On Optimizing Certification-Based Database Recovery Supporting Amnesia[⋆]

R. de Juan-Marín, M. I. Ruiz-Fuertes, J. Pla-Civera,
L. H. García-Muñoz, F. D. Muñoz-Escoí

Instituto Tecnológico de Informática, Universidad Politécnica de Valencia
Camino de Vera, s/n, 46022 Valencia (SPAIN)
e-mail: {rjuan,miruifue,jpla,lgarcia,fmunyoz}@iti.upv.es

Abstract. Certification-based database replication protocols are a good basis to develop replica recovery when they provide the snapshot isolation level. For such isolation level, no readset needs to be transferred between replicas nor checked in the certification phase. Additionally, these protocols need to maintain a historic list of writesets that is used for certifying the transactions that arrive to the commit phase. Such historic list can be used to transfer the missed state of a recovering replica. In this paper we design a basic recovery approach –to transfer all missed writesets– and a version-based optimization –to transfer the latest version of each missed item, compacting thus the writeset list–, and study and solve problems associated to the adoption of the crash-recovery with partial amnesia failure model.

1 Introduction

Performance, high availability and fault tolerance are key aspects in the information systems success. So, many computer science research efforts have been oriented to reach them, being replication the regular solution for achieving them.

A key aspect of these replicated systems is to recover or substitute crashed replicas. In this research area, database replication is a special kind of highly-available service since in this case replica recovery implies the application of the missed updates, being inefficient a complete state transfer since it needs a long time to be completed as has been detailed in many previous works as [1,2,3,4,5].

The aim of this paper is to design a replica recovery protocol for replicated databases which economises as much as possible recovery processes. As the recovery cost depends on several aspects, we have selected for the most important ones –failure model, replication protocol and recovery strategy– the techniques which can provide better behavior from a cost point of view when talking about large databases.

The failure model that we have adopted is the crash recovery with partial amnesia [6] instead of the fail-stop one, because the first one allows crashed replicas to be recovered while the second one forces to substitute them –being necessary to transfer the

[⋆] This work has been partially supported by FEDER and the Spanish MEC under grant TIN2006-14738-C02-01.

whole database–. Then, the first model gives us a more economic approach from the beginning.

In relation to the replication technique, we have tried to select the database replication kind [7] that provides the best support for developing an easy recovery: certification-based replication. In this replication variant, a historic list of the applied writesets needs to be maintained in order to certificate transactions (i.e., validate and locally decide in each replica about the success of each terminating transaction). Such a historic writeset list can be stored and used for transferring the missed updates to recovering replicas. Additionally, the resulting replication protocol does not need any voting termination [8] and provides very good performance if the conflicting rate is low [7]. Moreover, for the snapshot isolation level, a certification-based replication protocol is the natural solution, since it does not demand readset transfers. So, such kind of replication protocol provides an ideal basis to research on replica recovery and a basic recovery protocol can be easily developed.

But such a basic recovery protocol can be optimized in order to get better results. To this end, we have combined a version-based approach, similar to those proposed by other research groups (e.g., in some of the recovery variants of [5]) and in some of our previous papers [2,9] but specifically adapted to a certification-based replication protocol.

Moreover, we have studied in this paper the possible problems that can arise in the designed recovery protocol for assuming the crash recovery with partial amnesia failure model as it has been done in [10,11]. On one hand, we have analyzed the amnesia phenomenon support from the recovery information to transfer perspective. And on the other hand, we have studied the possible replication inconsistent state effects of combining the amnesia phenomenon with the selected progress condition and how they can be solved.

The rest of this paper is structured as follows. Section 2 presents the assumed system model. Section 3 describes the replication protocol taken as the basis for our recovery proposals. Section 4 thoroughly explains the recovery strategies. Section 5 discusses the amnesia support. Finally, Section 6 presents some related work and Section 7 gives the conclusions.

2 System Model

We assume a partially synchronous distributed system –where clocks are not synchronized but the message transmission time is bounded– composed by N nodes where each one holds a replica of a given database; i.e., the database is fully replicated in all system nodes. These replicas might fail according to the partial-amnesia crash failure model proposed in [6], since all already committed transactions are able to recover but ongoing ones are lost when a node crashes. We consider this kind of failures as we want to deal with node recovery after its failure.

