Adding Amnesia Support and Compacting Mechanisms to Replicated Database Recovery

L. H. García-Muñoz, R. de Juan-Marín, J. E. Armendáriz-Íñigo and F. D. Muñoz-Escoí

Instituto Tecnológico de Informática - Universidad Politécnica de Valencia
Camino de Vera, s/n - 46022 Valencia, Spain

{lgarcia, rjuan, armendariz, fmunyoz}@iti.upv.es

Technical Report TR-ITI-ITE-07/08

Adding Amnesia Support and Compacting Mechanisms to Replicated Database Recovery

L. H. García-Muñoz, R. de Juan-Marín, J. E. Armendáriz-Íñigo and F. D. Muñoz-Escoí

Instituto Tecnológico de Informática - Universidad Politécnica de Valencia
Camino de Vera, s/n - 46022 Valencia, Spain

Technical Report TR-ITI-ITE-07/08

e-mail: {lgarcia, rjuan, armendariz, fmunyoz}@iti.upv.es

Abstract

Nowadays eager update everywhere replication protocols are widely used in replicated databases. They work together with recovery protocols in order to provide highly available and fault-tolerant information systems. This paper provides a general way for adding amnesia support and compacting mechanisms to these recovery protocols. The idea of these enhancements is to consider on one hand a more realistic failure model scenario which fits better with the real world and on the other hand to minimize the recovery cost obtaining then a more efficient recovery protocol.

1 Introduction

Database replication consists in maintaining identical copies of a given database at multiple network nodes. This provides a higher performance, clients access their local replica or are forwarded to the less loaded one; and availability: whenever a node fails, its associated clients are silently redirected to another available one. Replication protocols can be designed for eager or lazy replication [12], and for executing updates in a primary copy or at all node replicas [18]. With eager replication we can keep all replicas exactly synchronized at all nodes, but this could have an expensive cost. With the lazy alternative we can introduce replication without severely affecting performance, but it can compromise consistency. Many replication protocols are based on eager update everywhere with a *read one, write all available* (ROWAA) approach. As we have briefly highlighted before, these replication protocols provide high availability, in the sense that clients executing transactions at a node that fails are transparently forwarded to another available replica. However, only a few of them deal with the possible reconnection of the failed node, which is managed by recovery protocols [2, 5, 6, 14, 15, 16, 17]. The aim of the recovery protocols is to bring failed or temporarily disconnected nodes back into the network as fully functional peers, by reconciling the database state of these recovering nodes with that of the active nodes. This could be done by logging transactions and transferring this log to recovering nodes so they can process missed transactions, or transferring the current state of the objects that have been updated in the database since the recovering node failed.

In the design and development of a generic recovery protocol it is necessary to: consider the characteristics of the used replication protocol; take into account that the recovery process must be as fast as possible; have a minimum interference with the user transactions; and, to introduce the minimum system overhead.

Previous works in this area include recovery protocols for replication protocols that make use of Group Communication Systems (GCS) [7] and virtual synchrony [4], enriched view synchrony [17], recovery based on logs [14], parallel recovery [16], lazy recovery [15], recovery algorithms based on configurable logging for broadcast protocols [5], version-based recovery[6], and protocols that intend to be a little more general for a group of replication protocols based on eager update everywhere and ROWAA [2].

This paper is focused in the recovery protocol for eager update everywhere replication protocols, proposing some optimizations to the work presented in [2]. These enhancements include amnesia support, and a better performance reducing the amount of data to save in the actions done before recovering and the amount of data to transfer at recovering time. The main idea in the last case is to compact recovery data eliminating redundant information.

The main contributions of this paper are: to add support for the amnesia in the original protocol, to improve the performance of the original protocol compacting the necessary information for the recovery. In addition, we provide a table with the results of a simulation, where the advantages of the compacting approach are verified.

The rest of this paper is distributed as follows. In Section 2 we provide the system model. Section 3 deals with the basic recovery protocol. In Section 4 we explain the necessary actions for the amnesia support. Next, Section 5 relates the process of compacting recovery information. Later, Section 6 shows the simulation results resumed in a table, followed by the related works in Section 7. In the final Section 8, we provide our conclusions.

