A Database Replication Protocol Where Multicast Writesets Are Always Committed

J.R. Juárez-Rodríguez, J.E. Armendáriz-Iñigo,
J.R. González de Mendívil
Universidad Pública de Navarra
31006 Pamplona, Spain
{jr.juarez, enrique.armendariz, mendivil}@unavarra.es

F.D. Muñoz-Escoí
Instituto Tecnológico de Informática
46022 Valencia, Spain
fmunyoz@iti.upv.es

Abstract

Database replication protocols based on a certification approach are usually the best ones for achieving good performance. The weak voting approach achieves a slightly longer transaction completion time, but with a lower abortion rate. So, both techniques can be considered as the best ones for replication when performance is a must, and both of them take advantage of the properties provided by atomic broadcast. We propose a new database replication strategy that shares many characteristics with such previous strategies. It is also based on totally ordering the application of writesets, using only an unordered reliable broadcast, instead of an atomic broadcast. Additionally, the writesets of transactions that are aborted in the final validation phase are not broadcast in our strategy. Thus, this new approach reduces the communication traffic and also achieves a good transaction response time (even shorter than those previous strategies in some system configurations).

1. Introduction

Database replication based on group communication systems has been proposed as an efficient and resilient solution for data replication. Protocols based on group communication typically rely on a broadcast primitive called *atomic* [10] or *total order* [6] broadcast. This primitive ensures that messages are delivered reliably and in the same order on all replicas. This approach ensures consistency and increases availability by relying on the communication properties assured by the total order broadcast primitive.

This primitive simplifies greatly the development of replication protocols: prevents the usage of an atomic commitment protocol [4], avoid the occurrence of distributed deadlock cycles and offer a constant interaction for committing a transaction. A comparison of database replication techniques based on total order broadcast is introduced in [19]. From those presented there, the two foremost techniques, in performance terms, are: certification-based [16] and weak-voting [13] protocols.

Certification-based protocols keep at each replica an ordered log of already committed transactions. When a transaction requests its commit the writeset[1] is multicast in a message, using the total-order service. Messages are treated by the replication protocol in the same order in which they are delivered at the replicas. Each writeset is certified against the information contained in the log, according to some predefined rules that depend on the required isolation level [8], in order to abort or commit the delivered transaction. In the former case, the writeset is discarded (except for its the delegate replica, where the transaction gets aborted) and, in the latter case, the writeset will be applied and committed at the remote replicas while it will be straightly committed at its delegate replica.

The weak-voting protocols also send the writeset using the total-order multicast upon the commit request of a transaction. When a writeset is delivered to a replica, it is *atomically* applied so it may cause the abortion of other existing local transactions. If an aborted transaction had already sent its writeset the protocol would multicast an abort message (using a weaker multicast delivery service [6]) to notify that the transaction must be aborted at all the replicas. When the writeset is delivered at its delegate replica and the transaction remains active (i.e. no other previous delivered writeset has rolled it back), the transaction will be committed and an additional message will be multicast to commit the transaction at the rest of the replicas [13].

From the previous descriptions, it should be clear that certification-based protocols need just one total-order message round per transaction whereas weak-voting ones need an additional round. Thus, the first ones present a bet-

[1] Depending on the transaction isolation level, it may be needed the readset, i.e. the set of objects read by the transaction

0-7695-3102-4/08 $25.00 © 2008 IEEE
DOI 10.1109/ARES.2008.62

ter behavior in terms of performance but higher abortion rates [19], since transactions may stay too long in the log (raising conflicts with new transactions being certificated) until removed from it after having committed in all replicas. Recently, Database Management Systems (DBMS) providing Snapshot Isolation (SI) [3] have become widely used, since this isolation level allows that read-only transactions are never blocked, using multi-version concurrency control. In this way, several certification-based protocols have been proposed to achieve this isolation level in a replicated setting [8, 14, 20, 15], whilst quite a few weak-voting ones [11, 12].

As it is well-known, database replication ensures higher availability and performance of accessed data. Hence, it is important to study other alternatives to the already presented replication protocols; unfortunately, it is quite difficult to find an alternative to those based on the total-order multicast. From our point of view, total-order based replication protocols offer two key properties: they reliably send the writeset to all replicas; and they provide the same scheduling of transactions and hence all replicas reach the same decision for each transaction submitted to the replicated system.

