differ is the timestamp (or the NID), since \( S_2 \) is greater than \( S_1 \) only the former will survive. The same can be applied to both sessions if a recover lock request from \( S_3 \) comes to that object, both will be aborted. If a local session (\( S_4 \)) tries to access this object, \( S_4 \) will wait until the lock is released by \( S_2 \). Finally, if \( S_2 \) or \( S_3 \) tries to access an object currently assigned to \( S_1 \), then the latter will be aborted. Again, \( S_4 \) will restart the session and if this sequence of abortions continues, the session may become an elderly one. The session \( S_1 \) is an example of an elderly session. It is a session that has been repeatedly rolled back. These kind of sessions will abort the remainder kind of sessions except the recovering ones.

At this point it is important to note that although we have defined a conflict between two operations over the same object that led to the compatibility matrix definition of Table 1. This definition is not enough for compatible
Figure 8: Deadlock cycle formation between compatible operations due to the existence of a prior waiting session.

operations between two sessions in order to grant deadlock prevention, specially when there are sessions waiting for
the same object. Let us see this with an example (see Figure 8), suppose we have session $S_1$ owning in read mode
the object $O_1$, $S_2$ owning $O_2$ in write mode. Some time after, $S_2$ requests for a write operation on $O_1$ since it is not
compatible, we must compare the timestamps and the result is that $S_2$ is younger than $S_1$ (the subindex reflects
the session timestamp) then $S_2$ will wait. Meanwhile, a new session is created ($S_3$), it issues a read operation on
$O_1$ (it does not conflict with $S_1$) and, afterwards, it requests for a write operation on $O_2$ as this is not a compatible
operation the timestamps must be compared, $S_3$ is put to sleep. Hence, we have a deadlock cycle among $S_2$, $S_3$,
$O_1$ and $O_2$. From this example we derive that additional checks must be done in order to grant an object to a
session whenever there are sessions waiting to access on that object.

6.4 Datatypes
- Node identifier. $NID \equiv id \in \mathbb{N}$.
- Object identifier. $OID \equiv \langle oid.class::oid.repository::oid.node::oid.id \rangle$.
- Session identifier. $SID \equiv \langle sid.node::sid.id::sid.timestamp::sid.restarts \rangle$.
- Object state. $OBJ\_STATE \equiv \text{State of an object}$.
- Session state. $SESSION\_STATES \equiv \{\text{run, blocked, pre-commit, commit, abort}\}$.
- Node state. $NODE\_STATES \equiv \{\text{crashed, recovering, recovering\_recoverer, alive, recoverer, alive}\}$.
- Session update (or transaction update). $UPDATE \equiv \{\text{SQL update statements}\}$.

6.5 Constants
- We consider a fixed set of nodes $\{1 \ldots N\}$. $NODES \equiv \{i : i \in 1 \ldots N\}$.
- We assume that there is a bounded set of objects in the system. In other words, there are at most $MAX\_OID$
objects in the system.
- We have defined a fixed four different valid modes for object accessing inside a repository. These are equivalent
to four different kind of lock request actions for each object defined in a repository. These request are: read-
lock, write-lock, copy-lock and recover-lock.
- We apply the same criterion as before to the session identifiers, there are at most $MAX\_SID$ sessions running
in the system. Each time a session is created it has a valid session identifier inside this range.
- There is also a maximum number of views. Each time a new view is generated, because of the current set of reachable nodes is modified, the membership monitor will generate an identifier inside this interval $1 \ldots \text{MAX\_VIEW}$.

- A threshold must be established so as to define the maximum number of times that a session must be started in order to become an elderly session, it is called \texttt{NUM\_RESTARTS}.

### 6.6 Node state variables for each object repository

- **Node identifier.** This variable uniquely identifies a node inside the replicated architecture. $i \in \text{NODES}$.

- **Lock table.** This table contains all objects created for each object repository containing sessions assigned and waiting to access due to an incompatibility with the current lock held in the associated object.

  $$L_{\mathcal{T}_i}[0 \ldots \text{MAX\_OID}] \equiv \begin{cases} L_{\mathcal{T}_i}[o_j].\text{assigned} = \{(s, m): s \in \text{SID} \land m \in \{\text{read, write, copy, recover}\}\} \\ L_{\mathcal{T}_i}[o_j].\text{waiting} = \{(s, m): s \in \text{SID} \land m \in \{\text{read, write, copy, recover}\}\} \end{cases}$$

- **Session state.** For each session created in this repository, this variable keeps track of their states throughout all their lifetime in the node.

  $$\text{session\_state}_i[s_1 \ldots \text{MAX\_SID}] \equiv \text{session\_state}_i[s_j] \in \text{SESSION\_STATES}$$

![Session state transitions diagram.](image)

The \texttt{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, \texttt{pre\_commit}, commit, abort. For each index entry has, as its associate value, one of all of the states shown in Figure 9.

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 \texttt{run} to \texttt{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 (\texttt{pre\_commit}) on the session master site. 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 nodes 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 rolled back on the underlying RDBMS and all locks will be released; no communication is necessary since the session is entirely local.
During the pre-commit state of a session at a master site, a remote node may decide that the given session must be aborted, mainly due to deadlock prevention among local and remote sessions, or remote session exclusively. Therefore a session may switch to the abort state due to a conflict at a remote site.