2 System Model

The original recovery protocol has been designed for database replicated systems composed by several replicas –each one in a different node–. These nodes belong to a partially synchronous distributed system: their clocks are not synchronized but the message transmission time is bounded. The database state is fully replicated in each node.

This replicated system uses a group communication system (*GCS*) [7]. Point-to-point and broadcast deliveries are supported. The minimum guarantee provided is a FIFO and reliable communication. A group membership service is also assumed, who *knows* in advance the identity of all potential system nodes. These nodes can join the group and leave it either explicitly or implicitly by crashing, raising a *view change event*. Therefore, each time a membership change happens, i.e. any time the failure or the recovery of one of the member nodes occurs, it supplies consistent information about the current set of reachable members as a view. The group membership service combined with the *GCS* provides *Virtual Synchrony* [4] guarantees, which is achieved using *sending view delivery* multicast [7] enforcing that messages are delivered in the view they were sent. A *primary component* [7] model is followed in case of network partitioning.

The replicated system assumes the *crash-recovery with partial-amnesia* model instead of the crash or fail-stop model [13] for node failures. This implies that an outdated node must be recovered from two "lost of updateness": forgotten state and missed state. This assumption supports a more realistic and precise way to perform the recovery process. So the assumed model allows to recover failed nodes from their previous crashing state maintaining their assigned node identifiers. Consequently, when a node crashes, every active node must abort any transaction started by the failed node whose commit messages have not been yet delivered. A similar behavior is adopted when the system can not go on because the progress condition has been lost. In this situation, the nodes in minority (e.g. disconnected) must also abort the started transactions whose commit message has not been yet delivered. Thus, the whole activity that was not committed during the working life is aborted.

3 Original Recovery Protocol

The original recovery protocol presented in [1] has been thought for *eager update everywhere* database replication protocols and proposes the use of *DB-partitions*. In fact, it was originally designed for providing recovery support for the *ERP* and *TORPE* [2] replication protocols. They are ROWAA, which use voting techniques and propagate transaction writesets. Therefore, for each transaction that performs updates, its replication execution is performed in two messages: first it is broadcast the *remote* message which propagates the updates and secondly it is spread the *commit* message which confirms the replicated execution. Their main difference is that the first one is based on reliable but unordered multicast using a dynamic

deadlock prevention scheme, while the last one is based on total order broadcast. This proposed recovery protocol can be outlined as follows:

- The system has a database table named $MISSED$, which maintains all the information that will be needed for recovery purposes. Each time a new view is installed a new entry is inserted in the $MISSED$ table if there are failed nodes. Each entry in $MISSED$ table contains: the view identifier, the identifiers of crashed nodes in this view –$SITES$–, and the identifiers list of data items modified during this view –OID_LIST–. The two first ones are established at the beginning of the view, while the last one increases as long as the view passes.

- When a set of crashed nodes reconnects to the replicated system, the recovery protocol will choose one node as the *recoverer* with a deterministic function *oldest alive* based on the metadata of the recovery protocol. Then in a first step the *recoverer* transfers the metadata recovery information to all reconnected nodes. This metadata information contains: the identifiers of modified objects, and the crashed node identifiers in each view lost by the oldest crashed node being recovered. After, *recoverer* and *recovering* nodes start to set up with processes or threads *DB-partitions*, which help the *recoverer* to know the objects to be updated in each outdated node, while each *recovering* node knows which objects must be updated in itself. These *DB-partitions* are also used in order to block in each replica the current user transactions whose modified objects conflict with its *DB-partitions*. Subsequently, the *recoverer* starts to recover each *recovering* node view by view. For each lost view, the *recoverer* transfers the state of the modified objects during this view. And, once the view has been recovered in the *recovering* node, it notifies the recovery of this view to all alive nodes. The recovery process ends in each *recovering* node once it has updated all its lost views.

- As a transaction broadcast is performed spreading two messages –*remote* and *commit*–, it is possible that a reconnected node receives only the second one, unknowing then the changes to commit [14]. In this case the replication protocol will transfer the associated writesets to these nodes. This behavior implies that transaction writesets are maintained in the sender node until the *commit* message is broadcast.