Each system node has a local DBMS that is used for locally managing transactions. On top of the DBMS a middleware is deployed in order to provide support for replication. More information about our MADIS middleware can be found in [12,13]. This middleware has also access to a group communication service (abbreviated as GCS, on the sequel).

A GCS provides a communication and a membership service supporting virtual synchrony [14]. The communication service features a total order multicast for message exchange among nodes through reliable channels. Membership service provides the notion of view (current connected and active nodes with a unique view identifier). Changes in the composition of a view (addition or deletion) are delivered to the recovery protocol. We assume a primary component membership [14]. In a primary component membership, views installed by all nodes are totally ordered (there are no concurrent views), and for every pair of consecutive views there is at least one process that remains operational in both views. The GCS groups messages delivered in views [14]. The uniform reliable multicast facility [15] ensures that if a multicast message is delivered by a node (correct or not) then it will be delivered to all available nodes in that view. All these characteristics permit us to know which writesets have been applied in the context of an installed view. In this work, we use Spread [16] as our GCS.

We use a replication protocol based on certification [7], which does not require any kind of voting in order to decide how a transaction should be terminated (either committing or aborting).

This replication protocol transfers writesets, compound by pairs (*item identifier - value*) in order to propagate transaction updates, instead of spreading the corresponding SQL sentences that generated these updates. This form of update propagation is also used by the designed recovery protocol due to the adoption of a version-based approach.

3 Replication Protocol

We have selected the SIR-SBD protocol (see Figure 1) described in [13] for a case study of our recovery mechanisms, because it is a good sample of a certification-based [7] database replication protocol, providing the Generalized Snapshot Isolation level [17] and thus avoiding the transfer of transaction readsets.

This protocol uses an atomic multicast [15], i.e., a reliable multicast with total order delivery, and thus it ensures that the writesets being multicast by each replica at commit time are delivered in all replicas in the same order. It uses two in-memory data structures for dealing with writesets: `ws_list`, which stores all the writesets known (i.e., delivered) until now, and `tocommit` queue, which holds those writesets locally certified but not yet applied in the local database replica. Moreover, for each transaction, the attributes `start` and `end` hold something similar to the transaction start and commit timestamps, respectively. Due to the total order multicast and the behavior of the protocol, the second counter is the same for a system transaction in all the replicas, i.e., all the replicas identify with the same commit timestamp a system transaction.

Note that we have tacitly assumed that the underlying database system is supposed to be able to check for conflicts, and to abort transactions whose access patterns violate the snapshot isolation level rules.

This protocol is also based on the existence of a block detection mechanism [13]. We have assigned the following priorities to the transactions. All transactions are initialized with a 0 priority level. They get level 1 when they are multicast in their local node or

```
Initialization:
  1. lastvalidated_tid := 0
  2. lastcommitted_tid := 0
  3. ws_list := $\emptyset$
  4. tocommit_queue_k := $\emptyset$
I. Upon operation request for $T_i$ from local client
  1. If select, update, insert, delete
    a. if first operation of $T_i$
      - $T_i$.start := lastcommitted_tid
      - $T_i$.priority := 0
    b. execute operation at $R_k$ and return to client
  2. else    /* commit */
    a. $T_i$.WS := getwriteset($T_{ik}$) from local $R_k$
    b. if $T_i$.WS = $\emptyset$, then commit and return
    c. $T_i$.priority := 1
    d. multicast $T_i$ using total order
```

```
II. Upon receiving $T_i$ in total order
  1. obtain wsmutex
  2. if $\exists$ $T_j$ $\in$ ws_list : $T_i$.start $<$ $T_j$.end $\wedge$
     $T_i$.WS $\cap$ $T_j$.WS $\neq \emptyset$
    a. release wsmutex
    b. if $T_i$ is local then abort $T_i$ at $R_k$ else discard
  3. else
    a. $T_i$.end := ++lastvalidated_tid
    b. append $T_i$ to ws_list and tocommit_queue_k
    c. release wsmutex
III. $T_i$ := head(tocommit_queue_k)
  1. if $T_i$ is remote at $R_k$
    a. begin $T_{ik}$ at $R_k$
    b. apply $T_i$.WS to $R_k$
    c. $\forall$ $T_j$ : $T_j$ is local in $R_k$ $\wedge$ $T_j$.WS $\cap$ $T_i$.WS $\neq \emptyset$
       $\wedge$ $T_j$ has not arrived to step II
       (this is analyzed by our conflict detector,
       concurrently with the previous step III.1.b)
       - abort $T_j$
  2. commit $T_{ik}$ at $R_k$
  3. ++lastcommitted_tid
  4. remove $T_i$ from tocommit_queue_k
```

