Lazy Recovery in a Hybrid Database Replication Protocol *

Luis Irún-Briz, Francisco Castro-Company, Félix García-Neiva, Antonio Calero-Monteagudo, and Francesc D. Muñoz-Escoí

Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
46022 Valencia, SPAIN
Email: {lirun,fcastro,fgarcia,acalero,fmunyoz}@iti.upv.es

Abstract. COLUP is a hybrid database replication protocol that may be configured to behave either as an eager or a lazy update protocol. When a faulty replica joins again the system, a recovery subprotocol is needed to update its database state. Independently of the chosen update strategy, these recovery tasks are lazy and introduce a minimal overhead in the non-recovering replicas.
This recovery protocol is based on the roles carried out by each database replica when an object is considered. There are several roles: owner, synchronous, and asynchronous. The owner is unique and is the object creator. It manages the access control requests on its owned objects. Synchronous replicas are those that receive updates before transactions are committed. Finally, asynchronous replicas only receive lazy updates and may have a stale state. Thus, the recovering replica requests an up-to-date version of its owned objects once a local or remote transaction has made an access control request on them, but will rely on the common update propagation approach for all its non-owned objects.

1 Introduction

Fault-Recovery has been presented in the literature as the process needed by the system to stabilize the normal behavior when some node has suffered any fault in its functionality, or when a faulty node recovers from its abnormal state, and is re-included in the system.

In a fault-tolerant system the recovery is usually performed with specific (and often costly) algorithms designed to reestablish the normal functionality of the system when a fault is detected, or when a faulty node recovers and is re-included in the system.

Moreover, the specific algorithms mentioned above must be run at least when a *system reconfiguration*[12] arises. This is due to two facts: first, a fault-tolerant system is based on replication, and the different replicas must be adjusted to have an adequate behavior; second, a distributed system maintains a distributed state, that must also be treated in presence of system reconfigurations.

In a replicated database system the most costly management is related to ensuring consistency[9]. A node recovery is a worst case in the system reconfigurations, because

* This work has been partially supported by the Spanish MCYT grant TIC2003-09420-C02-01.

the information managed by the recovering node may be incomplete, outdated and even inconsistent with the information stored in the rest of replicas. Thus, the information managed by the recovering node must be synchronized using the data held in the rest of database nodes. Some recovery algorithms need that, once a node has failed, the surviving nodes keep track of some kind of activity log. This log maintenance makes the solution unscalable, due to the unpredictable log lengths.

Regardless the origin of the information needed to update the recovered node, a huge amount of information may be transmitted, and the consistency may be compromised if the process is not carefully achieved. Thus, in some algorithms the recovering node is not able to initiate any transaction until the information held in its database is updated. In addition, the rest of the system nodes should take into account that the recovering node is not completely updated, and perform additional actions to guarantee the consistency of their locally initiated transactions.

On the other hand, the recovery of a node can be accomplished in a more "graceful" way. Making use of laziness, a recovering node may consider that the information held in its database is asynchronously maintained, and the entire system can continue its normal functionality without the need of any additional actions. In addition, the recovering node can reestablish asynchronously the original state of the accessed objects when a locally initiated transaction requests such objects.

This approach can be named *Lazy Recovery*, because the re-included node performs such recovery in a lazy manner, updating its state from the rest of the system with a sequence of asynchronous operations. Lazy Recovery reduces the overhead of the recovery process, and allows any node in the system to proceed normally, even before the recovering node has completely updated its state.

In the GlobData project[4,13], a software platform is used to support database replication. This platform, is called COPLA, and provides an object-oriented view of a network of relational DBMSs. We use the COPLA architecture as a platform to experiment with different consistency algorithms working on top of a replicated database.

