Optimizing Certification-Based Database Recovery[*]

J. Pla-Civera, M. I. Ruiz-Fuertes, 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: {jpla,miruifue,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. We study the performance of the 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 the results show that such optimization reduces a lot the recovery time.

1 Introduction

Replication has been the regular solution for achieving high availability. But such level of availability requires that crashed replicas were recovered. 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. Even transferring only the missed updates, there is no easy way to complete such recovery in a short time.

There have been many good works devoted to database replication recovery [1, 3, 10, 13, 14], but almost none of them has provided a rigorous performance study of the proposed approaches. The aim of this paper is to show that replica recovery is not easy when the load of the replicated system is not light, and that some optimizations can partially overcome such problem. To this end, we have tried to select the database replication kind [19] 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 [20] and provides very good performance if the conflicting rate is low [19]. 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 does not provide good performance (i.e., a short recovery time). So, some optimizations are needed in order to get acceptable 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 [14]) and in some of our previous papers [3, 4] but specifically adapted to a certification-based replication protocol. Such optimization introduces a negligible overhead and shortens the recovery time, as shown in Section 5.

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 performance results. Finally, Section 6 presents some related work and Section 7 gives the conclusions.

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

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 on-going 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, 15]. This middleware also has access to a group communication service (abbreviated as GCS, on the sequel).

A GCS provides a communication and a membership service supporting virtual synchrony [5]. The communication service features a total order multicast for message exchange among nodes through reliable channels. Membership services provide 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 [5]. 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 [5]. The uniform reliable multicast facility [9] 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 [18] as our GCS.

We use a replication protocol based on certification [19], which does not require any kind of voting in order to decide how a transaction should be terminated (either committing or aborting).

3 Replication Protocol

We have selected the SIR-SBD protocol (see Figure 1) described in [15] for a case study of our recovery mechanisms, because it is a good sample of a certification-based [19] database replication protocol, providing the snapshot isolation level [2] and thus avoiding the transfer of transaction readsets.

This protocol uses an atomic multicast [9], 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 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 –this will be handy when studying the performance graphs.

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

```
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
```

Figure 1. SIR-SBD **algorithm at replica R_k**

detection mechanism [15]. 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 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.

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 [7] 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 it nothing happens in the system. To achieve this we take the ideas outlined in [7].

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 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 messages delivered 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.

This kind of recovery inherits the main ideas of the second approach described in [14] ("Data transfer within the database system") and, up to our knowledge, had been already implemented and studied in other projects (e.g., GlobData, in order to add recovery capacity to the protocols presented in [17], but its performance results were only described in an internal project report).

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, each writeset is applied in a new transaction –like any other replica does in normal function. 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 as we will see later in the performance results. The time needed by the recoverer node to prepare this compacted message is not negligible, but we will see in the graphs that 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 [14] ("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 [19] do not need any historic writeset log).

5 Performance Study

In this work we intend to measure several aspects of our recovery implementation: (a) under which circumstances (work load and crash length) a failed node can recover and reach the state of the other replicas; (b) how long does it take to reach the state of the other replicas; and (c) compacting impact.

To accomplish the comparison, we use PostgreSQL [16] as the underlying DBMS, and a database with a single table with two columns and 10000 rows. One column is declared as primary key, containing natural numbers from 1 to 10000 as values.

All protocols have been tested using our MADIS middleware with 4 replica nodes. Each node has an AMD Athlon(tm) 64 Processor at 2.0 GHz with 2 GB of RAM running Linux Fedora Core 5 with PostgreSQL 8.1.4 and Sun Java 1.5.0. They are interconnected by a 1 Gbit/s Ethernet. In each replica, there is a varying number of concurrent clients (from 4 to 12). Each client executes an endless stream of sequential transactions, each one accessing a fixed number of 20 items for writing, with a fixed pause of 500 ms between each consecutive transaction. Each test begins with the execution of 500 global transactions, after that, a failure occurs in a random replica (the failure of a replica consists in the termination of its process). The failure lasts for a period in which a varying number of global transactions is executed by the other replicas. After this time, the failed node restarts and begins the recovery process until it reaches the state of any of the other replicas. The test continues once the recovery ends, until the completion of 500 more global transactions, when the experiment finishes.

In the figures we show the evolution of nodes in committing transactions in the system. All the transactions have a global identifier –the end counter– and must be committed locally in each replica. This way, one global transaction requires a local transaction in each replica, and we can know how quick a node goes by seeing the last committed global identifier at that node (see the vertical axes in both Figure 2 and Figure 3). This way, each graph shows this evolution in three nodes in the system: the failed, the recoverer and another node. The bigger the slope of that curve, the faster the node goes committing global transactions.