If we go further in the same-scheduling property, we may derive a replication protocol where a certification and a weak-voting protocol converge: establishing a deterministic and "*a priori*" known scheduling policy for all transactions in the system. The ideal case for a certification-based protocol is when all delivered writesets coming from a replica are known in advance to be successfully certified and hence the use of the log with previous certified transactions does not make any sense. Besides, notice that this is also the ideal case for a weak-voting protocol. In this case, there would be no need to multicast the additional message of the outcome of the transaction, since all delivered writesets would have been successfully "certified" and therefore they could be committed straight away. Nevertheless, this ideal replication protocol must ensure that all transactions which need to be aborted due to conflicts with remote ones be local, i.e. executed at their respective delegate replicas. However, it is not intuitive to set up in advance an appropriate scheduling of transactions so that it does not penalize certain transactions and at the same time maintains the property of non-aborting writesets that have been multicast; this is closely related to establish load balancing techniques [1].

In this paper, we propose a new approach through the description of a deterministic protocol. This protocol follows at each replica the most straightforward scheduling policy: at a given slot, only those writesets coming from a given replica are allowed to commit; other conflicting local transactions should be aborted to permit those writesets to commit. Actually, this is a round-robin policy based on replica identifiers which is unique and known by all the nodes of the system (since all replicas may know the identifiers of the other ones). In this deterministic protocol, a transaction is firstly executed at its delegate replica and once it requests for its commit, its updates are stored in a data structure and it will be committed when the turn of its delegate replica arrives (at its corresponding slot). Then, the replica will multicast all writesets from transactions that requested their commit since the last slot and they will be sequentially applied at the rest of replicas (after all writesets from the previous slot have been applied). Hence, it is easy to show that all local conflicting transactions are aborted and only those that survived will be multicast in their appropriate slot.

This generates a unique scheduling configuration of all replicas, in which all writesets are applied in the same order at all replicas. If we assume that the underlying DBMS at each replica provides SI the deterministic protocol will provide Generalized SI [8] (GSI). The atomicity and the same order of applying transactions in the system have been proved in [9] to be sufficient conditions for providing GSI. We provide some discussion about this fact in this paper.

This new approach provides several advantages over the previously presented techniques. Compared to the certification-based protocols, our approach does not use a certification log with the writesets of committed transactions and therefore there is no need of using a garbage collector to avoid its boundless growing. Compared to the weak voting protocols, it avoids the second message round to confirm the outcome of the transaction, since all multicast writesets are going always to commit. For the same reason, this approach reduces the network traffic and also the resource consumption of the replicas. A replica will never multicast writeset that finally aborts, avoiding its unnecessary delivery through the network and processing at the remote replicas.

We have simulated a scenario to compare this approach with a typical distributed certification protocol and we have verify that the abortion rate is reduced in many cases while maintaining very similar response times. Finally, we provide some outlines about how to deal with fault-tolerance issues, such as the failure of a replica and its subsequent re-join to the system.

The rest of the paper is organized as follows. The system model is presented in Section 2. The deterministic replication protocol is introduced in Section 3. A discussion of its correctness is given in Section 4. Section 5 shows its performance comparison against a certification-based replication protocol. Fault-tolerance issues are covered in Section 6. Finally, conclusions end the paper.

2. System Model

For our protocol proposal, we take advantage of the capabilities provided by a middleware architecture called

MADIS [15]. Users and applications submit transactions to the system. The middleware forwards them to the respectively nearest (local) replica for their execution, i.e. its delegate replica; the way it is chosen is totally transparent to the behavior of the protocol.

We assume a partially synchronous distributed system where message propagation time is unknown but eventually bounded. The system is composed by N replicas ($R_0, \ldots, R_{N-1}$) and each one of them holds a complete copy of a given database, i.e. full replication. An instance of the deterministic protocol is running in each replica and runs on top of a DBMS that provides SI.