We have designed a hybrid approach to manage the consistency in such systems. Our proposal consists of a new algorithm that is flexible enough to be configured either as a pure eager protocol, and as a pure lazy protocol. In addition, the algorithm can be configured to bring a hybrid, improved behavior, providing a configured degree of asynchrony to the underlying transactional environment, with a certain level of replication along the system. On top of this algorithm, a fault-tolerance mechanism has been developed, providing self-recovery techniques that make use of a lazy approach to complete the node re-inclusion.

In this paper we describe our recovery algorithm, as a modification of the hybrid consistency protocol already described in [6,5]. In addition, measurements of a real implementation are also included, showing the low overhead our approach introduces in a recovering system.

The rest of the paper is structured as follows. Section 2 outlines the basic hybrid COLU protocol. Section 3 includes the modification introduced in the basic algorithm to provide self-recovery ability to the consistency control. An exhaustive study of our proposal is shown in section 4, where an implemented system is detailed, and the ob-

tained results are described. Finally, section 5 outlines the related work and section 6 gives some conclusions.

2 The Hybrid Optimistic Protocol

The replication protocol outlined in this section has been implemented as a consistency protocol in the GlobData Project. This hybrid optimistic protocol implemented in GlobData is named "*Cautious Optimistic Lazy Update Protocol*" (COLUP), and makes use of the algorithm widely detailed in [6].

The algorithm uses the concept of *node role*, giving special relevance to the node where a particular object is created (i.e. the *owner node* of such object).

During the consensus process performed at commit time of a transaction, the owner of an object will be asked to allow this transaction to complete the commit. Thus, it is the manager for the object accesses, and it is also the responsible to coordinate the propagation of the last versions of the object.

Additionally, the identifier of the node where an object was created; i.e., its owner node ID, is included in the object ID. Thus, it will be easy to find which node inherits the ownership in case of failure.

For an object, a set of nodes will participate in the algorithm maintaining a replica of such particular object in a synchronous way. These *synchronous nodes* receive any update performed on the object before the updating transaction terminates in the corresponding active node (i.e. the node where a transaction is initiated). The minimal synchronous set is composed by the owner node of an object.

Finally, all other nodes replicating the object constitute the *asynchronous set of nodes* of such object. In these nodes, the transaction updates regarding this object will be eventually received.

2.1 Algorithm Outline

The COLUP consistency protocol multicasts object updates to the asynchronous nodes beyond the transaction completes its commit phase. Consistency conflicts among transactions are resolved with an optimistic approach, using object versions and checking them during the commit phase. Thus, object accesses are allowed along the transaction execution without any locking treatment.

The main inconvenience of this Lazy Update Propagation is the increase of the abortion rate the use of Laziness introduces in the system. An expression for the probability for an object to cause the abortion of a transaction (i.e. to be a stale-access) was detailed in [6]. The COLU protocol makes use of this expression to predict the convenience for a transaction to ensure that an object asynchronously updated has a recent version in the local database.

In order to apply these results, it becomes necessary to establish a threshold of $PA(o_i)$ (i.e., the abort probability of an access to o_i due to a stale access) to consider the object "convenient to be updated". An adequate value for this threshold should minimize the number of abortions caused by accesses to outdated objects, and keeping low the number of updates for the system. Thus, when a transaction requests an object

access, $PA(o_i)$ is calculated, and compared with the stablished threshold. As a result, the algorithm obtains an updated version for objects predicted to be stale (i.e. out of date).

The implementation of this principle introduces a new request in the protocol. Now, the active node for a transaction will send "Update requests" to the owners of the stale accessed objects, in order to get their updated versions. This update request message is sent along the transaction execution, in order to maintain updated (in a certain degree) the objects accessed by the transaction.

The minimization of the number of updates (obtained with higher thresholds), will increase the number of transactions executed in the system per second, because it will decrease the resources used by the update propagation. On the other hand, this minimization will cause an increase in the number of aborted transactions, because the number of outdated objects will also be increased. The increase of the abortion rate can also degrade the productivity of the system, because the time spent in transactions that are finally aborted is futile. An adaptative algorithm to adjust in run-time the threshold in order to obtain an optimum behavior has been presented in [6].