But this recovery protocol presents the following two problems:

- Amnesia phenomenon. Instead of assuming the *crash-recovery with partial amnesia* failure model, the system [9] does not handle it in a perfect way. This problem arises because once the replication protocol propagates the *commit* message associated to one transaction, and it is delivered, the system assumes that this transaction is being committed locally in all replicas. But this assumption even using strong virtual synchrony [7] is not always true. It is possible that a replica receives a transaction *commit* message, but before applying the commit the replica crashes, as it is commented in [19] –the basic idea is that message delivery does not imply correct message processing–. At this time, this replica has not committed the transaction and has lost the *commit* message, but other replicas assume that the transaction has been committed in all replicas even in the crashed one that has triggered a view change event, because it received the *commit* message in the previous view. The problem will arise when this crashed node reconnects to the replicated system, because it will not have committed this transaction and the rest of the system will not include among the necessary recovery information the updates performed by this transaction, only will contain transactions whose *commit* message has been propagated after the replica crash event, being impossible for the recovering protocol to update this transaction, arising then a problem of replicated state inconsistency.

- Large $MISSED$ table and redundant recovery information. If in the system there are long-term crashed nodes –meaning nodes failed during many views– and there are also high update rates it is possible that the $MISSED$ table enlarges significantly with high levels of redundant information, situation that is strongly discouraged. Redundant recovery information will appear because it is possible that the same item has been modified in several views where the crashed nodes set is very similar. In this case if an item is modified during several views, only knowing the last time –meaning the last view– it was updated is enough. Therefore, it will be interesting to apply algorithms that

avoid redundant recovery information, because the larger $MISSED$ tables the greater the recovery information management overhead becomes.

In the following section we will present and study different approaches for solving these problems improving the original recovery protocol.

4 Amnesia Support

In order to provide amnesia support different approaches can be considered. These approaches can be classified depending on which recovery information they use. On one hand, there are the ones using the broadcast messages –log-based– [5, 16, 14] and, on the other hand there are the ones using the information maintained in the database –version-based– [2, 5, 6, 15, 17].

But before talking about how amnesia support can be provided in the basic recovery protocol, it must be considered how this amnesia phenomenon manifests. In [11] it has been already detailed how the amnesia phenomenon manifests in replicated transactional systems and how it can be dealt with using log-based recovery approaches. In such work, it is said that the amnesia phenomenon manifests at two different levels:

- *Transport level*. At this level, amnesia implies that the system does not remember *which messages have been received*. In fact, the amnesia implies that received messages non-persistently stored are lost when the node crashes, generating a problem when they belong to transactions that the replicated system has committed but which have not been already committed in the crashed node.
- *Replica level*. The amnesia is manifested here in the fact that the node "forgets" *which were the really committed transactions*.

Once it has been detailed the ways in which the amnesia manifests we will present how it can be solved. Obviously, depending on the recovery policy used, log-based or version-based, the information that must be maintained to solve the amnesia problem differs.

4.1 Logging Approach

If a logging approach is adopted, the information maintained in order to perform the amnesia recovery process will be the broadcast replication messages, in this case two messages for each propagated transaction: *remote* and *commit*. And the amnesia recovery must be performed before starting the recovery of missed updates –the latter will be done by the original recovery protocol–. The amnesia recovery process will consist in reapplying the messages belonging to non really committed transactions.

The first question that must be answered is who must store persistently the broadcast messages, in order to overcome the amnesia at transport level. The replica which started the transaction is a possible solution. But it forces the senders to control for each broadcast transaction which replicas have really committed it, being only possible to discard the associated messages once all alive nodes have acknowledged its commit. Moreover, the amnesia recovery process can only be performed when all senders are alive, in order to have available the necessary messages. The other option, and more interesting for an update everywhere approach, is that each node stores persistently the received messages, maintaining them as long as the associated transaction, t, has not been committed and discarding them as soon as, t its really committed in the replica. But, it must be remarked that the message persist process must be performed atomically inside the delivery process as already discussed in [19] with its "successful delivery" concept. Moreover, messages belonging to aborted or rolled-back transactions must be also deleted.