- The pre-commit phase. During this phase a session waits to receive a response from the rest of nodes about the acquisition of copy-locks on their own sites of objects modified on its master site.

\[ \text{wait\_response}_i[s_1 \cdots s_{\text{MAX\_SID}}] \equiv \{ j : j \in \text{NID} \} \]

- Objects to recover. This state variable stores for each non-completed recovery view the OIDs missed by crashed nodes. Each time a node finishes its recovery for a given view, it is deleted from the view entry. Besides if an object is modified on two consecutive views, only the last one will remain in this table.

\[
\text{objects\_to\_recover}_i[1 \cdots \text{MAX\_VIEW}] = \begin{cases} 
\text{objects\_to\_recover}_i[k].oids = \{ o : o \in \text{OID} \} \\
\text{objects\_to\_recover}_i[k].nodes = \{ j : j \in \text{NID} \}
\end{cases}
\]

- Recovery sessions. Each site has a variable that monitors sessions started to recover each view missed by a joining node. These sessions will be deleted as soon as the changes are persistently applied. Therefore, this variable will be an array, whose length is the total number of nodes, whose value are sets containing recovery sessions to be applied on the respective node. The structure of these session are special, since they must be global. Each one of them is composed by the node identifier of the node being recovered and the view to be recovered \((s.\text{node}\_i:s.\text{view}\_id)\).

\[ \text{recovery\_sessions}_i[1 \cdots \text{NODES}] \equiv \{ s : s \in (\text{NID}, \text{N}) \} \]

- Node state. It monitors the state of each node \((\text{status})\) and its age \((\text{join})\). This last field means the view in which this repository started its recovery.

\[ \text{node\_state}_i[n_1 \cdots n_{\text{NODES}}] = \begin{cases} 
\text{node\_state}_i[n_k].\text{status} \in \text{NODE\_STATES} \\
\text{node\_state}_i[n_k].\text{join} \in \{ 1 \cdots \text{MAX\_VIEW} \}
\end{cases} \]

Figure 10: Node state transitions diagram.

Figure 10 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. At this state no local session is started although it participates in remote updates produced by sessions coming from the rest of alive nodes. The transition from the recovering to the recovering\_recoverer state, is done once the oldest alive node, i.e. the one with the lowest \text{join} field on its node\_state state variable has multicast OIDs lost by the recovering node. A site entering in this state may start local sessions as long as they do not interfere with objects being recovered, these sessions may be immediately aborted.

As it was explained in the previous paragraph, the node with the lowest \text{join} field of the node\_state state variable multicasts OIDs missed to the recovering nodes, this node enters in the alive\_recoverer state. This means that this node is in charge of transferring objects missed to a recovering node. Once it finishes its recovery process, it switches to the alive state. Respectively, the transition from recovering\_recoverer to alive is done once all the missed updates for all views are persistently stored on the RDBMS.
The node state is changed every time a view\_change action is fired by the respective group membership monitor. Equivalent actions depicted in our algorithm are join and crashed, these actions are not directly invoked by the membership monitor. Both are fired each time a view\_change action is fired. The first action is invoked whenever a node (or several at a time) joins the system. This action passes as a parameter the new nodes joining the system on nodes whose previous state was different from crashed, in the joining nodes passes the set of the current reachable nodes respectively. The other action (crashed) is obviously invoked in nodes that are not crashed, it contains the set of nodes that have crashed in this change of view.

- **Current view.** Each view\_change action fired by the group membership monitor fills this variable and the current content of it is passed to the next variable containing the previous view of the system. Initially, i.e. at start-up time, both variables contain the total nodes with the identifier field containing a zero value. The membership monitor passes an index indicating the current view in the system along with the set of nodes that are currently reachable. This information is contained in this variable (V).

\[
\mathcal{V}_i \equiv \begin{cases} \mathcal{V}_i, id \in \{1 \ldots MAX\_VIEW\} \\ \mathcal{V}_i, nodes = \{j : j \in NODES\} \end{cases}
\]

- **Previous view.** During a view\_change action, the old view stored in the current view variable (V) is stored here. Therefore, failed and joining nodes may be easily calculated. This variable, as well as the previous one, are the key information for firing join and crashed actions.

\[
\mathcal{V}_{prev}_i \equiv \begin{cases} \mathcal{V}_{prev}_i, id \in \{1 \ldots MAX\_VIEW\} \\ \mathcal{V}_{prev}_i, nodes = \{j : j \in NODES\} \end{cases}
\]

### 6.7 Concurrency and Recovery Algorithm

\[\text{\textcopyright lock\_request}(o, s, mode): o \in OID, s \in SID, mode \in \{\text{read, write, copy, recover}\} \]

\[\text{pre: (node\_state},[i], status \neq \text{crashed } \lor \text{node\_state},[i].status \neq \text{recovering} ) \land \text{session\_state},[i] = \text{run}.\]

\[\text{eff: } L_T[i].o.waiting \leftarrow + \{ (s, mode) \}; \]

\[\text{session\_state},[i] \leftarrow \text{blocked}; \]

\[\forall \text{sid} \in \text{deadlock}\_prev(o, (s, mode)): \]

\[\forall \text{sid} \in (\text{sid})\_s: \text{abort(sid)}; \]

\[\text{* Optional by reiterated *}
\]

\[\text{if } L_T[i].o.assigned \ni (s, mode) \text{ then } \text{session\_state},[i] \leftarrow \text{run}; \]

\[\{\text{s: s} \in \text{SID} : \text{deadlock}\_prev(o \in OID, (s, mode) \in (\text{SID,} \{\text{read, write, copy, recover}\})) \equiv \]