2.2 A Hybrid Approach

One of the most interesting characteristics of the COLUP approach is that the protocol can be parameterized to have the behavior of either an eager or lazy update protocol.

Extending the set of synchronous replicas, for each object, to the entire system, we will obtain an update propagation protocol which ensures that, for each node in the system, all the updates are propagated within the commit phase, and no other updates are needed anymore. This is precisely the definition of an eager update propagation protocol.

On the other hand, if no update is forced along the life of a transaction (i.e. it has been established threshold to 1.0), and the synchronous set for each object is restricted to the owner node, no update will be performed in the commit phase. In addition, these updates will only be performed when a transaction is aborted due to any access to outdated objects. This is the definition of a pure lazy update propagation protocol.

Intermediate configurations can be established to provide hybrid behavior to the transactional system. Thus, availability of the information can be improved by increasing the size of the synchronous set, and the common advantages of replication, such as scalability and fault-tolerance, will be consequently obtained. In contrast, to relax the synchrony guarantees provided by the system, the threshold should be decreased. The lower the threshold is, the more asynchronously the replicas will be managed.

Desired Behavior	**|S|**	**T**
Pure Eager	N (the entire system)	-
Pure Lazy	1 (only the owner)	1.0
Improved Asynchronous	$1 + d_R \times (N - 1)$ (the owner, plus a set)	d_S

Table 1. Configurations of the Hybrid Algorithm

The different data expressed in the table correspond with the following:

- $|S|$, the size of the Synchronous set for each object in the system.
- T, the Threshold of the algorithm.
- N, the number of nodes in the system.
- d_R, the degree of replication ($d_R \in [0 \ldots 1]$) to be set in the system.
- d_S, level of synchrony ($d_S \in [0 \ldots 1]$) to be set in the system.

The table summarizes the capability of the presented algorithm to provide the versatility enough to cover a wide range of solutions, from a pure Eager approach (with synchronous management of the replication), to a pure Lazy approach (managing asynchronously the replicated data), including a hybrid approach, that allows certain degree of synchrony. Finally, the algorithm also provides an improved asynchronous model, that relaxes the inconveniences of the use of asynchronous techniques.

3 Providing Lazy Self-Recovery Ability to COLUP

In this section, the recovery ability is included in the basic algorithm as a modification in some steps of the COLU protocol, outlined in the previous section.

This modification makes a number of considerations about the underlying network connectivity, and other issues, that must be clarified before further explanations are introduced.

- As occurred in the basic protocol, each node in the system is labeled with a number, identifying it, and providing an order between every node in the system.
- Each node runs a copy of a *membership monitor*. This monitor is a piece of software that observes a preconfigured set of nodes, and notifies its local node about any change in this set (either additions or eliminations). The membership monitor used for the self-recoverable COLU Protocol is described in [10], but has been extended to support a *primary partition* model; i.e., in case of a network partition, only the subgroup with a majority of the preconfigured members can continue, if any.
- The communication protocols guarantee a uniform delivery of the network broadcasts.

The rest of this section shows the differences between the COLU and the "self-recoverable" COLU protocols.

1. When the membership monitor notices a node failure (let N_f be the failed node), a notification is provided to every surviving node in the system. This notification causes for each receiving node to update a list of *alive nodes*. The effect of these notifications will be a logical migration of the ownerships of the failed node. Further steps will explain the term *logical*.
2. During the execution of a transaction, a number of messages can be sent to the different owners of the objects accessed by this transaction. If a message must be sent to a failed owner N_f, then it will be redirected to the new owner for the involved object. This new owner can be assigned in a deterministic way from the set

of synchronous replicas of the object (e.g. electing as new owner the surviving synchronous replica with an identifier immediately higher to the failed one). Let N_n be the new owner for the accessed object.
The determinism of the election is important to guarantee that every surviving node redirects its messages to the same node (N_n).
Note that the messages sent to a node can involve more than one object. This will generate a unique message to the new owner, because every object in the original message had the same owner, and so, will have the same substitute.