Fig. 1. SIR-SBD algorithm at replica R_k

when their writeset is delivered in their remote nodes. This ensures the correctness of this alternative, since our blocking detection mechanism aborts a transaction only if all of these conditions are satisfied. Otherwise, no particular action is taken:

- The transaction to be aborted is local.
- It has not locally requested its commit; i.e., its writeset has not been multicast.
- The transaction that causes its abortion has been generated for applying a remote writeset.

This approach satisfies the correctness criteria of the snapshot isolation level, since the writeset above mentioned is associated to a transaction that has successfully passed its global validation phase. It already has a commit timestamp which of course is in the range of the [start, commit] interval of the local transaction, since the latter has not yet requested its commit.

An important replication protocol detail is that the *ws_list* in each node maintains the writesets necessary to perform the certification work at this replica and the writesets that have not been processed by other replicas. A writeset only can be deleted from the list once it has been processed by all replicas –being then not necessary for the certification work–. This deleting policy has a key aspect for recovery purposes.

4 Recovery Strategies

We describe a basic recovery in Section 4.1 and its optimized version in Section 4.2. The optimization consists in compacting the list of missed writesets, maintaining only the last version of each missed item.

4.1 Basic Recovery

As a general overview of the main goal of our recovery protocol, let us say that one node (recoverer) will transfer the missed writesets to the recovering node arranged by their respective versions. This means that user application transactions executed on the recovering node will run under GSI [17] in a slower replica. As it may be seen there are no restrictions to execute user transactions in the replica and transactions executing at other replicas will behave as if nothing happens in the system. To achieve this we take the ideas outlined in [17].

A recovering replica R_i joins the group, triggering a view change. As part of this procedure, the recovering protocol instance running in R_i multicasts in total order an *ask-for-help* message indicating the $version_i$ of its last applied writeset –this version corresponds to the commit timestamp of the last transaction applied in that node. No message activity in the recovering node is done –all delivered messages are ignored– until this message is delivered. At this moment, the recovering node starts to enqueue the total order delivered messages –with writeset information about other transactions in the system sent by the rest of the replicas– to be processed later.

In parallel to this, a deterministic procedure takes place to choose a recoverer replica. The recoverer replica (R_j), after receiving the *ask-for-help* message, starts a recovery thread that sends a point-to-point message with all the missed writesets starting from $version_i + 1$, i.e., the recoverer node sends the portion of its `ws_list` that covers from $version_i + 1$ to the end of the `ws_list` at that moment. Note that this `ws_list` is one of the elements on which the replication protocol algorithm is based, and it can also be used for our recovery purposes, as it contains all the information we need. This way, we reuse the data maintained by the replication protocol, minimizing the overhead introduced by the recovery support in normal operation (i.e. no additional data collection is needed to support possible recovery processes). Note also that this approach guarantees that the recovering node will receive, on one hand, the writesets from $version_i + 1$ to the last known version of the recoverer at the moment of the *ask-for-help* message reception. On the other hand, the writesets delivered in total order in the system after the *ask-for-help* message will be enqueued in the recovering message buffer. This way, it is ensured that the recovering node will not miss any writeset.

When this point-to-point recovery message is delivered to the recovery protocol, it stores this information in both the `ws_list` and the `tocommit` queue, as all these writesets were already certified in the recoverer node. Then, the replication protocol is ready to directly apply in the database the writesets in the `tocommit` queue and to start certifying its own enqueued total order messages –delivered after the *ask-for-help* message. Note that the certification of the enqueued messages must wait for the recovering information to be stored in the `ws_list`, as this structure is used in the certification process, but it is not necessary to wait to the application of these missed writesets in the database. In other words, just after the storage of the transmitted writesets in both data structures, the recovering node can act as in normal mode.