Once the amnesia phenomenon is solved at transport level, it is necessary to manage the amnesia problem at replica level. At this level the amnesia implies that the system can not remember which were the really committed transactions. Even for those transactions for which the "commit" message was applied, it is possible for the system to fail *during* the commit.

Then the amnesia recovery process in a replica will consist in reapplying the received and persistently stored messages in this replica that have not been already deleted, because it implies that the corresponding

transactions have not been committed in the replica. These messages are applied in the same order as they were originally received.

It must be noticed, that some of the permanent stored and not already deleted messages in a replica can belong to really committed transactions whose messages have not been deleted because the node crashed before doing it. Thus, they would be applied twice. This situation will not be desired in two different replication scenarios. On one hand, if the process replication propagates operations, because it will lead to diverging states in different replicas for applying twice the same operation. And on the other hand, when the replication system propagates updates –target replication protocols of this recovery protocol–, applying twice a writeset does not lead to diverging states but it can imply high overhead if writesets are large.

In order to avoid these undesired situations it is necessary to manage extra information. One possibility is that each replica has an extra database table where a new entry is created each time a new transaction starts to be processed storing its identifier and assigning the *processing* state value. Later, when its transaction commit is performed in the replica, the system must change atomically with the commit process its state in the extra table to *committed*. Subsequently, as soon as a transaction has been successfully committed –and also in the rolled-back case– in a replica, the system must delete first its associated messages in this replica and secondly its entry in the extra database table.

The information maintained in this table will be helpful in the amnesia recovery process in order to distinguish which persistently stored messages must be reapplied and which not. The idea, as it has been said before, is to discard the messages that the system would have deleted in failure absence because they belong to a committed transaction, but which have not been erased due to the replica crash.

It also must be noticed, that in this process is not needed to apply the *remote* messages whose associated *commit* messages have not been received, because it implies that they have been committed in the subsequent view, and therefore their changes are applied during the recovery of its first missed view.

Finally, once the amnesia recovery process ends, the original recovery protocol mechanism can start.

4.2 Identifier Approach

This is a version-based amnesia recovery approach, therefore this approach does not need to store propagated messages and does not consider the amnesia at the transport level.

The background idea of this solution is that the reconnected node transfers to the *recoverer* node the identifier of its last committed transaction. Then the *recoverer* can transfer to the *recovering* node the objects modified in subsequent committed transactions before the system installed the new view after the *recovering* node crash. Once the *recoverer* node has transferred this information the system can follow the recovery process as it is proposed in the original recovery protocol.

But, in order to deploy this amnesia recovery approach, it is necessary that replicas mark which is the last transaction that modified each data object, being necessary a mechanism that performs it. Moreover, it is needed that each replica remembers which is the identifier of its last committed transaction, updating it each time a new transaction is being terminated as an internal step of its commit process. In spite of solving this problem, two main considerations discourage the use of this approach.

On one hand, the amnesia recovery information for this technique presents a finer granularity –transaction identifier– than the missed recovery information –view identifier– used by the original recovery protocol. And this information must be generated always because we do not know when the amnesia problem could appear.

On the other hand, and deriving from the first consideration, it will be necessary to include the transaction level granularity in the $MISSED$ table and create entries for views without crashed nodes. Otherwise the system will maintain two different sets of recovery information appearing a problem of redundant recovery information.

4.3 Summary

Which of these two approaches must be adopted? Both approaches imply an overhead during the normal work in the system in order to generate and manage the information that will help the system to solve the amnesia problem –one persisting messages, the other generating metadata information at the database level–.

Another consideration is that in the first approach the crashed node is the one who has the information to perform the amnesia recovery process, while in the second one it relies on the information maintained in alive nodes.

Therefore, depending on the necessities of our system it will be decided the approach that best fits the recovery protocol requirements. It must be noticed that the version-based approach can use the same mechanism for generating the information needed in both recovery processes: amnesia and missed state. Anyway, if to maintain the basic recovery protocol work way is a design requirement the only approach that can be adopted is the logging one.