3. A *"previous grants"* message is sent by each node to the N_n one, giving the set of *access confirmation requests* (ACRs) granted to that sender node (even when such a set is empty). The new owner won't process any new ACR until it receives all these *"previous grants"* messages. Thus, the new owner knows which objects are in the readsets or writesets of sessions approved by the previous owner, and it will reject new requests that have conflicts with those previously granted ACRs.
4. Later on, if the synchronous replica N_n receives a message considering the node as an owner, the message can be processed as if N_n was the original owner. To this end, if the received message was an ACR, then the access conflict management must be performed by N_n, replying the request as usual in COLUP. Moreover, if the received message was an *update request*, then the new owner should reply to the message sending the local version of the object. The update message will be detailed in further steps.
   This behavior maintains the consistency because the new owner of an object will be always elected from the set of synchronous replicas of the object. This guarantees that the value for the object maintained in the new owner is exactly the same value the failed owner had.
5. Whenever the original owner node N_f is recovered from the failure, every alive node will be notified by its local membership monitor. Then, further messages sent from the nodes to the owner N_f must not be redirected to N_n, because the node N_f has been recovered now. In addition, the recovering node sends a specific message (*"I am back"*) to the node that managed its owned objects (i.e. the temporary owner N_n). This message synchronizes the activity of both nodes.
6. Nevertheless, a recently recovered node N_f will receive request messages concerning owned objects that may have been updated during the failure period. In order to manage this situation, a recovered node must consider every object held in its local database as an "asynchronous replica". This consideration will be done for an object o_i until either an *update reply* or *access confirmation reply* is received from a synchronous replica of the object. These replies will be received in the situations described in step 7.
7. If an ACR is received by a recently recovered node N_f, and the involved object has not been already *synchronized* in the node (i.e. the concerning object has not been already updated from a synchronous replica), then N_f must force the synchronization. This synchronization is performed with an *update message* sent to node N_n. The reply to this *update message* will ensure the N_f's local database holds the latest version of the requested object.
   Once the object is updated in the local database, the ACR can be processed as described for a standard owner node.

8. In order to ensure that a recently-recovered node N_f achieves a correct state for its originally synchronized objects (i.e. the node receives an update message for each object o_j that satisfies $N_f \in S(o_i)$), an asynchronous process has to be run.
   This process, will be executed as a low-priority process, and will send an *update request* for each object not already synchronized in N_f.
   Note that the interference of such process in the performance of N_f should be low, because it will only be scheduled during idle periods.
9. The asynchronous process should also include the update, in the local database of the recovered node N_f, of any new object created during the time the node was failed. To perform this update, a simple algorithm may be followed by N_f just at the beginning of its recovery:
   - When N_f recovers from a failure, a query is performed to the local database in order to retrieve the identifier for the more recently inserted object owned by every node in the system. This can be done due to the construction of the object identifiers. As a consequence of these requests, N_f knows, for each node, the last inserted object.
   - Until N_f receives such information from its local database, it will lock any update to its local database. This ensures that the response of the requests does not include any update performed after the recovery of N_f.
   - In addition, and concurrently with these requests, every node in the system sends a *greeting message* to the recovered node. This message includes the identifier of the most recently created object in node N_i.
   - The comparison of the information contained in the *greeting messages*, with the values collected from the local database, makes N_f know the *lost insertions* in each node (i.e. the range of objects inserted during the failure).
   - In addition to these object identifiers, the asynchronous process performs further requests to its local database in order to retrieve a complete list of object identifiers owned by each node in the system, and managed in a synchronous way in N_f.
   - Objects contained in this list of *synchronous identifiers* will be considered as asynchronously maintained objects, until an update message will be sent to its owner node and, as a response of this request, an up-to-date value is obtained, and it is possible to guarantee in the local database of N_f the "synchronism" of such objects. Then, the identifier can be removed from the list of *synchronous identifiers*.