4.2 Compacting

This basic procedure can be enhanced by compacting the point-to-point message in order to minimize the transmission and application time. The point-to-point recovery message has to provide all the changes in the database made from $version_i + 1$ to the current version of the recoverer node. This information can be sent in a raw mode, i.e., sending the writesets of all the transactions committed during this period of time. Then, in the recovering node, all writesets are applied in the context of a single transaction. This is the way used in the basic recovering protocol explained before.

All this procedure can be enhanced if the recoverer replica elaborates a special writeset composed by the last version of each modified object in all the transactions committed during the crash time, i.e., if the same object was modified by more than one transaction, only the last version of it would be transmitted along with its corresponding `end` timestamp. This special writeset, built only in the recoverer replica at recovery start time, would be applied by the recovering replica in a single transaction, which can greatly improve the committing time, not only for being just one –although possibly big– transaction, but also for avoiding useless updates of the same object. This way, compacting will reduce both the transmission and the checking time. The time needed by the recoverer node to prepare this compacted message is not negligible, but it does not imply any noticeable overhead.

Note also that the regular function of the replication protocol is not compromised by this optimization. Indeed, the recovering replica can start processing transactions immediately. The writesets transferred in the recovery message are not needed by the recovering replica in order to certify any new local writeset, since such new writesets should be certified against the writesets regularly delivered in the new view in which such recovering replica has rejoined the group. However, such compacted writesets can be needed for certifying remote transactions in such recovering replica, but its compacted version is enough for such kind of certification. Note that can exist long remote transactions that have started before the recovery process started, and their [start, commit] interval might overlap the end of some transactions included in the compacted missed writesets. Since at least the latest version of each missed updated item is present in such compacted set, all conflicts detectable with the original writeset list will be detectable with such compacted sequence. For instance, assume that there were N transactions T_1, T_2, ..., T_N in the original missed writeset list and that each writeset contained M items a_{11}, a_{12}, ..., a_{1M}, ..., a_{N1}, a_{N2}, ..., a_{NM}, and each of these transactions has a consecutive logical commit timestamp (t_i for T_i, being t_{i+1}=$t_i + 1$). Without generalization loss, let us assume that there are only M/2 items per writeset that have not been updated in any of its successive writesets (except in the last writeset of such compacted list that is the single one that cannot be compacted –their updated items are their trivially latest versions in the recovery transfer set–), being a_{iK_i} those items (where $K_i \subset \{j \in \mathbb{N} : 1 \leq j \leq M\}$ and $|K_i| = M/2$). So, if a given "future" transaction T_j was started between, e.g. T_1 and T_2, its writeset should be checked against all writesets in the range [T_2, T_{j-1}]. Note that T_j has been terminated after T_N, and as a result of this, all items updated by all transactions in the range [T_2, T_{j-1}] are also included in its "compacted variant" since $N < j - 1$, and our compacting process guarantees that only items rewritten during the [T_1,T_{N-1}] interval are removed from the T_1..T_N writeset sequence (but if any item has not been rewritten, it appears in such compacted sequence, and this guarantees that exactly one version of all original writeset elements appears in the compacted version). Additionally, we have the advantage of a boost in the checking time, since instead of having the complete sequence of [T_1, T_N] writesets, we only have a compacted item sequence $a_{1K_1}..a_{NK_N}$, as assumed above (i.e., half of the items, in this hypothetical example).

Our optimization shares some of the characteristics of the fifth recovery strategy described in [5] ("Restricting the set of objects to check") but further optimizes that technique. To this end, our compacting is able to restrict the objects being checked without needing any additional table where the objects are being recorded during the crash interval. Additionally, it still shares the advantage of getting such set of items to be transferred without requiring any read lock nor global read operation on the items stored in the regular database tables. But, on the other hand, it is partially dependent on the replication protocol approach (certification-based), and can not be easily adapted to all other database replication variants (e.g., the active and weak-voting variants [7] do not need any historic writeset log).