The results obtained show that the basic recovery technique was very poor in comparison with the compacting approach. The graphs included show the results when both recovery techniques allow the immediate start of new local transactions in the recovering replica.

The results obtained without compacting (see Figure 2.a) and light load show that the recovering node can easily reach the current state of the system. We can see in the figure that all the replicas have a linear evolution and when the failure occurs, the failed node does not make any advance –and so its line is horizontal. Then, when the recovery process begins, the recovering node starts to progress with more slope than the other nodes, i.e., it commits more transactions per second, and thus it can reach the global state and continue with the same previous linear behavior.

With medium and heavy loads (Figures 2.b and 2.c) the recovery trend is too slow and the recovering replica is not able to cope with its intended work. Note that our MADIS middleware is not a commercial prototype and performance is not our main goal. So, the load parameters considered are relative to the current capabilities of our middleware. This way, with the previously described conditions and 12 clients per node, the system has to deal with near 20 transactions per second, which is almost our middleware saturation point. Because of this, this load parameter combination has been called heavy.

Obviously, the recovering node can catch the other replicas as long as the current load in the system provides enough idle time (due to no clients to attend) to apply the missed writesets. Local transactions starting in the recovering node during the recovery process are very likely to abort because of their outdated snapshot. These local transactions will delay the application of the missed writesets due to the conflicts arisen in the underlying database. Thus, the lighter the load of local transactions in the recovering node, the faster these missed writesets will be applied. To sum up, the system load affects in two ways: it determines the idle time available to reduce the gap with the other replicas, and also the amount of local transactions in the recovering node possibly delaying the application of these missed writesets.

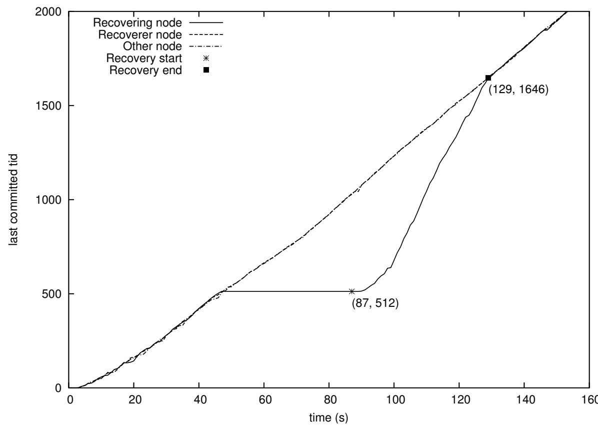
In the light environment, replicas have an important amount of this idle time and the recovering node has no problem to catch the rest. In the medium one, the replicas have little idle time, and the amount of local transactions make impossible to reduce the gap. And in the heavy case, there is no idle time and local transactions even broaden this gap.

As it can be seen in the graphs, the compacting technique (see Figure 3) allows the failed replica to quickly achieve a state close to those of the other replicas. In all the tests, the recovering replica is able to do so without too much delay. Specifically, when the load and the crash time are small, we can observe that the evolution of the recovering node after the crash is not as progressive as in the basic technique, but it has two phases. The first one is a big step towards the global state due to the application of the compacted writeset; and the second, the final evolution during the application of the enqueued messages delivered in the meanwhile. Comparing with the previous basic technique, it can be noticed that the recovery process lasts quite less with the compacting approach. Indeed, the recovery time is only 30 seconds with this approach, while it was 42 seconds with the basic recovery; i.e., almost a 28.6% reduction.

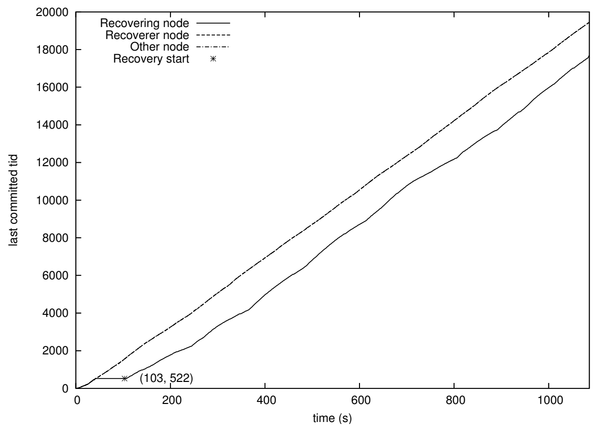
The next graph (Figure 3.b) shows the behavior of the system when both the load and the crash time are medium. The shape of the curves is similar to that of a light load and the recovery process is much more faster than with the basic technique. In the example provided in our figures, the recovery takes only 72 seconds with the compacting optimization, whilst the recovery was not possible using the basic recovery strategy.