A replica interacts with other replicas thanks to a Group Communication System [6] (GCS) that provides a reliable multicast communication service without any other ordering assumption rather than the reliable delivery of messages despite failures. Besides, the GCS also provides a membership service, which monitors the set of participating replicas and provides them with consistent notifications in case of failures, either real or suspected.

3. Deterministic Protocol

This Section explains the operation of the Determ-Rep protocol executed by the middleware at a replica R_k (Figure 1), considering a fault-free environment. Details about the failure and rejoin of a replica will be depicted in Section 6.

All operations of a transaction T are submitted to the middleware of its delegate replica (explicit abort operations from clients are ignored for simplicity). At each replica, the middleware keeps an array (towork) that determines the same scheduling of transactions in the system. Here, for the sake of understanding, it is assumed a round robin scheduling based on replica identifiers. In other words, towork is in charge of deciding which replica is allowed to send a message, or which writesets have to be applied and in which order. Each element of the array represents the actions that have to be performed when their turn arrives and each one is processed cyclically according to the work_turn.

The middleware forwards all the operations but the commit operation to the local database replica (step I of Figure 1). Each replica maintains a list (tocommit_wslist) which stores local transactions (T.replica $= R_k$) that have requested their commit. Thus, when a transaction requests its commit, the writeset is retrieved from the local database replica (T.WS). If it is empty the transaction will be committed straight away, otherwise the transaction (together with its writeset) will be stored in tocommit_wslist.

In order to commit transactions that have requested it, the replica has to multicast the stored writesets in a **tocommit** message and then wait for the reception of this message to finally commit the transactions (this is just for fault-tolerance issues explained later in Section 6). Since our protocol follows a round robin scheduling, the replica hast to wait for its turn (work_turn $= R_k$ in step III), so as to multicast all the writesets contained in tocommit_wslist using a simple reliable service. When the turn of a replica arrives and there are no transactions stored in tocommit_wslist, the replica will simply advance the turn to the next replica, sending a **next** message to all the replicas.

Upon delivery of any of these messages (**next** and **tocommit**) at each replica, they are stored in their corresponding positions in the towork array, according to the site R_k which the message came from (step II). It is important to note that, although these messages were sent since it was its turn at its replica, all replicas run at different speed and there can be replicas still handling previous positions of their own towork. Messages from different sites may be delivered disordered (as we do not use total order), but this is not a problem since they are processed one after another as its turn arrives. Disordered messages are stored in their corresponding positions in the array and their processing will wait for the deliver and processing of the previous ones. This ensures that all the replicas process messages in the same order and as a result all transactions are committed in the same order in all of them.

Thus, the towork array is processed in a cyclical way. At each turn, the protocol checks the corresponding position of the array (towork[work_turn]). If a **next** message is stored, the protocol will simply remove it from the array and change the turn to the following position (step IV) so as to allow the next position to be processed. If it is a **tocommit** message, we can distinguish between two cases (step V). If the sender of the message is the replica itself, transactions grouped in its writeset are local (already exist in the local DBMS) and therefore the transactions will be straightforwardly committed. In the other case, a remote transaction has to be used to apply and commit the transaction.

In this case, special attention must be paid to local existing transactions since they may conflict with the remote writeset application, avoiding it to progress. To partially avoid this we stop the execution of write operations in the system (see step I.2.a in Figure 1) when a remote writeset is applied at a replica, i.e. turning the ws_run variable to true. However, this is not enough to ensure the writeset application in the replica; the writeset can be involved in a conflict with local transactions that already updated some data items that intersect with the writeset.

This is ensured by a block detection mechanism, presented in [15], which aborts all local conflicting transactions (VI) allowing the writeset application to be successfully applied. Besides, this mechanism prevents local transactions that have requested their commit (T.precommit $=$ true) from being aborted by other local conflicting transactions, ensuring their completion. Note also that the writeset application