In this paper we adopt the logging one, because if the version-based approach is adopted, the original recovery protocol taken as point of departure will need many modifications or it will have a lot of redundant information.

Once it has been detailed how the amnesia support can be provided in the recovery protocol, we will continue with the second proposed improvement.

5 Compacting Recovery Information

In order to increase the performance at the moment of determining and transferring the necessary information for the synchronization of recovering nodes, we propose some modifications based on packing information that enhance the original recovery protocol described in [1]. This could be done by compacting the records in the $MISSED$ table, and with this, minimize the objects to transmit and to apply them in the recovering node, reducing thus the transmission and synchronization time.

Originally the $MISSED$ table stores in each record, i.e. view, the identifiers of updated objects whose changes have been lost by crashed nodes. Therefore, these identifiers can be repeated in different $MISSED$ view entries because these objects have been modified in two or more views where there were failed nodes.

These object identifiers can be packed due to the fact that the recovery information only maintains the identifiers of updated objects. The state of these objects is retrieved by the *recoverer* from the database at recovering time. Moreover, if a *recovering* node, k, has to recover the state of an object modified in different views lost by k it will receive as many times the item value, when transferring its state only once is enough. As a consequence, it is not relevant to repeat the identifier of an updated object across several views, being only necessary to maintain it in the last view it was modified and can be erased, if it is, in other previous views.

During DB-partition generation, as user transactions are blocked, there is no compacting process going on in the system. Hence, possible generation of non-correct DB-partitions is avoided. Once this metadata has been transferred, establishing the *DB-partitions*, the compacting process is restarted. This blocking process is not necessary if the whole set of failed nodes in the previous view is contained in the current set of failed nodes. In fact, it must be remarked that this work behavior is already provided by the original recovery protocol due to the established *DB-partitions*, which block any update access.

We consider that the actions for the amnesia support are performed during the execution of user transactions. Whenever one (or more than one) node fails, the recovery protocol starts the execution of the actions to advance the recovery of failed nodes. To this end:

- When a transaction commits, the field which contains the identifiers of the updated objects, OID_LIST, will be updated in the following way:

    1. For each object in the $WriteSet$, the OID_LIST is reviewed to verify if the object is already included in it or not. If it is not, it is included and is looked for in previous views OID_LIST, eliminating it from the OID_LIST in which it appears, compacting thus the OID_LIST, i.e. the information to transfer when a node recovers.
    2. If as a result of this elimination, an OID_LIST is emptied, the content of the field $SITES$ is included in to the field $SITES$ of the next record, and the actual record in the table $MISSED$ can be eliminated.

When a node reconnects to a replicated system, the new view is installed and the actions for the amnesia recovery are performed locally at the recovering node. This is a lightweight process (i.e. only a few stored messages have to be processed) in comparison to the database state recovery process itself. The other nodes know who is the recovering node, and every one performs locally the next actions:

1. The $MISSED$ table is scanned looking for the recovering node in the field $SITES$ until the view that contains the recovering node is found. The objects for which the recovering node needs to update its state are the elements of OID_LIST of this view and the subsequent views.

2. At the recoverer node, the recovery information is sent to the recovering node according to the original protocol.

3. Once the recovering node has confirmed the update of a view, the node is eliminated from the $SITES$ field in this view, and if it is the last item, also the record that contains this view is eliminated.

4. If a recoverer node fails during the recovering process, then another node is elected to be the new recoverer, according to the original protocol. And it will create the partitions pending to be transferred, according to the previous points, and then it will perform the object transfer to recovering nodes, again as in the original protocol.

It is important to note that in a view change consisting in the join and leave of several nodes, we must first update the information about failed nodes, and later execute the recovery process. As a final remark, this compacting process will help the recovery protocol to minimize the needed recovery information to be transferred. However, its compression rate will depend on the user application. If replication updates concentrate in few data items among several views the compacting will have high rates, but if these changes are highly scattered the compacting rate values will be low.