\[\begin{cases} & \text{if } L_T[i].o.assigned = \emptyset \text{ then } \\
& L_T[i].o.waiting \leftarrow -\{ (s, mode) \}; \\
& L_T[i].o.assigned \leftarrow + \{ (s, mode) \}; \\
& \text{session\_state},[i] \leftarrow \text{run}; \\
& \text{deadlock}\_prev, \leftarrow \emptyset \\
& \text{else} \\
& \text{if } \text{mode} = \text{read} \text{ then } \\
& \text{(* We assume they are sorted by the levels stated in the deadlock prev alg *)} \\
& \forall (\text{sid}, m) \in L_T[i].o.assigned: \\
& \text{if } m = \text{write} \text{ then } \\
& \text{if } \text{session\_state},[\text{sid}] = \text{pre\_commit} \text{ then } \text{deadlock}\_prev, \leftarrow \emptyset \\
& \text{else} \\
& \text{if } s.\text{restarts} > \text{NUM\_RESTARTS} \text{ then } \\
& \text{if } s.\text{restarts} > \text{sid.\text{restarts}} \text{ then } \text{deadlock}\_prev, \leftarrow \text{sid} \\
& \text{else if } s.\text{timestamp} < \text{sid.\text{timestamp}} \text{ then } \text{deadlock}\_prev, \leftarrow \text{sid} \\
& \text{else } \text{deadlock}\_prev, \leftarrow \emptyset \\
& \text{else} \\
& \text{if } s.\text{restarts} > \text{NUM\_RESTARTS} \text{ then } \text{deadlock}\_prev, \leftarrow \emptyset \\
& \text{else if } s.\text{timestamp} < \text{sid.\text{timestamp}} \text{ then } \text{deadlock}\_prev, \leftarrow \text{sid} \\
& \text{else } \text{deadlock}\_prev, \leftarrow \emptyset \\
& \text{else if } m = \text{copy} \text{ then } \text{deadlock}\_prev, \leftarrow \emptyset \\
& \text{else if } m = \text{recover} \text{ then } \\
\end{cases} \]


14
if node.state[i].status = recovering_recoverer then
  if sid.node = i then deadlock.previ ← ∅
else
  if s.restarts > NUM_RESTARTS then
    sids ← ∅;
    ∀ (s2, m2) ∈ L_T_i_0.assigned:
      if m2 = read then
        if (session.state[i] ≠ pre_commit) ∧ ((s2.restarts < s.restarts) ∨
          ((s2.restarts = s.restarts) ∧ (s.timestamp < s2.timestamp)))
        sids ← s2
      if (sids = ∅) ∧ (first(L_T_i_0.waiting) = (s, mode))
        L_T_i_0.waiting ← −{(s, mode)};
        L_T_i_0.assigned ← +{(s, mode)};
        session.state[i] = run;
        deadlock.previ ← ∅
else deadlock.previ ← sids
else /* It is not an elderly session */
  ∀ (s2, m2) ∈ L_T_i_0.assigned:
    if m2 = read then
      if s2 > NUM_RESTARTS then deadlock.previ ← ∅
else
  if s.restarts > NUM_RESTARTS then
    sids ← ∅;
    ∀ (s2, m2) ∈ L_T_i_0.assigned:
      if m2 = read then
        if first(L_T_i_0.waiting) = (s, mode)
          L_T_i_0.waiting ← −{(s, mode)};
          L_T_i_0.assigned ← +{(s, mode)};
          session.state[i] = run;
          deadlock.previ ← ∅
else deadlock.previ ← sids
else /* It is not an elderly session */
  ∀ (s2, m2) ∈ L_T_i_0.assigned:
    if m2 = read then
      if first(L_T_i_0.waiting) = (s, mode)
        L_T_i_0.waiting ← −{(s, mode)};
        L_T_i_0.assigned ← +{(s, mode)};
        session.state[i] = run;
        deadlock.previ ← ∅
else deadlock.previ ← ∅
if (session_state[s2] \neq \text{pre-commit}) \land ((s2\text{restarts} < s\text{restarts}) \lor 
(s2\text{restarts} = s\text{restarts}) \land (s\text{timestamp} < s2\text{timestamp})) then 
sids = s2 
if (sids = \emptyset) \land (\text{first}(L_T[o].waiting) = \langle s, \text{mode} \rangle) then 
L_T[o].waiting = -\{\langle s, \text{mode} \rangle\}; 
L_T[o].assigned = + \{\langle s, \text{mode} \rangle\}; 
session_state[s] = \text{run}; 
deadlock_{prev} \leftarrow \emptyset 
else deadlock_{prev} \leftarrow sids 
else (* It is not an elderly session *) 
\forall \langle s2, m2 \rangle \in L_T[o].assigned: 
if m2 = \text{read} then 
if s2 > \text{NUM}_\text{RESTARTS} then deadlock_{prev} \leftarrow \emptyset 
else if m2 \neq \text{read} then 
if s:restarts > \text{NUM}_\text{RESTARTS} then 
if (s:restarts > sid:restarts) \land session_state[sid] \neq \text{pre-commit} then 
sids = + \{sid\} 
else if sid:restarts > \text{NUM}_\text{RESTARTS} then deadlock_{prev} \leftarrow \emptyset 
else 
if session_state[s] \neq \text{pre-commit} \land (s\text{timestamp} < sid\text{timestamp}) then sids = + \{sid\} 
else deadlock_{prev} \leftarrow sids 
else if m2 = \text{write} then 
if s:restarts > \text{NUM}_\text{RESTARTS} then 
if (s:restarts > sid:restarts) \land session_state[sid] \neq \text{pre-commit} then 
sids = + \{sid\} 
else if sid:restarts > \text{NUM}_\text{RESTARTS} then deadlock_{prev} \leftarrow \emptyset 
else 
if session_state[s] \neq \text{pre-commit} \land (s\text{timestamp} < sid\text{timestamp}) then sids = + \{sid\} 
else deadlock_{prev} \leftarrow sids 
else if m = \text{write} then 
if s:restarts > \text{NUM}_\text{RESTARTS} then 
if session_state[sid] = \text{pre-commit} then deadlock_{prev} \leftarrow \emptyset 
else if s:restarts > sid:restarts then deadlock_{prev} \leftarrow \emptyset 
else 
if session_state[sid] = \text{pre-commit} then deadlock_{prev} \leftarrow \emptyset 
else deadlock_{prev} \leftarrow \emptyset 
else if sid:restarts > \text{NUM}_\text{RESTARTS} then deadlock_{prev} \leftarrow \emptyset 
else 
if session_state[sid] = \text{pre-commit} then deadlock_{prev} \leftarrow \emptyset 
else deadlock_{prev} \leftarrow \emptyset 
else deadlock_{prev} \leftarrow \emptyset (* It is a copy-lock *) 
deadlock_{prev} \leftarrow sids 
else if mode = \text{copy} then 
sids = \emptyset; 
\forall \langle sid, m \rangle \in L_T[o].assigned: 
if m \neq \text{write} then 
(* Sessions willing to commit precedence *) 
(* Assigned sessions are either on the acquisition phase or the pre-commit phase *) 
if session_state[sid] = \text{pre-commit} then
if s.timestamp < sid.timestamp then deadlock.prev ← sid;
else deadlock.prev ← s;
else
    if oids.copied(s) ≠ ∅ then
        if s.timestamp < sid.timestamp then deadlock.prev ← sid
        else deadlock.prev ← s
    else
        if sid.restarts > NUM.RESTARTS then deadlock.prev ← ∅
        else deadlock.prev ← sid
    else if m = copy then
        if s.timestamp < sid.timestamp then deadlock.prev ← sid
        else deadlock.prev ← s
        (* Sessions willing to commit precedence *)
        (* Assigned sessions are either on the acquisition phase or the pre-commit phase *)
    else if m = read then
        if session.state[sid] = pre_commit then
            if s.timestamp < sid.timestamp then sids ← s + {sid};
        else deadlock.prev ← s
        else deadlock.prev ← ∅
    else (* It is a recover-lock *) deadlock.prev ← ∅
    else (* It is a recover-lock request *)
        sids ← ∅;
∀ (sid, m) ∈ L_T[i].assigned:
    if (m = write) ∨ (m = copy) then deadlock.prev ← sid
    else if m = read then
        if (node.state[i].status = recovering_recoverer ∧ s.node = i) then
            sids ← s + {sid}
        else
            L_T[i].waiting ← L_T[i].waiting − {(s, mode)};
            L_T[i].assigned ← L_T[i].assigned + {(s, mode)};
            session.state[i][s] ← run;
            deadlock.prev ← ∅
        else
            L_T[i].waiting ← L_T[i].waiting − {(s, mode)};
            L_T[i].assigned ← L_T[i].assigned + {(s, mode)};
            session.state[i][s] ← run;
            deadlock.prev ← ∅
    else deadlock.prev ← sids