   In order to update every object in the local database, the asynchronous process will use the collected information about *lost insertions* to perform *update requests* to each owner node about these objects.

3.1 The Extended Modification

In the previous section, a basic technique to provide fault tolerance has been described in terms of a modification to the COLU protocol. In this approach, when a node fails, the remaining nodes don't need to perform any specific action.

The implementation of a *replication degree* of T is accomplished with the synchronous replication of the information of each object at least along T nodes.

Let's suppose a system where it has been established a *replication degree* of two. This means that the system will only guarantee the full functionality of the system in presence of less than two failures over the original configuration; i.e. it can occur that subsequent failures of two nodes deal with a system stop.

This undesirable effect can be avoided (or at least attenuated) if the number of synchronous replicas of each object is always maintained over the established *replication degree*. To achieve this, it becomes necessary, whenever this level is decreased (i.e. when a failure of a synchronous replica is detected), to promote the role of a node previously considered asynchronous, making it synchronous for the objects maintained by the failed node in a synchronous way.

The modification of the recovery flavor of COLUP performs the following steps:

- When the system detects a failure of the node N_f, the Membership Monitor of each remaining node in the system notifies the corresponding local Consistency Manager of such failure.
- For the rest of nodes, one of the nodes N_p acting as an asynchronous replica for the set of objects owned by the node N_f, will be promoted, for this set of objects, to synchronous replica.
- The election of the promoting node N_p must be done with a deterministic algorithm. This algorithm is quite simple: It must be always guaranteed the following property.
  "For each object o_i, considering $N_w(o_i)$ as the owner node for o_i, and the replication degree as T, then the set of synchronous replicas for the object is always $S(o_i) = \{N_k | k = w, w+1, \ldots w+T-1\}$, where the operation $+$ only considers the alive nodes".
- To satisfy this property, when a node N_f fails, every node N_i proceeds to recalculate the set $S(o_i)$ for each object owned by the failed node. Three situations can occur at this point:
  - If N_i was synchronous, and it is not the new owner of the objects, then the normal behavior described in section 3 is applied.
  - If N_i was synchronous, and becomes the new owner of the object, then additional actions will take place.
  - If N_i was asynchronous, and now it is synchronous, but not the owner, then it should be considered synchronous, but *not synchronized* for o_i. Thus, the node N_i will follow the same actions as if it was recovering from a failure for the objects o_i.

The depicted technique will promote in a lazy way an asynchronous node N_p to synchronous replica for the set of objects owned by the failed node. Moreover, this new synchronous replica, at the beginning of its promotion, will not be synchronized for each object in this set. Hence, it is possible that a sequence of failures of the new owner nodes $N_{f_2}, \ldots, N_{f_T}$ deal with a situation of lack of synchronous, synchronized replicas of a particular object.

As a consequence, it becomes necessary for the owner node of o_i to maintain the count of synchronous replicas that are currently synchronized with itself. If a node is the unique synchronized replica of an object, and the node fails, no other node will be able

to recover the adequate version for this object. Therefore, no other node will be able to promote to owner of this object, and there will only exist a set of nodes synchronous, but *desynchronized* for this object.

When a unique synchronized node for the object is recovered, then it must be again considered, as described above, synchronous for this object. But, in contrast to the common case, it can be considered *synchronized*, because no other node may have been considered owner of this object during the failure, and hence, it has been impossible for any transaction to change the value of the object.

4 Obtained Results

We have implemented the COLU Protocol as a Consistency Manager for COPLA, and we have experimented with real applications, as well as with particular, specific-purpose test applications. The obtained results for the COLU Protocol showed that the use of prediction of stale-accesses can dramatically decrease the abortion rate of the initiated transactions. In addition, the prediction has a reasonable cost in terms of transaction service time. It was also proven that the adaptative adjustment of the optimum threshold provides an adequate behavior, close enough to the theoretical optimum to consider the algorithm validated.