6 Simulation Results

We have simulated the compacting enhancement in order to know which level of improvement provides. We have considered three replicated scenarios with 5, 9 and 25 nodes each one. The replicated database has 100000 data objects. All simulations start having all replicas updated and alive. Then, we start to crash nodes one by one –installing a new view each time a node crashes–, until the system reaches the minimum primary partition in each scenario. At this point two different recovery sequences are simulated. In the first one, denoted as order 1, the crashed nodes are reconnected one by one in the same order as they crashed, while in the second, denoted as order 2, they are reconnected one by one but reversing the order that they have crashed. In both cases, each time a node reconnects a new view is installed, and immediately the system starts its recovery, ending its recovery process before reconnecting the following one. In any installed view we assume that the replicated system performs 250 transactions successfully, and each transaction modifies 20 database objects. All simulation parameters are described in Table 1.

This simulation has not considered the costs of: managing the amnesia problem, and recovery information compacting. The amnesia problem, as it has been said before, is solved using a log-based approach, persisting the delivered messages during the replication work, and applying those not committed during the amnesia recovery process. Thus, it implies two costs: one in the replication work and another in the recovery work. The first cost is not considered because does not happen in the recovery process. The second one, although appears in the recovery process, is not considered because it is very low compared to the recovery process itself –usually it will consist in applying few messages (writesets) and in our simulation are very small–. The recovery information compacting cost is not taken into account because this work is performed online, therefore its associated overhead penalizes the replication work performance, but not the recovery.

The simulation results show that the more views a crashed node looses the better the compacting technique behaves, which is a logical result. In fact, when more updates a crashed node misses the probabilities of modifying the same object increases. Either in the Table 2 and in the Figure 1 we can observe the same behavior. When a crashed node has lost only one view the compacting technique does not provide any

Parameter	*Value*	*Parameter*	*Value*
Number of items in the database	100000	Time for a read	4 ms
Number of servers	5, 9, 25	Time for a write	6 ms
Transactions per view	250	Time for an identifier read	1 ms
Transaction length	20 modified objects	Time for an identifier write	3 ms
Identifier size	4 bytes	CPU time use for an I/O operation	0,4 ms
Object size	200 bytes	Time for a point to point message	0,07 ms
Maximum message size	64 Kbytes	Time for a broadcast message	0,21 ms
CPU time for a network operation	0,07 ms		

Table 1: Simulator Parameters.

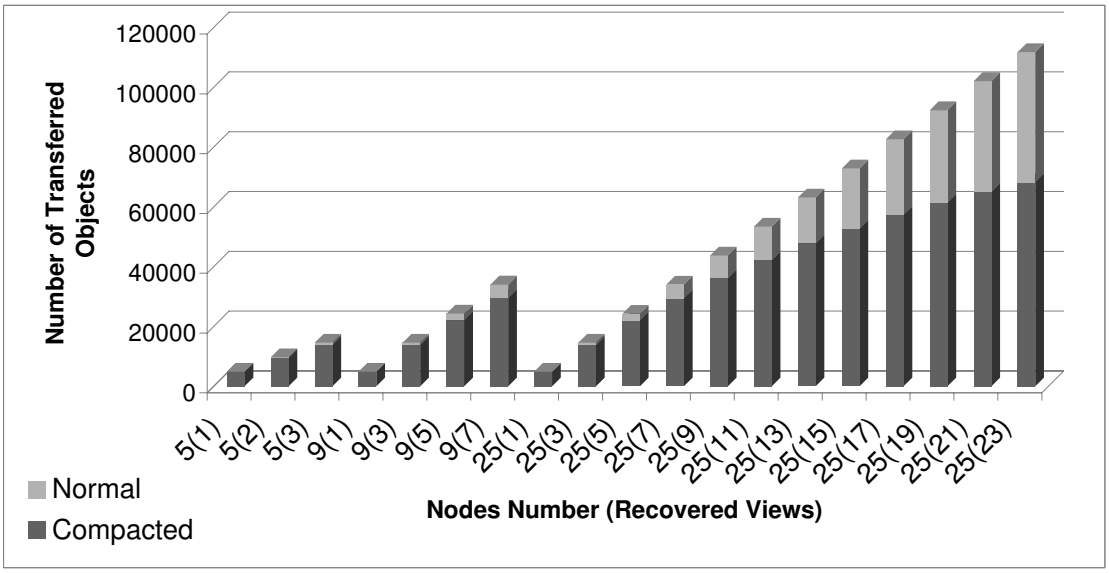

Figure 1: Object Compactness

improvement because it has been unable to work. But, as long as the crashed node misses more views the compacting technique provides better results.