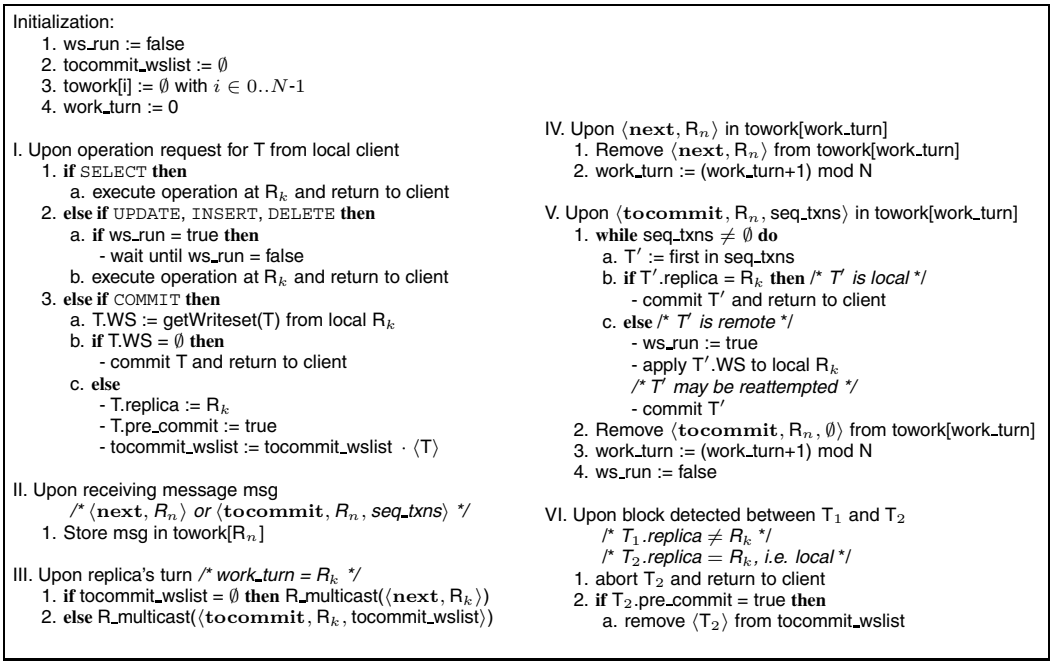


Figure 1. Determ-Rep **algorithm at replica** R_k

may be involved in a deadlock situation that may result in its abortion and hence it must be re-attempted until its successful completion.

3.1. Protocol Optimizations

Several optimizations can be considered for this protocol. If we take a look on how writesets are applied at remote replicas, there can be several alternatives. In Figure 1, we have chosen to submit the writesets one by one into the local database. However, they can be grouped in a single remote transaction, reducing the system overhead and therefore increasing the system performance.

Another important enhancement permits reducing the time a replica must spend to multicast a message when its turn arrives. Upon receiving a **tocommit** message, it is possible to know in advance which local committing transaction (stored in tocommit_wslist) are going to abort and which not. Therefore, if it is the turn of the replica, it may gather the transactions that are not going to abort and send the corresponding message without waiting for the writeset application in the local database replica. This optimization has been considered when it comes to performing our tests.

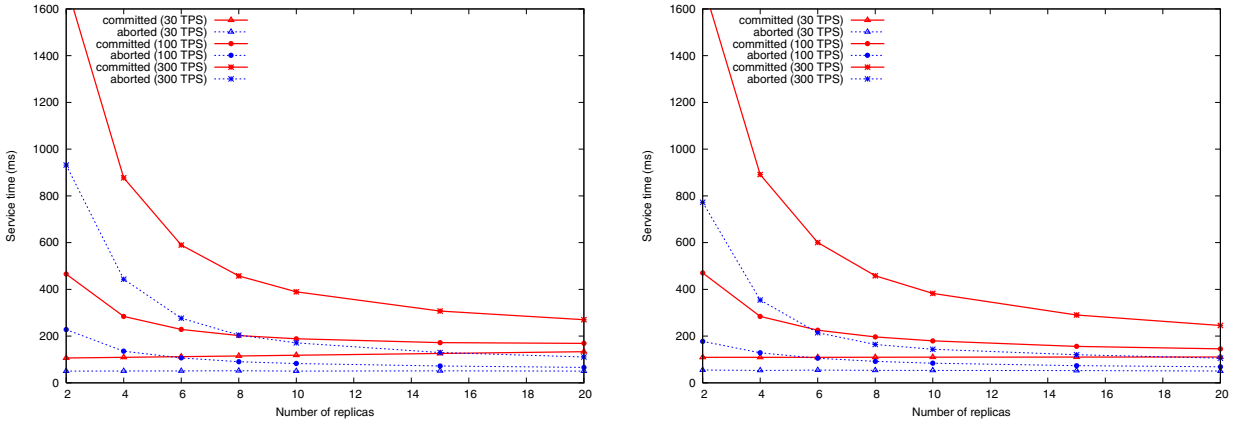
Finally, we can also consider a scheme of transactions based on conflict classes [5, 18] in order to increase the system performance. A conflict class represents a partition of the data and each application may partition the data depending on its requirements (e.g. there could be a class per table). If these partitions are well chosen, increasing the protocol concurrency may be possible and hence the system performance. When it comes to work with conflict classes, it is possible to avoid blocking all transaction operations in the local database in order to apply a remote writeset. In this case, existing transactions belonging to the same conflict class of the writeset are aborted in order to guarantee its successful application. However, it is only necessary to block operations from transactions of the same conflict class, while allowing the others to progress normally.