We have also measured the fault-recovery modification of the COLU Protocol. In these measurements, we have encountered that lazy recovery can be implemented with a minimal impact in the service time of the recovered node, as well as of the surviving nodes.

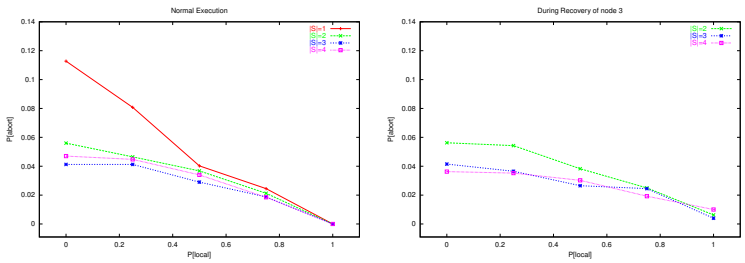
In the experiments, four nodes running COPLA have been used, each one initiating 1000 consecutive (i.e. establishing a thinking time of zero) transactions against the replicated database. Each node is preconfigured to own the same quantity of objects stored in the database. As the total number of objects in the database was established in 600, each node owned a number of 150 objects.

For each transaction, the locality is established in base of a probability (i.e. P[local]). For such transactions elected to be local, any object accessed by the transaction is chosen from the owned set of objects of the node initiating the transaction. This means that no stale access should appear for this transaction. For the non-local transactions, the accessed objects are elected in a uniform way from the rest of nodes (i.e. 450 objects) of the system.

Each transaction accesses in read-write mode to two objects elected in the described way.

For each object, it has been parameterized the size of the synchronous set of replicas (i.e. $|S|$). The minimum value for $|S|$ is 1, because the owner node is always a synchronous replica for its owned objects. We have established the synchronous set of an object $S(o_i) = \{N_{own(o_i)}, N_{own(o_i)+1}, ...\}$, that is, starting with the owner node of the object, the next $|S| - 1$ nodes.

For the recovery measurements, we have always considered N_3 as the faulty node (note that the election of the faulty node has no relevance, due to the symmetry of the algorithms).

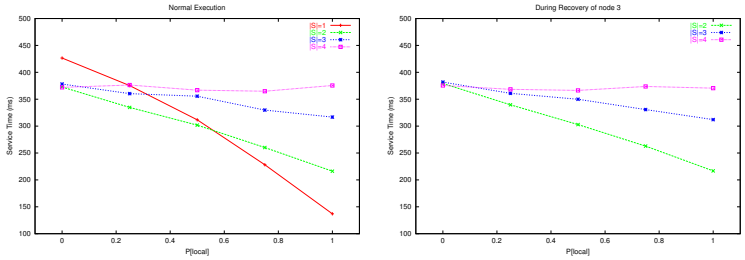


(a) Normal Execution (b) During Recovery of Node 3

Fig. 1. Evolution of the abortion rate for different P[local]

Figure 1 shows a comparison of the abortion rate provided by the COLU protocol during a normal execution (subfigure 1(a)), in contrast to the abortion rates provided when the node 3 is recovering using the lazy recovery technique (subfigure 1(b)). In both situations, when the locality is increased, the abortion rate decreases, reaching the value of 0 when every transaction accesses exclusively to local objects. The maximum value of the abortion rate is obtained when all transactions access to remote objects. It can be seen that the adaptative algorithm for the optimum threshold keeps the abortion rate below reasonable limits. The only exception is encountered when the system is configured to have $|S| = 1$, where the abortion rate reaches the maximum value. In addition, this configuration avoids the system to provide fault-tolerance, because there exists a single replica for each object.

Finally, the figure also shows the similarities between the abortion rates in the rest of scenarios.