It must also be noticed that the original recovery protocol could arrive to transfer a greater number of objects than objects has the original database. This occurs because it transfers for each lost view all the modified (and created objects in this view) independently they are transferred when recovering other views where these objects have been also modified. This situation with our recovery protocol enhancement is avoided. And in the worst case the proposed solution will transfer the whole database because during the inactivity period of the recovered node all the objects of the database have been modified.

Obviously, we must say that the improvement provided by our approach depends on the replicated system load activity, the update work rate, and the changed items rate. For the first two ones, we can consider in a general way that when higher they are better our compacting technique behaves. This is because the probabilities of modifying the same object in different views increase. This consideration drives us to the changed items rate, which is really the most important parameter. It tells us if the performed updates are focused in few items or not. Then for our technique it is interesting that changes are focused in as few items as possible. In fact, the worst scenario for our technique will be the one in which all the modifications are performed in different objects.

Order	*Nodes*	*Recovered Views*	*Normal Time*	*Compacted Time*
1	5	2	165.33	161.76
2	5	1	82.82	82.82
2	5	3	247.74	236.09
1	9	4	330.43	307.45
2	9	1	82.52	82.52
2	9	3	247.60	235.92
2	9	5	413.18	376.31
2	9	7	578.32	501.90
1	25	12	990.95	766.20
2	25	1	82.76	82.76
2	25	3	247.70	235.59
2	25	5	412.48	374.81
2	25	7	577.78	500.04
2	25	9	742.98	614.22
2	25	11	908.20	717.70
2	25	13	1073.80	812.69
2	25	15	1239.09	897.48
2	25	17	1404.16	973.79
2	25	19	1569.50	1042.92
2	25	21	1734.96	1104.76
2	25	23	1899.88	1160.78

Table 2: Recovery times in seconds.

As final conclusion, we can say that our enhanced recovery protocol works better in some of the worst scenarios from a recovery point of view: when the crashed node has lost a lot of updates and the changed items rate is not very high.

7 Related Work

For solving the recovery problem [3] database replication literature has largely recommended the crash recovery failure model use as it is proposed in [17, 6, 5, 1, 16], while process replication has traditionally adopted the fail stop failure model as [4] proposes. The use of different approaches for these two areas is due to the fact that usually the first one manages large data amounts, and it adopts the crash recovery with partial amnesia failure model in order to minimize the recovery information to transfer.

The crash-recovery with partial amnesia failure model adoption implies that the associated recovery protocols have to solve the amnesia problem. This problem has been considered in different papers as [19, 10, 11] and different recovery protocols have presented ways for dealing with. The *CLOB* recovery protocol presented in [5] and the *Checking Version Numbers* proposed in [17] support amnesia managing it in a log-based and version-based way, respectively. They are protocols that have been proposed for replicated systems with the following characteristics: update everywhere, eager and using total order delivery.

In regard to the compactness technique, [8] uses it in order to optimize the database recovery. In this case, this technique is used to minimize the information size that must maintained and subsequently transferred in order to perform the recovery processes. Such paper also presents experimental results about the benefits introduced by using this technique, reaching up to 32% time cost reductions.

The background idea of our compacting technique is very similar to the one used in one of the recovery protocols presented in [17]. This protocol maintained in a database table the identifiers of the modified objects when there were failed nodes. Each one of these object identifiers was inserted in a different row, storing at the same time the identifier of the transaction which modified the object. Therefore, when an object was modified the system checked if its identifier was already inserted in this table. If it has not, the protocol created a new entry where inserted the identifier object and the transaction identifier. If it

already existed an entry with this object identifier, the protocol simply updated in this entry the transaction identifier. So, this recovery protocol also avoids redundant information, but it uses a more refined metadata granularity –transaction identifier– than our enhanced protocol –view identifier–.