(* This action is invoked by the final user or the application *)

◊ user.abort(s); s ∈ SID
pre: node.state[i] ≠ crashed ∧ session.state[i][s] = run.

eff: session.state[i][s] ← abort; local.abort(s)

◊ abort(s); s ∈ SID
pre: node.state[i] ≠ crashed ∧ session.state[i][s] ∈ {run, blocked, pre_commit}.

eff: if s.node = i then
    if session.state[i] = pre_commit then multicast(V, nodes, msg_abort(s))
else session.state[i] ← abort; local.abort(s)
else (* s.node = i *) send(s.node, msg_remote_abort(s));
\[\text{local\_abort}(s) \equiv \text{db\_rollback}(s); \text{release\_locks}(s);\]

\[\forall o \in \text{oids\_read}(s) : \]
\[L[T_i][o].\text{assigned} \leftarrow \{-\{s, \text{read}\}\};\]
\[\text{if } (L[T_i][o].\text{assigned} = \emptyset) \land (L[T_i][o].\text{waiting} \neq \emptyset) \text{ then}\]
\[\forall \langle \text{sid, mode} \rangle \in L[T_i][o].\text{waiting} :\]
\[\text{if compatible}(o, \langle \text{sid, mode} \rangle) \text{ then}\]
\[L[T_i][o].\text{waiting} \leftarrow \{-\{\text{sid, mode}\}\};\]
\[L[T_i][o].\text{assigned} \leftarrow + + \{\{\text{sid, mode}\}\};\]
\[\text{session\_state}_{i}[\text{sid}] \leftarrow \text{run};\]
\[\text{else break};\]
\[\forall o \in \text{oids\_written}(s) : \]
\[L[T_i][o].\text{assigned} \leftarrow \{-\{s, \text{write}\}\};\]
\[\text{if } L[T_i][o].\text{waiting} \neq \emptyset \text{ then}\]
\[\forall \langle \text{sid, mode} \rangle \in L[T_i][o].\text{waiting} :\]
\[\text{if compatible}(o, \langle \text{sid, mode} \rangle) \text{ then}\]
\[L[T_i][o].\text{waiting} \leftarrow \{-\{\text{sid, mode}\}\};\]
\[L[T_i][o].\text{assigned} \leftarrow + + \{\{\text{sid, mode}\}\};\]
\[\text{session\_state}_{i}[\text{sid}] \leftarrow \text{run};\]
\[\text{else break};\]
\[\forall o \in \text{oids\_copied}(s) : \]
\[L[T_i][o].\text{assigned} \leftarrow \{-\{s, \text{copy}\}\};\]
\[\text{if } L[T_i][o].\text{waiting} \neq \emptyset \text{ then}\]
\[\forall \langle \text{sid, mode} \rangle \in L[T_i][o].\text{waiting} :\]
\[\text{if compatible}(o, \langle \text{sid, mode} \rangle) \text{ then}\]
\[L[T_i][o].\text{waiting} \leftarrow \{-\{\text{sid, mode}\}\};\]
\[L[T_i][o].\text{assigned} \leftarrow + + \{\{\text{sid, mode}\}\};\]
\[\text{session\_state}_{i}[\text{sid}] \leftarrow \text{run};\]
\[\text{else break};\]
\[\forall o \in \text{oids\_recovered}(s) : \]
\[L[T_i][o].\text{assigned} \leftarrow \{-\{s, \text{recover}\}\};\]
\[\text{if } (L[T_i][o].\text{assigned} = \emptyset) \land (L[T_i][o].\text{waiting} \neq \emptyset) \text{ then}\]
\[\forall \langle \text{sid, mode} \rangle \in L[T_i][o].\text{waiting} :\]
\[\text{if compatible}(o, \langle \text{sid, mode} \rangle) \text{ then}\]
\[L[T_i][o].\text{waiting} \leftarrow \{-\{\text{sid, mode}\}\};\]
\[L[T_i][o].\text{assigned} \leftarrow + + \{\{\text{sid, mode}\}\};\]
\[\text{session\_state}_{i}[\text{sid}] \leftarrow \text{run};\]
\[\text{else break};\]