Finally, when the load and crash time are maximum (Figure 3.c), the optimized technique increases its completion time, as expected, but it is still able to complete the recovery in an acceptable time (92 seconds).

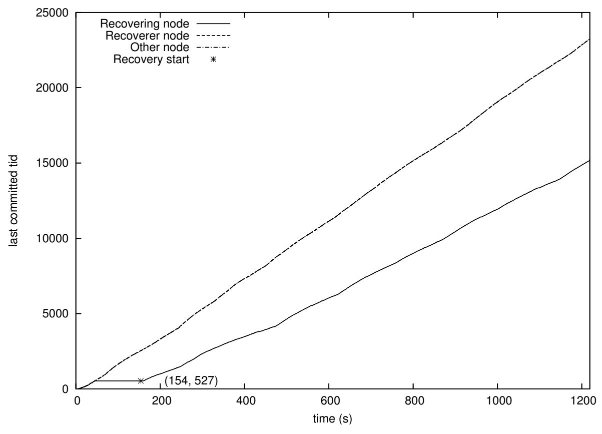
Note that in these two last cases –with a medium and heavy load–, the time needed for completing the recovery is quite long –72 and 92 seconds, respectively–, but the non-recovering replicas and the recoverer one have been able to process new transactions at a regular pace; i.e., their availability is not compromised by the recovery of another replica. Additionally, the recovering replica has accepted new transactions as soon as possible, and this introduces a



a) Light load (4 clients per node), short crash (500 transactions).

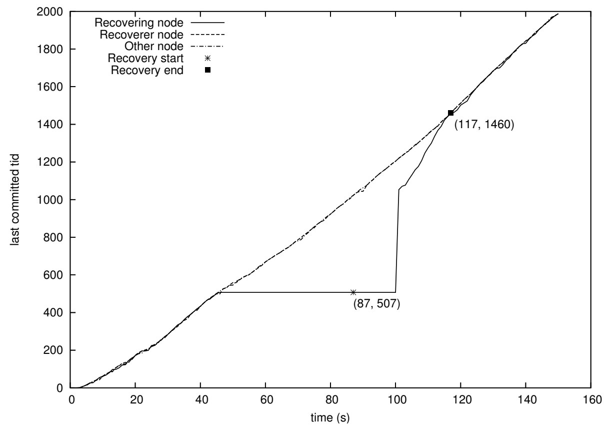


b) Medium load (8 clients per node), medium crash (1000 transactions).

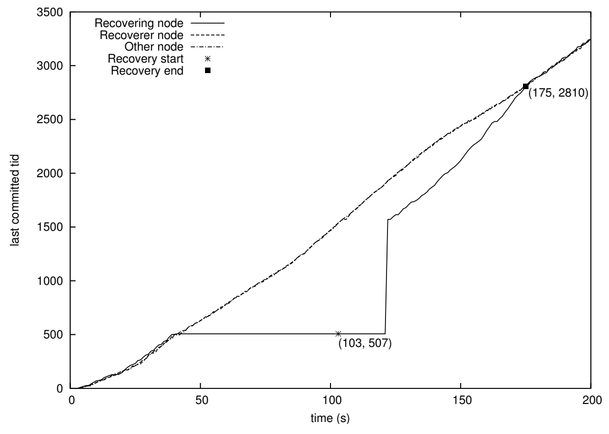


c) Heavy load (12 clients per node), long crash (2000 transactions).

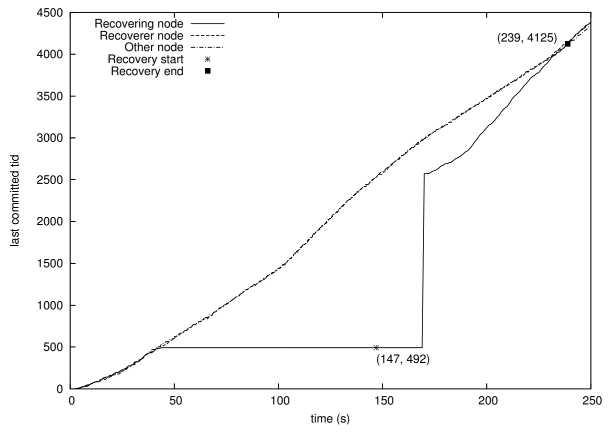
Figure 2. Recovery without compacting



a) Light load (4 clients per node), short crash (500 transactions).



b) Medium load (8 clients per node), medium crash (1000 transactions).



c) Heavy load (12 clients per node), long crash (2000 transactions).