4. Correctness Discussion

In this Section we outline the discussion about the correctness of our replication protocol. In the following, we assume that we are under a failure-free environment. First of all, we have to show that every writeset submitted to a database will be eventually committed.

Theorem 1. *Given a transaction $T_i \in T$, whose delegate replica is R_k with $k \in N$, then its associated multicast writeset (T_i.WS) will be eventually committed.*

Proof. We have to distinguish whether it is executed at R_k or at a remote one R_j with $j \neq k$. In the first case, it will be committed as soon as the reliable message is delivered (see Figure 1, step V) since it already acquired all items it has to update. While in R_j, its associated $tocommit$ message has to be delivered and scheduled (i.e. the turn in R_j reaches the position of the message in towork, which corresponds



(a) Deterministic. (b) Certification-based.

Figure 2. Transaction completion time with 0% RO.

with its delegated replica). At that moment, it is submitted to the database (at the same time none local transactions are allowed to write anymore nor any other remote writeset is scheduled) and thanks to the block detection mechanism between transactions all possible local conflicting transactions will be eventually rolled back and, since no new write operations are allowed but the ones issued by T_i, the T_i.WS will be applied and committed. □

In the following we proof the atomicity of transactions; informally, if a transaction is committed at a given replica, it will be eventually committed at all replicas.

Theorem 2 (Atomicity of transactions)**.** *If a* $\langle \mathbf{tocommit}, R_n, \mathit{seq\_txns} \rangle$ *is processed and committed at a replica* R_k *with* $k \in N$*, then it will be eventually processed and committed at all replicas.*

Proof. This proof can be split into several parts. Let us denote as $\mathsf{R}_{k'}$ with $k' \in N \wedge k \neq k'$ the replica to analyze.
Reception of $\langle \mathbf{tocommit}, R_n, \mathit{seq\_txns} \rangle$ *at* $R_{k'}$. The message will be received since it has been received by R_k due to the fact that it was multicast by its associated delegate replica (in this case R_n) using the reliable channels between replicas.
The $\langle \mathbf{tocommit}, R_n, \mathit{seq\_txns} \rangle$ *message reaches its turn in* towork *at* $R_{k'}$ (i.e. work_turn $= \mathsf{R}_n$). First of all, it is worth noting that replica $\mathsf{R}_{k'}$ may run slower (if it is faster, it will be the other way round, exchanging replica identifiers) and its respective work_turn may be different and hence R_k may have already processed some items. Let us denote as n the position of the message $\langle \mathbf{tocommit}, \mathsf{R}_n, \mathsf{seq\_txns} \rangle$ in towork at $\mathsf{R}_{k'}$. Thus, we must ensure that the distance between n and work_turn is decreased and hence the message will be processed (i.e. writesets contained in seq_txns will be applied and committed). If we consider that the current position (work_turn) of towork is a **next** message, it will be removed from towork and the turn will advance to the next position; hence, the distance shortened. Otherwise, it is a **tocommit** message and its associated writesets will be eventually committed, by Theorem 1, and they will be removed from towork and the turn will advance to the next position; hence, the distance again decreased. Therefore, the turn will eventually reach the position of the message in the towork array.
The $\langle \mathbf{tocommit}, R_n, \mathit{seq\_txns} \rangle$ *is processed at* $R_{k'}$. This is easily shown by Theorem 1. Once the turn reaches the position of the message, the writesets will be successfully applied in the database. □

The atomicity of transaction does not ensure that all transactions are committed in the same order. If we ensure that all transactions are committed in the same order at all replicas then it will be satisfied a sufficient condition for generating GSI histories [9].