5 Amnesia Support

As we have said in Section 2, replicas fail according to the partial-amnesia crash failure model. This assumption allows crashed replicas to be recovered when they reincorporate to the replicated database system. Therefore, in order to update them it is only necessary to transfer them the changes they have missed during their disconnection time. This characteristic is a great advantage compared to the fail-stop failure model –which forces to transfer the whole database– when we are talking about great databases as explains [10]. The basic idea is that the recovery process must transfer less information, implying smaller recovery times, and so:

- shorter periods with low replicated system performance due to the overhead introduced by the recovery process work,
- shorter periods with decreased fault tolerance support because outdated replicas are updated in a faster way –only fully updated replicas can be used to guarantee the correct and consistent state evolution in the replicated system–.

But, adopting this failure model presents several problems as described in [10]. In fact, it can lead to replicated inconsistencies in two different scenarios if they are not correctly managed by the combination of the replication and recovery protocol used by the replicated database.

In the following subsections we will present these two scenarios and study if our solution –replication and recovery protocol combination– manages them correctly.

5.1 Recovery Information to Transfer

Firstly, the recovery protocol must ensure that the state in the recovering node after the recovery process is consistent. In this scenario, consistent means that the recovered node reaches the replicated system state. Thus, the replicated system must know which is the last consistent state reached by the crashed node, in order to transfer the exact needed information, avoiding the problems of losing some changes –which would lead to an inconsistent state in the recovered node–. Notice that applying changes twice or more in a version-based recovery strategy as ours –where are transferred the values and not the operations– does not imply inconsistencies, simply is redundant work.

This problem arises in those replicated database systems whose crashed members can not remember exactly which was their last state before their crash occurrence. In other words, it can be said that when nodes crash they expect to have a theoretic last state that can differ from their real last state due to amnesia. This happens in those protocols which use atomic broadcast, and virtual synchrony, and rely only on their message order delivery, making the assumption that all delivered messages are correctly processed. But, as it has been demonstrated in [18] this assumption is not correct. It is possible that some delivered transactions have not been processed before the node crashed, therefore when this node reconnects it has a gap between its last delivered transaction and its last committed one.

Our replication proposal avoids this problem because does not rely on this assumption. In fact, both recovery protocols, the basic one described in subsection 4.1 and the compacted one detailed in subsection 4.2, use the information of the last applied writeset in the node being recovered –$version_i$, the commit timestamp of its last committed transaction–. Then when a node reconnects –needing to be updated– it multicasts this information which is used by the recoverer replica to determine which subset of database information must be transferred for updating the recovering replica. This is possible because each node marks which is its last committed transaction. Thus, our recovery protocols deal with this problem correctly, ensuring that it will not be any gap in the recovery information transferred avoiding then possible diverging state evolution in recovered replicas.

Now, after describing how our recovery protocols handle accurately one of the possible problematic scenarios we will study the second one.

5.2 Amnesia and Progress Condition

The second scenario that can lead to replicated state inconsistencies –an undesired situation– appears when combining the amnesia problem with the replicated system progress condition –primary partition–. The problem is that the system is unable to guarantee the correct system data state progress. We will explain this scenario with an example.

Consider that we have a replicated system, $\alpha = \{R_1, R_2, R_3\}$, compound by three replicas. All three replicas are fully updated and working at the instant t_0, being the last delivered and committed transaction in all nodes T_1. Then the replicated system delivers T_2 to all replicas, which is committed in R_1 and R_2 but not in R_3 because it crashes before being able to commit it, moreover, R_3 loses the T_2 writeset because the replication protocol does not persist it. And R_3 crash triggers the installation of a new view which still fulfils the primary partition progress condition, therefore R_1 and R_2 can go on working. At this time the replicated system delivers T_3 to the currently alive nodes R_1 and R_2, but this transaction is only committed in R_1 because R_2 crashes before committing it –losing the T_3 writeset. Then a new view is installed, but as it is compound only by R_1 it does not fulfil the progress condition, so the replicated system stops working. At this point, R_2 reconnects to the system, and a new view compound by R_1 and R_2 is installed. Therefore, the recovery process of R_2 starts –it has lost T_3 updates–. But R_1 crashes before recovering R_2 –before sending T_3 updates–. Then the system installs a new view which does not fulfil the progress condition, so the system can not work. Later, R_3 reconnects to the system, triggering the installation of a new view which fulfils the progress condition –R_2 and R_3 are alive–. But then arises a consistency problem, R_2 can update R_3 with the changes associated to T_2 but no one has the changes belonging to T_3. Therefore, as they fulfil the progress condition they can go on working, but if they work they will start from the state reached after committing T_2 and not T_3 –the last really committed transaction in the replicated database system– leading to a diverging state evolution to R_1 state –which is the correct one–.