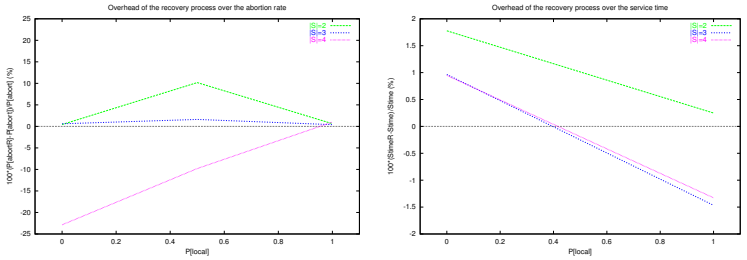


(a) Normal Execution (b) During Recovery of Node 3

Fig. 2. Evolution of the service time for different P[local]

Figure 2 performs an analogous comparison with the provided service time. The obtained results showed that the service time obtained is more sensible to the locality when $|S|$ is decreased. This fact is also kept during the recovery of a node. Finally, it

was shown that the performance of the system was very similar during the recovery in contrast to the normal execution of the system.



(a) Abortion rate (b) Service time

Fig. 3. Overhead introduced by the Recovery, for different P[local]

In the figure 3, we show the overhead introduced in the system by the lazy recovery process. It is shown that the abortion rate (subfigure 3(a)) is increased below a 10% when the number of synchronous replicas is lower than 4 (for $|S| = 3$ it is kept below 1.5%). In addition, when the number of replicas is increased, the overhead is reduced, even introducing an improvement. This improvement is caused by the lower level of concurrency introduced by the recovering node.

Regarding performance (subfigure 3(a)), the locality of transactions reduces the overhead of lazy recovery. Higher values of $|S|$ also reduce this overhead, even producing better performance for certain configurations.

As a conclusion, *lazy recovery* produces overheads in terms of performance and abortion rate lower that 10%, and in some configurations it may provide better results than in the non-recovering case.

5 Related Work

One of the most important characteristics of a good recovery protocol consists of the capability of the non-recovering replicas of serving incoming transactions during the recovery of a node. This guarantees a higher availability while the recovery process is carried out.

Moreover, avoiding the maintenance of activity logs in the survining nodes when a failure is detected is another desirable feature of a recovery protocol. This reduces the overhead introduced by the system for providing fault-tolerance to the distributed service.

In replicated databases this means that transactions must be served even when any replica is being recovered. Moreover, a minimal overhead is wanted in this recovery process, as well as during the time a node is considered down.

Some eager replication protocols[1,8,13] are based on a total order delivery policy such as atomic broadcast primitives[3]. Many of them do not discuss their recovery subprotocols, although they share the same principles of their replication techniques. The

main disadvantage of such algorithms consists of their blocking nature[9], introducing a "not-so-minimal" interference in the system. This problem was also present in our previous eager replication protocols[11].

Newer approaches make use of the principle of *enriched view synchrony* [2], but the need of logging the activity often introduces a poor scalability in the system during relatively long time failures.

Regarding the amount of transferred data, when the update information requested by the recovering node exceeds a certain amount, synchronous recovery approaches are unpracticable, due to the interference introduced in the system. To avoid this, *cascading reconfiguration with lazy transfer* was proposed in [9] as an asynchronous alternative.

In [7] another non-blocking recovery solution is presented, solving also some minor problems of the work described in [9]. However, the recovery tasks of [7] need a transaction log that maintains all writesets of the lost transactions for faulty nodes, and it has to transfer such logs to the recovering nodes. In order to minimize the amount of data that needs to be transferred, this log is periodically shortened using checkpoints. Thus, when a replica recovers, the latest checkpoint is initially applied, followed by the logged updates stored after such checkpoint. Our solution only considers the latest versions of the objects that have been updated during the failure period, so the costs of these two approaches have to be similar.

The main advantage offered by our algorithm is that our recovery tasks are natively supported by the basic hybrid protocol. Thus, the recovery is part of the basic algorithm, and no additional code is needed to consider failures. Another advantage is that, as the update is performed on demand (i.e. when the recovering node accesses an outdated object), there is no need of locking any object in the rest of replicas. Finally, our proposal makes use of a less expensive communication primitive (just a uniform causal reliable multicast is needed, instead of a uniform atomic one) to carry out the recovery tasks. Unfortunately, the transaction service time of COLUP is usually longer than that of pure lazy database replication protocols.