Figure 3. Recovery with compacting

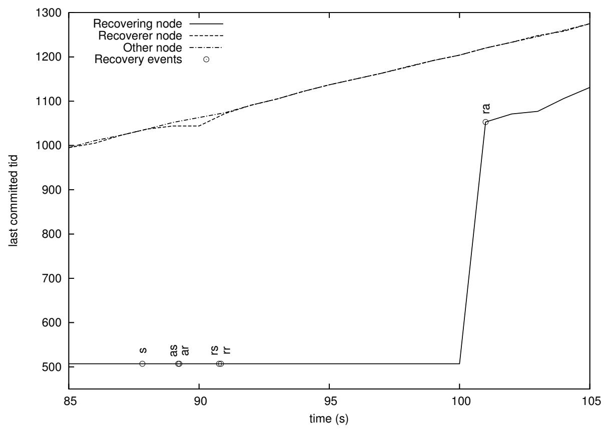


Figure 4. Recovery events

non-negligible delay in its recovery, but also shortens a lot the interval where the service in such replica is not available.

More tests were performed disabling the start of new local transactions in the recovering replica in order to quantify the improvement when both techniques achieve the completion of the recovery process. The results show that, in a medium loaded environment the compacting technique achieves the completion of the process in 36 seconds while the basic one lasts 59 seconds (38.98% of improvement). In the heavy loaded environment case, the reduction obtained is up to 60.82% (97" versus 38"). Note that the compacting technique, even allowing new local transactions in the recovering node, is faster than the basic one without these new local transactions.

In order to analyze more deeply the cost related to each step of the recovery process, Figure 4 has been included showing the recovery events from the starting point of the recovery process to the compacted writeset application in the case of a light load environment. These events are the following: *s*, the starting point of the recovery (when the failed replica restarts); *as*, the *ask-for-help* message is sent by the recovering node; *ar*, this *ask-for-help* message is received in the recoverer node; *rs*, the recoverer node sends its compacted point-to-point message; *rr*, this point-to-point message is completely received and preprocessed in the recovering node (the information contained is provided to the necessary components); *ra*, the recovering node finishes the application of the writesets contained in the recovery message. At this point, the recovery algorithm is done, but the recovery process will last until the the recovering node, already in a normal function, processes all its buffered messages and the incoming ones until it reaches the state of the rest of the replicas.

This way, the interval between *s* and *as* includes the start of the MADIS replica and its node recovery process (regarding global issues independent of the repositories and protocols used), as this is not in the scope of this paper, no further analysis is necessary. The interval between *as* and *ar* is the time the group communication toolkit takes to deliver the *ask-for-help* message. The interval *rs*-*rr* is the time between the end of the submission and the end of the preprocessing of the recovery message; this time is, in all cases, negligible as the implementation used overlaps the creation, sending, receiving and processing of the message (the recovery message is transmitted via a stream). The two interesting intervals are, therefore, *ar-rs* and *rr-ra*. The first one is the time needed to create and send the compacted message in the recoverer node and it can be seen in the graph that during this time, the recoverer node decreases its performance drastically, but only momentarily, due to the effort required to compact the data. Note that such compacting period lasts approximately two seconds in this example, but the regular pace of other nodes is retaken by the recoverer in less than one additional second. The second important interval (*rr-ra*) is the time needed to apply locally all the updates contained in the compacted message. This is the greatest interval, as expected, as it implies interaction with the underlying database. As showed, the compacting technique introduces an extra step (only a light overhead compared with the total processing time) but achieves notable reductions in the recovery time.

The enlarged graphs corresponding to the other two cases show similar results to the above exposed.

To sum up, the optimization presented in this work has been able to reduce the recovery time to a 71.4% of the original recovery time in a lightly loaded environment. Moreover, with 1000 missed transactions, the basic recovery technique fails to complete the recovery, since the recovering replica is not able to process the queue of received writesets on time, and its receiving queue continuously grows, whilst the optimized version does not get overloaded with ten times such load.

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 [14], 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 [3, 4] 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 [8], 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 [19] replication variant in order to support the snapshot [2] isolation level.