Theorem 3 (Same commit order at all replicas)**.** *All terminated transactions should follow the same commit order in all replicas.*

Proof. Due to Theorems 1 and 2 we know that a transaction committed at a replica will be committed at all replicas. It is easy to show that they will be applied in the same order thanks to the way towork is built. When different **next** or **tocommit** messages are delivered they are inserted at their appropriate towork slots. Writesets are extracted and applied in the cyclical order they are located in towork and they are committed in the same order they are applied. Hence, it is ensured that all replicas commit the same set of transactions in the same order. □

5. Experimental Results

We have compared our deterministic protocol with a certification-based protocol [15]. The simulation parameters have been summarized in Table 5. As this table shows, we have chosen a slow local network. This provides the worst results for our deterministic protocol since it depends a lot on the network delays. In practice, the combination of a reliable broadcast and a turn-based sending privilege can be considered as a way of implementing a total-order broadcast, as already discussed above.

Parameter	*Values*	*Parameter*	*Values*
Replicas	2 to 20	Min trans. length	100 ms
Global TPS load	30, 100, 300	WS application time	30 ms
Database size	10000 items	Connections	6/replica
Item size	200 bytes	Message delay	3 ms
Mean WS size	15 items	Trans. per test	40000
Mean RS size	15 items	Read-only trans	0%, 80%

Table 1. Simulation parameters.

The number of transactions used in each experiment has ensured that the standard deviation is below 3% of the plotted mean values in all figures being presented in this Section. Each transaction consists of an update and a reading sentence. For read-only transactions, the update has been replaced by another reading sentence. Each operation involves 15 items on average. The minimal transaction length is set to 100 ms, adding an inter-sentence delay that ensures that the transaction time is at least 100 ms. Note also that the global loads shown in Table 5 refer to the transaction arrival rate.

In our tests, we have used 6 connections per node, checking the protocols behavior in a LAN environment and using a worst case load consisting only in read-write transactions and another with 80% of read-only transactions (the common case in many real applications). Besides the transaction completion time, we also analyze their abortion rate in both cases.

Figures 2.a and 2.b show the results in the worst case, i.e. when no read-only transactions are included in the simulated load. Completion times for committed transactions are the same when the system consists of less than 10 replicas. Bigger systems generate slightly longer times in the deterministic approach. On the other hand, the certification strategy is able to abort transactions earlier, although the differences are only significant with the highest simulated load (300 TPS). Response times for other tested loads follow the same trend, i.e. there are no significant differences between both protocols.

The abortion rates for both kinds of loads are summarized in Figure 3. Comparing both protocols, with an update-only load (Figure 3.a and 3.b), the deterministic one provides the best results for light and medium loads (30, 100 TPS), always at least 20% better than the certification-based protocol and in some cases more than 50% better. However, in the heaviest load case (300 TPS) things are not so clear. The deterministic protocol has its maximum value when 6 replicas are used, whilst the certification-based protocol has its maximum with 10 replicas. Due to this, the deterministic protocol is better than the certification-based one when the system has more than 8 replicas.