6 Conclusions

We have presented a configurable solution to replicated transactional systems making use of a hybrid approach, flexible enough to have a variety of behaviors from a pure eager protocol, to a pure lazy one.

As a result, the presented approach is able to provide the advantages of pure eager and pure lazy approaches, and it can also be configured to include asynchronous replication, forestalling their traditionally found disadvantages. Thus, our approach can reduce the abortion rate, improving the performance of the system.

A new *lazy recovering protocol* is presented. It makes use of the principles applied by the COLUP algorithms to get the full state of a recovering node without suspending the activity of such a node during the process. Moreover, there is no node in the system suspending its activity during the system recovery, because this recovery is performed following a lazy paradigm. In addition, the algorithm avoids the necessity of any log maintenance, providing thus a scalable behavior.

The abortion rate, however, is also managed with the statistical conservative techniques on which the COLU protocol is based. This decreases the abortion rate of the transactions initiated in the recovering node, and the recovery algorithm takes profit of the infrastructure used by the replication protocol to propagate the changes.

As a result, the proposed recovery algorithm will not interfere in the functionality of the system, allowing the recovering node to proceed immediately after its re-inclusion. Moreover, the performance of the rest of the system will be almost unchanged, because the only node making additional work will be the recovering one. On the other hand, this recovering node will suffer this overhead with a lazy policy, making it possible for the local scheduler of the recovering node to proceed with the update of the local replica with a conservative policy, using the idle time of the node to advance part of such updates.

References

1. D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. *Lecture Notes in Computer Science*, 1300:496–503, 1997.
2. Ö. Babaoglu, A. Bartoli, and G. Dini. Enriched view syncrhony: A programming paradigm for partitionable asynchronous distributed systems. *IEEE Transactions on Computers*, 46(6):642–658, June 1997.
3. V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. ACM Press, 2nd edition, 1993. ISBN 0-201-62427-3.
4. Instituto Tecnológico de Informática. GlobData Web Site, 2004. Accessible in URL: *http://globdata.iti.es*.
5. L. Irún-Briz, F. D. Muñoz-Escoí, and J. M. Bernabéu-Aubán. An improved optimistic and fault-tolerant replication protocol. In *Proc. of 3rd Workshop on Databases in Networked Information Systems*, volume 2822 of *Lecture Notes in Computer Science*, pages 188–200, Aizu, Japan, September 2003. Springer.
6. L. Irún-Briz, F. D. Muñoz-Escoí, and J. M. Bernabéu-Aubán. Improving the bevahior of optimistic lazy replication. In *XI Jornadas de Concurrencia*, Benicassim, Spain, June 2003.
7. 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.
8. B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333–379, September 2000.
9. B. Kemme, A. Bartoli, and Ö. Babaoglu. Online reconfiguration in replicated databases based on group communication. In *Proc. of the IEEE Int. Conf. on Dependable Systems and Networks*, pages 117–130, Göteborg, Sweden, July 2001.
10. F. D. Muñoz-Escoí, Ó. Gomis-Hilario, P. Galdámez, and J. M. Bernabéu-Aubán. HMM: A cluster membership service. In *Proc. of 7th International Euro-Par Conference*, volume 2150 of *Lecture Notes in Computer Science*, pages 773–782, Manchester, UK, August 2001. Springer.
11. 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*, pages 97–104, Valencia, Spain, November 2001.

12. V. P. Nelson. Fault-tolerant computing: Fundamental concepts. *IEEE Computer*, 23(7):19–25, July 1990.
13. L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. Strong replication in the GlobData middleware. In *Proc. of Workshop on Dependable Middleware-Based Systems (in DSN 2002)*, pages G96–G104, Washington D.C., USA, 2002.