Only in [3] and [11] there are some performance analyses of database recovery protocols. But, as already said, [3] is penalized by its non-standard features (non-standard API and non-standard isolation level), whilst the replication protocol assumed in [11] was hybrid (could be configured either as eager or lazy, but always with a lazy core) and this introduced a high abortion rate that was partially compensated with an outdateness estimation function. In all cases, the advantages of both approaches –and both were developed by our research group– have been improved by the solution presented now (shortest recovery time, and lowest abortion rate).

There have been many other works devoted to database replica recovery [1, 4, 10, 13, 14] but, up to our knowledge, none of them has presented a performance study of their proposed solutions.

7 Conclusions

We have presented a first basic recovery approach for certification-based recovery protocols, analyzing its recovery time when the system load varies. Up to our knowledge this is the first performance study for such kind of recovery techniques in the field of database replication. 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 update compacting has also been presented. The performance study shows that the overall recovery time can be reduced up to a 71.4% of the recovery time of the basic approach, in the less favorable configuration for the optimized technique.

References

[1] J. E. Armendáriz-Iñigo. *Design and Implementation of Database Replication Protocols in the MADIS Architecture*. PhD thesis, Universidad Pública de Navarra, Pamplona (Spain), Feb. 2006.

[2] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, pages 1–10. ACM Press, 1995.

[3] F. Castro, J. Esparza, M. I. Ruiz, L. Irún, H. Decker, and F. D. Muñoz. CLOB: Communication support for efficient replicated database recovery. In *PDP*, pages 314–321, Lugano, Switzerland, Feb. 2005. IEEE-CS Press.

[4] F. Castro, L. Irún, F. García, and F. D. Muñoz. FOBr: A version-based recovery protocol for replicated databases. In *PDP*, pages 306–313, Lugano, Switzerland, Feb. 2005. IEEE-CS Press.

[5] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):427–469, Dec. 2001.

[6] F. Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2):56–78, 1991.

[7] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database replication providing generalized snapshot isolation. In *24th IEEE Symposium on Reliable Distributed Systems*, pages 73–84, Orlando, FL, USA, Oct. 2005.

[8] J. Esparza, A. Calero, J. Bataller, F. Muñoz, H. Decker, and J. Bernabéu. COPLA: A middleware for distributed databases. In *3rd Asian Workshop on Programming Languages and Systems (APLAS '02)*, pages 102–113, 2002.

[9] 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.

[10] J. Holliday. Replicated database recovery using multicast communication. In *NCA*. IEEE-CS Press, 2001.

[11] 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*, Las Navas del Marqués, Ávila (Spain), 2004.

[12] L. Irún, H. Decker, R. de Juan, F. Castro, J. E. Armendáriz, and F. D. Muñoz. MADIS: a slim middleware for database replication. In *11th Intnl. Euro-Par Conf.*, pages 349–359, Monte de Caparica (Lisbon), Portugal, Sept. 2005.

[13] R. Jiménez, M. Patiño, and G. Alonso. An algorithm for non-intrusive, parallel recovery of replicated data and its correctness. In *SRDS*, pages 150–159. IEEE-CS Press, 2002.

[14] B. Kemme, A. Bartoli, and O. Babaoglu. Online reconfiguration in replicated databases based on group communication. In *IEEE Int. Conf. on Dependable Systems and Networks*, pages 117–130, Göteborg, Sweden, July 2001.

[15] F. D. Muñoz-Escoí, J. Pla-Civera, M. I. Ruiz-Fuertes, L. Irún-Briz, H. Decker, J. E. Armendáriz-Iñigo, and J. R. González de Mendívil. Managing transaction conflicts in middleware-based database replication architectures. In *SRDS*, pages 401–410. IEEE-CS Press, Oct. 2006.

[16] PostgreSQL. Web site. Accessible in URL: http://www.postgresql.org, 2007.

[17] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. Strong replication in the GlobData middleware. In *Workshop on Dependable Middleware-Based Systems (in DSN 2002)*, pages G96–G104, Washington D.C., USA, 2002.

[18] Spread. The Spread communication toolkit. Accessible in URL: http://www.spread.org, 2007.

[19] M. Wiesmann and A. Schiper. Comparison of database replication techniques based on total order broadcast. In *IEEE Trans. Knowl. Data Eng. 17(4)*, pages 551–566, 2005.

[20] M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso. Database replication techniques: A three parameter classification. In *SRDS*, pages 206–217, 2000.