\[\text{boolean: compatible}(o, \langle \text{sid, mode} \rangle, \langle \text{sid, copy, recover} \rangle) \equiv \]
\[\text{if } L[T_i][o].\text{assigned} = \emptyset \text{ then compatible } \leftarrow \text{true}\]
\[\text{else}\]
\[\text{if mode } = \text{ read then}\]
\[\forall \langle \text{sid, m} \rangle \in L[T][o].\text{assigned}:\]
\[\text{if m } = \text{ read then compatible } \leftarrow \text{true}\]
\[\text{else if } \text{m } = \text{ write } \lor \text{m } = \text{ copy then compatible } \leftarrow \text{false}\]
\[\text{else if } \text{m } = \text{ recover } \land \text{node\_state}_{i}[\text{sid}] = \text{recovery\_recovee}r \land \text{sid\_node } = \text{i then compatible } \leftarrow \text{false}\]
\[(* \text{If it enters here it is because it is a recover-lock of another recovering node } *)\]
\[\text{else compatible } \leftarrow \text{true}\]
\[\text{if } (\text{mode } = \text{ write}) \lor (\text{mode } = \text{ copy}) \text{ then compatible } \leftarrow \text{false}\]
\[\text{if mode } = \text{ recover then}\]
\[\forall \langle \text{sid, m} \rangle \in L[T][o].\text{assigned}:\]
\[\text{if m } = \text{ read then}\]
\[\text{if node\_state}_{i}[\text{sid}] = \text{recovery\_recovee}r \land \text{sid\_node } = \text{i then compatible } \leftarrow \text{false}\]
\[(* \text{It never enters here } *)\]
else compatible ← true

(* It never occurs that a copy-lock or a write-lock were assigned *)
else if (m = write) ∨ (m = copy) then compatible ← false

(* Nevertheless, it may happen that a recover-lock was already assigned *)
else compatible ← true

\(\triangleright\) deliver_msg_remote_abort(s): s ∈ SID

pre: session_state, [s] ≠ abort.
eff: if (session_state, [i] ≠ abort) ∧ (s.node = i) then

  session_state, [s] ← abort;

  multicast(\(\forall\), nodes, msg_abort(s));

\(\triangleright\) deliver_msg_abort(s): s ∈ SID

pre: session_state, [s] ≠ abort.
eff: if (session_state, [i] ≠ abort) ∧ (s.node ≠ i) then

  session_state, [s] ← abort;

  local_abort(s);

\(\triangleright\) pre_commit(s): s ∈ SID

pre: session_state, [s] = run ∧ s.node = i ∧ node_state, [i].status ≠ recovering.
eff: if (oids_written = \(\emptyset\)) then (* Read-only session *)

  session_state, [s] ← commit;

  local_commit(s)

  else

  session_state, ← pre_commit;

  wait_response, [s] ← \(\forall\), nodes;

  multicast(\(\forall\), nodes, msg_commit(s));

  else

  session_state, ← pre_commit;

  wait_response, [s] ← \(\forall\), nodes;

  multicast(\(\forall\), nodes, msg_commit(s));

  \(\triangleright\) local_commit(s ∈ SID) ≡ db_commit(s) ; release_locks(s);

\(\triangleright\) deliver_msg_remote_update(s, oids, update): s ∈ SID, oids ∈ P(OID), update ∈ UPDATE

pre: s.node ≠ i ∧ session_state, [s] = \(\bot\) ∧ node_state, [i].status ≠ crashed.
eff: update, [s] ← update;

  new_remote_session(s, oids);

\(\triangleright\) deliver_msg_ready(j, s): j ∈ NODES, s ∈ SID

pre: i ≠ j ∧ session_state, [s] = pre_commit ∧ s.node = i.
eff: wait_response, [s] ← {j};

  if wait_response, [s] = \(\emptyset\) then

    multicast(\(\forall\), nodes, msg_commit(s));

\(\triangleright\) deliver_msg_commit(s): s ∈ SID

pre: node_state, [s] ≠ crashed ∧ (session_state, [s] = pre_commit ∨ session_state, [s] = run).
eff: if s.node ≠ i then

  apply_updates(update, [s]);

  if \(\forall\), nodes ≠ NODES then

    objects_for_recover, [\(\forall\), id].oids ← \(\cup\) oids_written(s);

    local_commit(s);

  session_state, [s] ← commit;

\(\triangleright\) crashed, (id, nodes): id = \(\forall\), id, nodes ⊆ NODES

pre: node_state, [i] ≠ crashed.
eff: objects_for_recover, [id] ← (\(\emptyset\), nodes)

  \(\forall\) k ∈ nodes:

    if (node_state, [k].status = recovering) then

      node_state, [k].status ← crashed;

    if (node_state, [k].status = recovering) then

      \(\forall\) s ∈ recover,_session, [k];