First of all, it must be said that this situation or another events combination that leads to a similar situation is very improbable. And this probability diminishes as long as the number of replicas increases.

Anyway, our proposals as they are currently defined do not handle this problem accurately, and improbable does not mean impossible, so these situations must be avoided. The way for overcoming this problem is the same one proposed in [19] in order to adopt the majority of alive nodes progress condition, or the generic way for solving the amnesia problem presented in [10] in log-based recovery strategies for transactional systems.

The idea consists in storing persistently the delivered messages in each replica as an atomic step of the delivery message, being only possible to delete them as soon as they have been correctly processed by the replica and they are not necessary for the certification phase, in other words to persist the *ws_list*. Therefore, in the above described sequence of events, when R_2 crashes it will have not committed T_3 but it will have not lost its associated writeset because it would be persistently stored. Then, in the last reached view, compound by R_2 and R_3, the replicated database system would have the necessary information to apply T_3 in both replicas in the recovery process, allowing the system to go on working without generating replicated inconsistent states.

This work way when included in the general recovery protocol will alter it in a slight way. This evolution implies that as soon as a crashed node reconnects it starts a stage of self recovery, which implies to process all the persistently stored messages which have not been correctly processed yet. It must be noticed that this persistence policy wraps all the necessary information for certification purposes –broadcast messages–, ensuring a correct transaction execution. Once the node being recovered finishes this step it can go on using the original recovery protocols, sending the $version_i$ of its last committed transaction after the self recovery process.

Obviously, persisting messages as soon as they are delivered implies an overhead during the replication work. A study of the associated overhead cost is presented in [10]. An overhead that will penalize constantly the replication work in order to avoid problems for situations that will rarely occur.

Therefore, another possible solution is to do nothing and assume these situations can happen in our replicated system. In this case, the idea is that among the alive nodes that compound the new primary partition –instead of not having the last consistent state– decide a new last consistent replicated state, allowing the system to go on working from this point. Later, when a replica which really reached the last consistent state of the replicated system reconnects, it must undo the transactions not processed in the new consistent replicated state before being recovered. Then, in the previous example, R_2 and R_3 would be able to go on working, taking as starting consistent state the one reached after applying T_2. And later, when R_1 reconnects it would have to undo T_3 before recovering the updates performed by R_2 and R_3. This solution avoids the overhead of persisting messages and simply implies to undo –in very rare occasions– some transactions –usually very few–. This solution is similar in concept to some approaches used in reconciling processes for partitionable systems.

Which solution must be adopted? It depends. The first solution solves the problem ensuring that committed transactions are not lost, but implies a constant overhead during the normal work for solving a problem that will rarely happen. While the second solution avoids the problem without implying any overhead, but some transactions must be undone when this improbable scenario happens. So, it depends on the replicated system tolerance to undo some already committed transactions. If this tolerance is critical we have to select the first approach, while if there are not important problems of undoing some committed transactions, the second one can be adopted.

6 Related Work

The use of version-based recovery protocols –the same approach taken as the basis of our proposed optimization– had been already suggested in the fourth and fifth recovery variants of [5], but in both cases still demanded a lot of effort for maintaining the set of versions to be transferred to each crashed replica. Either a version-based DBMS was assumed or a special additional table needs to be managed and updated each time a transaction commits. We used the latter solution in [2,9] but in both papers such protocols were designed as a recovery approach for a replication system that did not provide any standard isolation level. Those solutions were developed in our COPLA system [20], and such middleware was targeted to provide an object-relational translation, with an object-oriented programming interface where the traditional isolation levels did not match. In the current paper, we have optimized the version-based approach taking as its basis a certification-based [7] replication variant in order to support the snapshot [21] isolation level.

A similar compacting technique has been also applied in [22], in order to optimize the recovery protocol presented in [23] but in this case it used view granularity and not transaction granularity.