8 Conclusions

In this paper we have reviewed the functionality of the original recovery protocol described in [2]. We have enhanced it providing an accurated amnesia support and incorporating a compacting method for improving its performance.

The amnesia support has been improved using a log-based technique which consists on persisting the messages as soon as they are delivered in each node, in fact they must be persisted atomically in the delivery process. This work way provides a similar support to the one proposed in [19].

Our compacting technique avoids that any data object identifier appears more than once in the *MISSED* table. Then this mechanism reduces the size of recovery messages, either the ones that set up the DB-partitions and the ones which transfer the missed values.

Tests have been made with a simulation model and the advantages of the enhanced recovery protocol have been verified when comparing the results of both protocols. The obtained results have pointed out how our proposed compacting technique provides better results when the number of lost views by a crashed node increases. Thus, our compacting technique has improved the recovery protocol performance for recoveries of long-term failure periods.

9 Acknowledgements

This work has been partially supported by FEDER and the Spanish MEC grant TIN2006-14738-C02.

References

[1] J. E. Armendáriz, F. D. Muñoz, H. Decker, J. R. Juárez, and J. R. G. de Mendívil. A protocol for reconciling recovery and high-availability in replicated databases. *21st International Symposium on Computer Information Sciences, Springer*, 4263:634–644, November 2006.

[2] J. E. Armendáriz-Íñigo. *Design and Implementation of Database Replication Protocols in the MADIS Architecture*. PhD thesis, Depto. de Matemática e Informática, Univ. Pública de Navarra, Pamplona, Spain, Feb. 2006.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, MA, EE.UU., 1987.

[4] K. P. Birman and R. V. Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1993.

[5] F. Castro, J. Esparza, M. Ruiz, L. Irún, H. Decker, and F. Muñoz. CLOB: Communication support for efficient replicated database recovery. In *13th Euromicro PDP*, pages 314–321, Lugano, Sw, 2005. IEEE Computer Society.

[6] F. Castro, L. Irún, F. García, and F. Muñoz. FOBr: A version-based recovery protocol for replicated databases. In *13th Euromicro PDP*, pages 306–313, Lugano, Sw, 2005.

[7] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 4(33):1–43, 2001.

[8] J. P. Civera, M. I. Ruiz-Fuertes, L. H. García-Muñoz, and F. D. Muñoz-Escoí. Optimizing certification-based database recovery. In *6th International Symposium on Parallel and Distributed Computing, ISPDC, Hagenberg, Austria*, 2007.

[9] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

[10] R. de Juan-Marín, L. Irún-Briz, and F. D. Muñoz-Escoí. Recovery strategies for linear replication. In *ISPA*, pages 710–723, 2006.

[11] R. de Juan-Marín, L. Irún-Briz, and F. D. Muñoz-Escoí. Supporting amnesia in log-based recovery protocols. In *Euro American Conference on Telematics and Information Systems, EATIS, Faro, Portugal*, 2007.

[12] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *ACM SIGMOD International Conference on Management of Data*, pages 173–182, 1996.

[13] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. ACM Press, 1993.

[14] J. Holliday. Replicated database recovery using multicast communication. In *NCA*, pages 104–107. IEEE Computer Society, 2001.

[15] L. Irún, F. Castro, F. García, A. Calero, and F. Muñoz. Lazy recovery in a hybrid database replication protocol. In *XII Jornadas de Concurrencia y Sistemas Distribuidos*, 2004.

[16] R. Jiménez-Peris, M. Patiño-Martínez, and G. Alonso. Non-intrusive, parallel recovery of replicated data. In *SRDS*, pages 150–159. IEEE Computer Society, 2002.

[17] B. Kemme, A. Bartoli, and O. Babaoğlu. Online reconfiguration in replicated databases based on group communication. In *Intl.Conf.on Dependable Systems and Networks*, pages 117–130, Washington, DC, USA, 2001.

[18] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *ICDCS*, page 464, Washington, DC, USA, 2000. IEEE Computer Society.

[19] M. Wiesmann and A. Schiper. Beyond 1-Safety and 2-Safety for replicated databases: Group-Safety. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT2004)*, 2004.