19
local_abort(s);
    session_state[s] ← abort;
    node_state[i].status ← crashed;
else
    ∀ s ∈ sessions,
        if s.node = k then
            local_abort(s);
            session_state[s] ← abort
        else
            if session_state[s] = pre_commit then
                wait_response[s] ← \{k\};
                (* We check that everyone has answered *)
                if wait_response = ∅ then
                    multicast(V, nodes, msg_commit(s));
                    node_state[i].status ← crashed;
                if (node_state[i].status = alive_recoverer) then
                    if oldest_alive() then
                        multicast(V, nodes, msg_new_recoverer(i, node_state[i].join));
                    else
                        ∀ k ∈ NOD:
                            if recov_sessions[k] ≠ ∅ then
                                ∀ s ∈ recov_sessions[k]; (* Recov session is (node, view_id) *)
                                oids ← objects_to_recover[s, view_id];
                                send(s.node, msg_objs_state(i, s.view_id, (oids, state(oids))));
                        end
                    N D:
                        ∧ join(id, nodes); id = V, id, nodes ⊂ NOD
                        pre: node_state[i] ≠ ⊥.
                        eff: if (V, nodes ∩ NODS ≠ NODS) then (* There are failed nodes yet *)
                            objects_to_recover ← ++ {id, (∅, (NODS \ nodes))};
                        ∀ k ∈ nodes: node_state[i] ← (recovering, id);
                        (* The only one that knows exactly which views are going to be recovered is the oldest one *)
                        if oldest_alive() then
                            multicast(V, nodes, msg_start_recovery(k, i, node_state[i].join, views_missed(k));
                    else
                        ∀ k ∈ NOD \ {i}: node_state[i] ← (⊥, id - 1);
                        node_state[i] ← (recovering, id);
                        ∧ boolean: oldest_alive() ≡
                        if (node_state[i].status = recovering) ∨ (node_state[i].status = recovering_recoverer) then oldest_alive ← false
                        else (* Node i is alive, we search the oldest among all alive nodes in order to see if it is itself *)
                            ∀ j ∈ NOD \ {i};
                            if ((node_state[j].status = alive) ∨ (node_state[j].status = alive_recoverer))
                                ∧ (node_state[i].join > (node_state[j].join then
                                    min ← j;
                            if (min ≠ i) then oldest_alive ← false
                            else oldest_alive ← true
                        ∧ {v, oids, nids}; views_missed(j ∈ NOD) ≡
∀ counter ∈ \{1, \ldots, MAX\_VIEW\}:
  \begin{align*}
    &\text{if } j \in \text{objects}_\text{to}_\text{recover}[^{\text{counter}}].\text{nodes} \text{ then} \\nonumber \\
    &\quad \text{views}_\text{oids}_\text{nodes} \leftarrow \text{counter}, \text{objects}_\text{to}_\text{recover}[^{\text{counter}}].\text{oids}, \text{objects}_\text{to}_\text{recover}[^{\text{counter}}].\text{nodes}; \\nonumber \\
    &\quad \text{views}_\text{missed} \leftarrow \text{views}_\text{oids}_\text{nodes}; \\nonumber \\
    &\text{end if} \\nonumber \\
    &\text{end if} \nonumber \\
  \end{align*}

\text{deliver}_\text{msg}_\text{start}_\text{recovery}((k, j), \text{joined}, \text{views}_\text{oids}_\text{nodes}); k, j \in \text{NODES}, \text{joined} ∈ \{1, \ldots, MAX\_VIEW\}, \text{views}_\text{oids} ∈ \{\{v, \text{oids}, \text{nids} \}: v ∈ \{1, \ldots, MAX\_VIEW\}, \text{oids} ⊆ P(\text{OID}), \text{nids} ⊆ P(\text{NID})\}

\text{pre: node}_\text{state}[i] \neq \text{crashed.}

\text{eff: node}_\text{state}[j] \leftarrow (\text{alive}_\text{recoverer}, \text{joined});
\text{node}_\text{state}[i].\text{status} \leftarrow \text{recovering}_\text{recoverer};
\text{(* Nodes being recovered do not know whether there are older recovering sessions *)}
\text{(* Alive nodes do not know if this pending recovery session exists or not *)}
∀ (v, \text{oids}, \text{nids}) ∈ \text{views}_\text{oids}:
  \text{(* These missed views may have not being stored in this node *)}
  \begin{align*}
    &\text{if } \text{objects}_\text{to}_\text{recover}[^{\text{v}}] = \bot \text{ then} \\nonumber \\
    &\quad \text{objects}_\text{to}_\text{recover}[^{\text{v}}] \leftarrow + + \{\{v, (\text{oids}, \text{nids})\}\} \\nonumber \\
    &\text{else} \\nonumber \\
    &\quad \text{objects}_\text{to}_\text{recover}[^{\text{v}}].\text{oids} ← \text{oids}; \\nonumber \\
    &\quad \text{objects}_\text{to}_\text{recover}[^{\text{v}}].\text{nodes} ← \text{oids}; \\nonumber \\
    &\quad \text{recovery}_\text{session}[k] ← + + \{v\}; \\nonumber \\
    &\quad \text{new}_\text{recovery}_\text{session}(\text{recovery}_\text{session}[k], v, \text{oids}); \\nonumber \\
    &\text{if } \text{oldest}_\text{alive}(i) \text{ then} \\nonumber \\
    &\quad \text{send}(k, \text{msg}_\text{obj}_\text{state}(i, v, (\text{oids}, \text{state}(\text{oids})))); \\nonumber \\
  \end{align*}

\text{deliver}_\text{msg}_\text{obj}_\text{state}(j, v, \text{oids}_\text{state}); j ∈ \text{NODES}, v ∈ \{1, \ldots, MAX\_VIEW\}
\text{oids}_\text{state} ∈ \{(o, \text{state}) : o ∈ \text{OID}, \text{state} ∈ \text{STATE}\}
\text{pre: node}_\text{state}[i].\text{status} ← \text{recovering}_\text{recoverer} \land \text{node}_\text{state}[i].\text{status} ← \text{alive}_\text{recoverer} \land v ∈ \text{recovery}_\text{session}[i].
\text{eff: } \forall (o, s) ∈ \text{oids}_\text{state} \text{: apply}_\text{recover}(v, o, s);
\text{multicast}(V, \text{nodes}, \text{msg}_\text{view}_\text{recovered}(v, i, j));
\text{apply}_\text{db}(o, s);