In regard to the failure model adopted, process replication has traditionally been oriented to the fail-stop failure model as in [24], while replicated transactional systems which manage large data amounts as databases have oriented to the crash-recovery with partial amnesia as [1,2,5]. In this last scenario few protocols have adopted the fail-stop model as [25].

Thus all these recovery protocols that have adopted the crash-recovery with partial amnesia failure model have to solve the amnesia problem. But, instead of this fact there are few recovery protocols that have analyzed the problem in a deep way.

In [18] authors noticed that message delivery does not imply message processing, situation that, as it has been seen in this paper, can lead to consistency problems after recovery processes. For solving this problem [18] presented the *successful delivery* concept.

This problem has been also studied in the context of log-based recovery techniques as in [10,11]. In this context, [2] proposes a solution for the amnesia phenomenon in log-based strategies based on logging received messages. [4] presents a different way to deal with the amnesia problem in similar scenarios. In this paper the authors mix checkpointing with message transfer; in other words, they combine as recovery information to transfer a snapshot of the data state and the needed messages from the snapshot instant point. This combination allows the protocol to overcome the amnesia problem. A possible drawback of this solution depends on the way in which the checkpoint process is performed, because if the whole data state is transferred, the benefits of adopting the crash-recovery failure model are lost.

In [5] several version-based protocols are proposed. Without considering the one consisting in transferring the whole database, it proposes the approach of checking version numbers. This recovery protocol manages correctly the amnesia problem because it marks each data object with the identifier of the last transaction that updated it –even when there are not failed nodes– as our proposals do. And they optimized it presenting a new recovery protocol which restricts the set of objects to check. This protocol maintained in a database table *–reconstruction 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 not, the protocol created a new entry with the object identifier and the transaction identifier. If it already existed an entry with this object identifier, the protocol simply updated in this entry the transaction identifier. But the problem of this new protocol is that it does not handle accurately the amnesia problem in transitions from a view without failed nodes to one with failed nodes. In this case, it is possible that the node whose crash triggered the new view has delivered before crashing one transaction without processing it correctly, so it loses its changes. The problem arises because the other nodes consider that this transaction belongs to the view where all nodes were alive, and therefore they do not store its changes in the reconstruction table, being afterwards impossible to transfer these updates in the recovery process.

In the version-based area a similar study of amnesia support, presented in [22], has been performed for the recovery protocol described in [23]. In this case the broadcast messages were also forced to be stored persistently in order to avoid the amnesia problem.

As far as we know, few studies have been performed considering the effects derived from combining the adopted progress condition in the replicated system, the replication and recovery protocols used. The papers [11,19] perform such study for replicated transactional systems which uses log-based recovery techniques.

7 Conclusions

We have presented a first basic recovery approach for certification-based replication protocols. Although certification-based replication protocols provide a good basis for developing recovery protocols, this first basic approach can be easily improved. A possible optimization based on a missed updates compacting has also been presented.

We have also detailed the advantages of using the crash-recovery with partial amnesia failure model instead of the fail-stop when talking about large databases. These reasons have led us to adopt this system failure model in our replicated database system from the beginning. But this assumption can cause different problems, therefore we have also included a study of the amnesia support provided by our certification-based recovery proposals, analyzing the associated problems, studying how they are managed by our protocols, and proposing enhancements when needed.