\text{apply}_\text{recovery}(v ∈ \{1, \ldots, MAX\_VIEW\}, o ∈ \text{OID}, s ∈ \text{STATES}) ≡
\text{while } (\text{recovery}_\text{session}[i], v, o, s) \notin \text{LT}[o].\text{assigned} \text{ do wait();}
\text{apply}_\text{db}(o, s);

\text{deliver}_\text{msg}_\text{view}_\text{recovered}(\text{recovered}_\text{view}, \text{recovered}_\text{id}, \text{recovered}_\text{join}); \text{recovered}_\text{view} ∈ \{1, \ldots, MAX\_VIEW\}, \text{recovered}_\text{id} \land \text{recovered}_\text{id} ∈ \text{NODES}
\text{pre: node}_\text{state}[i].\text{status} \neq \text{crashed.}
\text{eff: objects}_\text{to}_\text{recover}[\text{recovered}_\text{view}].\text{nodes} ← \text{recovered}_\text{id};
\text{recovery}_\text{session}[\text{recovered}_\text{id}] ← \{\text{recovered}_\text{view}\};
\text{local}_\text{commit}(\text{recovery}_\text{session}[\text{recovered}_\text{id}, \text{recovered}_\text{view}]));
\text{session}_\text{state}[\text{recovered}_\text{id}, \text{recovered}_\text{view}] ← \text{commit};
\text{release}_\text{locks}(\text{recovery}_\text{session}[\text{recovered}_\text{id}, \text{recovered}_\text{view}]);
\text{if } \text{node}_\text{state}[\text{recovered}_\text{id}].\text{state} = \bot \text{ then}
\text{node}_\text{state}[\text{recovered}_\text{id}] ← (\text{recovering}_\text{recoverer}, \text{recovered}_\text{view});
\text{(* Nodes that have not received their start\textunderscore recovery message *)}
\text{if node}_\text{state}[\text{recovered}_\text{id}].\text{status} = \bot \text{ then}
\text{node}_\text{state}[\text{recovered}_\text{id}] ← (\text{alive}_\text{recoverer}, \text{recovered}_\text{view});
\text{if } (i = \text{recovered}_\text{id}) \land (\text{recovery}_\text{session}[\text{recovered}_\text{id}] = \emptyset) \text{ then}
\text{node}_\text{state}[i] ← (\text{alive}_\text{recoverer}, \text{recovered}_\text{view});
\text{multicast}(V, \text{nodes}, \text{msg}_\text{alive}(\text{recovered}_\text{id}, \text{recovered}_\text{view}));
\text{else if node}_\text{state}[i].\text{status} = \text{alive}_\text{recoverer} \text{ then}
\text{recovery}_\text{finished} ← \text{true};
\forall k ∈ \text{NODES}:
\text{if node}_\text{state}[k].\text{status} = \text{alive}_\text{recoverer} \text{ then}
\text{node}_\text{state}[k].\text{status} ← \text{false};
\text{if node}_\text{state}[i].\text{status} = \text{false} \text{ then}
\text{multicast}(V, \text{nodes}, \text{msg}_\text{recovery}_\text{finished}(i, \text{node}_\text{state}[i].\text{joined});
\text{device}_\text{msg}_\text{alive}(\text{recovered}_\text{id}, \text{recovered}_\text{join}); \text{recovered}_\text{id} ∈ \text{NID}, \text{recovered}_\text{join} ∈ \{1, \ldots, MAX\_VIEW\}
6.8 Remote Session

A remote session is started in a given node by a session reaching the pre\textunderscore commit state at a different site. The session’s master site multicasts changes done to the given object repository, i.e. the transaction update and the OIDs of updated objects. Reception of this message at all alive nodes will fire this session at all nodes. This session will only request copy\textunderscore locks, therefore it may be aborted during its life due to deadlock prevention reasons.

\* new_remote\textunderscore session\(_i\)(s \in \text{SID}, oids \in \{o : o \in \text{OID}\}) \equiv
\forall o \in \text{oids}:
  \begin{align*}
  &\text{if } (\text{session\textunderscore state}_i[s] \neq \text{abort}) \text{ then } \text{lock\textunderscore request}(o, (s, \text{copy})); \\
  &\text{else break}; \\
  &\text{if } (\text{session\textunderscore state}_i[s] = \text{run}) \text{ then } \text{send}(s, \text{node}, \text{msg\textunderscore ready}(i, s));
  \end{align*}

6.9 Recovery Session

This session is started once a message sent by the oldest alive node is received at all alive nodes. This special session is very similar to a remote session. It performs successive invocations to the lock\textunderscore request action.

\* new_recover\textunderscore session\(_i\)(s \in \text{SID}, oids \in \{o : o \in \text{OID}\}) \equiv
\begin{align*}
  &\text{session\textunderscore state}_i[s] \leftarrow \text{run}; \\
  &\forall o \in \text{oids}: \text{lock\textunderscore request}(o, (s, \text{recover}));
\end{align*}

A Correctness Proofs

A.1 Deadlock Prevention

Theorem 1. This protocol is deadlock free.

Proof. According to [26] a deadlock may only happen if four necessary conditions hold in the system: mutual exclusion, wait for, no preemption, and circular wait. In our algorithm, the circular wait condition is easily broken, since we have set four priority levels for the sessions, and in each level we also distinguish a total order based on the timestamp\textunderscore NID pair assigned to the session. As a result, the wait for condition only arises if a session with a low priority level tries to access an item previously assigned to a session with higher priority level. This prevents a circular wait, since the first session of a chain of dependencies must have a lower priority level than the last one in the same chain. This last one cannot close the circular wait, because it will force the abortion of the first one if it requests the item assigned to it.

A.2 Liveness

Theorem 2. The protocol complies with liveness.

Proof. We show that each lock of a session \(S_j\) will eventually be granted. We have four different priority levels associated to sessions: recovery, elderly, remote, and local sessions. For each one of these sessions we have to look for possible deadlock situations.
- $S_j$ is a recovery session. This is the most important session, thus all locks needed by it will be granted, the rest of sessions owning those objects will be aborted. As a result, there is no problem with the completion of this kind of sessions.

- $S_j$ is an “elderly” session. The existence of recovery sessions trying to recover objects contained in the set of objects modified by $S_j$ will abort it or put to wait the elderly session. These recovery sessions may need some objects present in the set of objects accessed by $S_j$, if some other active session has updated them whilst $S_j$ was a regular session (i.e., it was not aborted the needed number of times to become an “elderly” session). This set of objects has to be transferred to the faulty nodes that now are in the recovering phase, but all these objects need to be transferred only once to each recovering node. In the worst case, remember that we are in a primary partition system, there will be at most $\frac{n}{2} - 1$ failed nodes, if they fail and recover following a round-robin policy. The elderly session has to wait at most for one round of failed nodes; i.e., once this round of failures and recoveries is completed, no recovering node will need to lock the objects accessed by $S_j$.

This guarantees that this “elderly” session can be committed once this round of recoveries terminates. In this situation, it will be the oldest alive session, no local or remote session will abort it. If a remote session is currently holding a copy-lock on an object, $S_j$ will be put to sleep.

- $S_j$ is a remote session. As in the latter case, all recovery sessions will abort or put to sleep this sort of sessions. If an elderly session is already holding a conflicting lock with $S_j$ (and $S_j$ is currently holding any lock) then the timestamp of both will be compared so as to abort one of them. If $S_j$ has no lock then it will be put to sleep, therefore they give a chance to elderly sessions to reach the pre-commit state so liveness is guaranteed.

- $S_j$ is a local or remote session. Sessions in this priority level can be aborted by sessions with higher priority level, but if this happens a given number of times, they will be promoted to the “elderly” level, and, in this new level, their successful completion is guaranteed.

A.3 Repeatable-Read Isolation Level

Theorem 3. This protocol provides a repeatable-read isolation level.

Proof. We base our proof on the anomalies that have to be avoided to achieve a given isolation level, described in [3]. In that work, four anomalies are described for the standard ANSI SQL isolation levels. They are the following:

- **Dirty writes.** In this anomaly, if a transaction $T_1$ writes on a given data item, and later another transaction $T_2$ also updates the same item before $T_1$ commits or aborts, then the updates of $T_2$ overwrite those of $T_1$. Moreover, if $T_1$ or $T_2$ aborts, then it is unclear which value of the updated item has to be restored.

Our algorithm prevents this anomaly since it uses write locks before updating any data item, and these locks are maintained until the transaction completes (either with a commit or rollback action).

- **Dirty reads.** This behavior corresponds to a transaction $T_1$ updating a data item and $T_2$ performing a concurrent read of it, hence $T_2$ has read an inconsistent value of the data item (this is the classical inconsistent analysis).

As this algorithm works with locks, read-lock and write-locks, both are not compatible and are held until transaction completion. Thus, this situation is avoided.

- **Non repeatable reads** (fuzzy reads). Transaction $T_1$ reads a data item. Another transaction $T_2$ then modifies or deletes that data item and commits. If $T_1$ then attempts to reread the data item, it receives a modified value or discovers that the data item has been deleted.

This anomaly is prevented using the same mechanisms described in the previous one, dealing with dirty reads.
- **Phantom reads.** Transaction \( T_1 \) reads a set of data items satisfying some search condition SC. Transaction \( T_2 \) the creates data items that satisfy \( T_1 \)'s SC and commits. If \( T_1 \) the repeats its read with the same SC, it gets a set of data items different from the first read.

The algorithm, as described in this document, does not avoid this anomaly. However, in the following section we outline the improvements needed to guarantee that phantom reads would not appear in our system.

In [3], the four ANSI isolation levels are mapped to these anomalies in the way outlined in Table 2. As we can see, the current anomalies avoided by our algorithm correspond to a **REPEATABLE READ** isolation level. We only need to ensure that phantom reads would not appear to guarantee a 1-copy-serializable behaviour.

<table>
<thead>
<tr>
<th>Isolation level</th>
<th>Anomalies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>dirty write</td>
</tr>
<tr>
<td>READ UNCOMMITTED</td>
<td>no</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>no</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>no</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 2: ANSI SQL isolation levels defined in terms of the four anomalies.

### A.3.1 Extensions Needed to Comply with 1-Copy-Serializability

In order to avoid the **phantom reads** anomaly, the algorithm has to be extended, managing locks on “search conditions”; i.e., maintaining the set of fields accessed in a given condition and the values that make such condition true. Hence, future update operations that do collide with these search conditions are not allowed to proceed. The class of **update operations** includes updates, inserts and removals.

### A.4 Repeatable-Read Recovery

**Theorem 4.** Upon successful completion of the recovery procedure, a recovery site reflects a state compatible with the actual repeatable-read execution that took place.

**Proof.** This property can be ensured by the recover-locks. None of the available nodes may update the values of the objects to be recovered. In our case, local transactions are allowed both in the source and recovery nodes, but the recover-locks ensure that the source objects are not updated, whilst in the recovering nodes such objects are neither read nor updated.

If the extensions outlined in Section A.3.1 have been included in our system, the recovery also complies with the 1-copy-serializability property.

### References