References

1. Armendáriz-Iñigo, J.E.: Design and Implementation of Database Replication Protocols in the MADIS Architecture. PhD thesis, Universidad Pública de Navarra, Pamplona (Spain) (2006)
2. Castro, F., Esparza, J., Ruiz, M.I., Irún, L., Decker, H., Muñoz, F.D.: CLOB: Communication support for efficient replicated database recovery. In: PDP, Lugano, Switzerland, IEEE-CS Press (2005) 314–321
3. Holliday, J.: Replicated database recovery using multicast communication. In: NCA, IEEE-CS Press (2001)
4. Jiménez, R., Patiño, M., Alonso, G.: An algorithm for non-intrusive, parallel recovery of replicated data and its correctness. In: SRDS, IEEE-CS Press (2002) 150–159
5. Kemme, B., Bartoli, A., Babaoglu, O.: Online reconfiguration in replicated databases based on group communication. In: IEEE Int. Conf. on Dependable Systems and Networks, Göteborg, Sweden (2001) 117–130
6. Cristian, F.: Understanding fault-tolerant distributed systems. Commun. ACM **34**(2) (1991) 56–78
7. Wiesmann, M., Schiper, A.: Comparison of database replication techniques based on total order broadcast. In: IEEE Trans. Knowl. Data Eng. 17(4). (2005) 551–566
8. Wiesmann, M., Schiper, A., Pedone, F., Kemme, B., Alonso, G.: Database replication techniques: A three parameter classification. In: SRDS. (2000) 206–217
9. Castro, F., Irún, L., García, F., Muñoz, F.D.: FOBr: A version-based recovery protocol for replicated databases. In: PDP, Lugano, Switzerland, IEEE-CS Press (2005) 306–313
10. de Juan-Marín, R., Irún-Briz, L., Muñoz-Escoí, F.D.: Supporting amnesia in log-based recovery protocols. In: Euro-American Conference On Telematics and Information Systems (EATIS 2007), Faro, Portugal. (2007)
11. de Juan-Marín, R., Irún-Briz, L., Muñoz-Escoí, F.D.: Recovery strategies for linear replication. In: ISPA. (2006) 710–723
12. Irún, L., Decker, H., de Juan, R., Castro, F., Armendáriz, J.E., Muñoz, F.D.: MADIS: a slim middleware for database replication. In: 11th Intnl. Euro-Par Conf., Monte de Caparica (Lisbon), Portugal (2005) 349–359
13. Muñoz-Escoí, F.D., Pla-Civera, J., Ruiz-Fuertes, M.I., Irún-Briz, L., Decker, H., Armendáriz-Iñigo, J.E., González de Mendívil, J.R.: Managing transaction conflicts in middleware-based database replication architectures. In: SRDS, IEEE-CS Press (2006) 401–410
14. Chockler, G.V., Keidar, I., Vitenberg, R.: Group communication specifications: A comprehensive study. ACM Computing Surveys **33**(4) (2001) 427–469
15. Hadzilacos, V., Toueg, S.: Fault-tolerant broadcasts and related problems. In Mullender, S., ed.: Distributed Systems. 2nd edn. ACM Press (1993) 97–145

16. Spread: The Spread communication toolkit. Accessible in URL: http://www.spread.org (2007)
17. Elnikety, S., Pedone, F., Zwaenepoel, W.: Database replication providing generalized snapshot isolation. In: 24th IEEE Symposium on Reliable Distributed Systems, Orlando, FL, USA (2005) 73–84
18. Wiesmann, M., Schiper, A.: Beyond 1-Safety and 2-Safety for replicated databases: Group-Safety. In: 9th International Conference on Extending Database Technology. (2004) 165–182
19. de Juan-Marín, R.: (n/2+1) alive nodes progress condition. In: Sixth European Dependable Computing Conference, EDCC-6. (2006)
20. Esparza, J., Calero, A., Bataller, J., Muñoz, F., Decker, H., Bernabéu, J.: COPLA: A middleware for distributed databases. In: 3rd Asian Workshop on Programming Languages and Systems (APLAS '02). (2002) 102–113
21. Berenson, H., Bernstein, P.A., Gray, J., Melton, J., O'Neil, E.J., O'Neil, P.E.: A critique of ANSI SQL isolation levels. In: SIGMOD, ACM Press (1995) 1–10
22. García-Muñoz, L.H., de Juan-Marín, R., Armendáriz, J.E., Muñoz-Escoí, F.D.: Adding amnesia support and compacting mechanisms to replicated database recovery. Technical report, ITI-ITE-07/08, Instituto Tecnológico de Informática (2007)
23. Armendáriz, J.E., Muñoz, F.D., Decker, H., Juárez, J.R., de Mendívil, J.R.G.: A protocol for reconciling recovery and high-availability in replicated databases. 21st International Symposium on Computer Information Sciences, Springer **4263** (2006) 634–644
24. Birman, K.P., Renesse, R.V.: Reliable Distributed Computing with the ISIS Toolkit. IEEE Computer Society Press, Los Alamitos, CA, USA (1993)
25. Lau, E., Madden, S.: An integrated approach to recovery and high availability in an updatable, distributed data warehouse. In: VLDB. (2006) 703–714