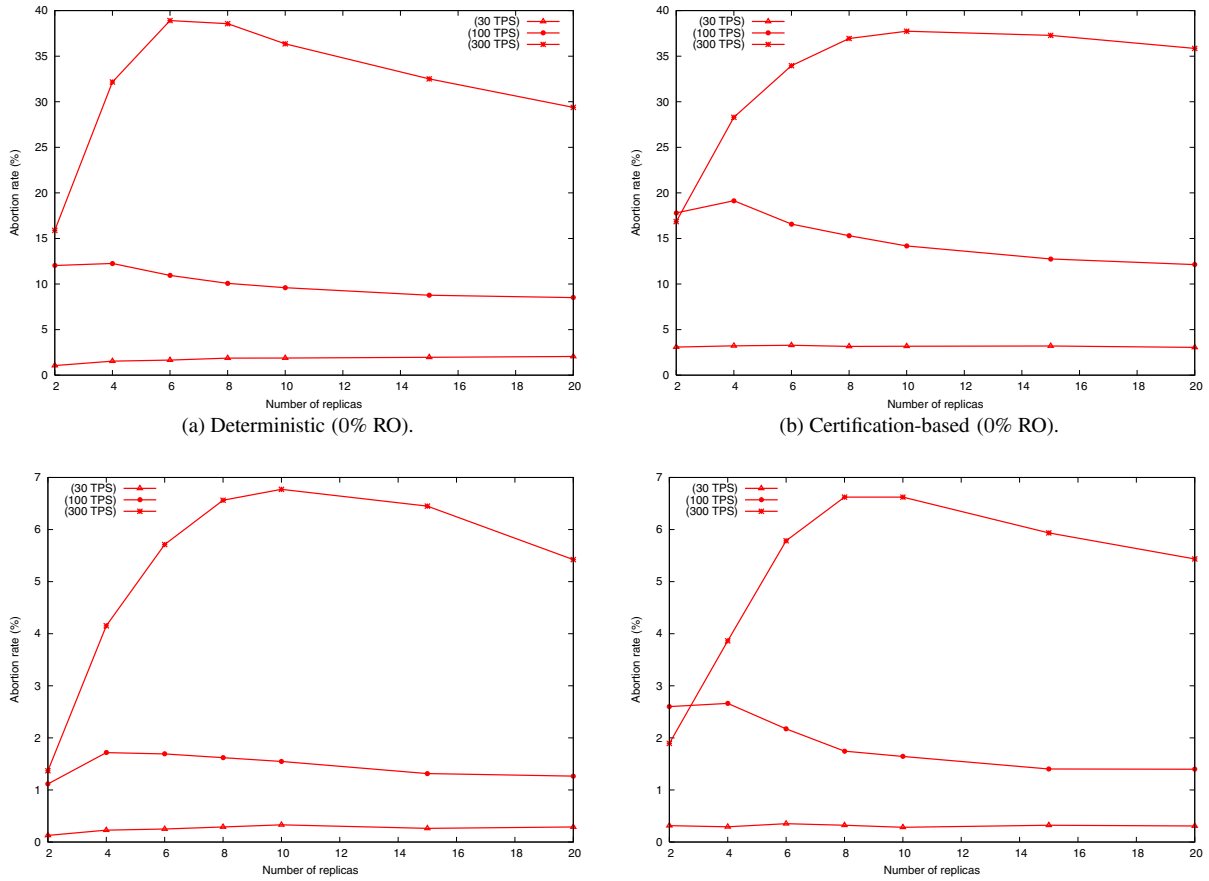
Finally, Figures 3.c and 3.d show the abortion rates when 80% of the transactions are read-only. In general, the results show similar trends to the previous case, i.e. with light and medium loads the deterministic protocol is much better than the certification-based one. With the heaviest load no clear winner can be identified, as it already occurred before, but now the curves follow different trends.

6. Fault Tolerance Issues

We have not covered until now any issue about the failure of a replica, what is something likely to happen in a replicated database system. It is interesting to give a dynamic nature of the composition of replicas in the system (a partially synchronous one). Thus, replicas may fail, re-join or new replicas may come to satisfy some performance needs. We suppose that the failure and recovery of a replica follows the crash-recovery with partial amnesia failure model [7]. Note that once a transaction has been committed, the underlying DBMS guarantees its persistence, but on-going ones are lost when a replica fails. This provides a partial amnesia effect.

These issues are handled by the GCS thanks to a membership service [6]. This service provides the notion of view [6], which is the set of current connected and active nodes. The view concept can be considered as a synchronization point for the replicated setting: each time a replica crashes or joins the system a view change event is fired, which provides a report of the number of connected members. This event is totally ordered for all replicas which install this new view and it also ensures that replicas contained in the former and in the new views deliver the same set of messages; hence the notion of view synchrony [6]. In replicated databases it is important to work under the primary component assumption [6], i.e. a replica may continue processing transactions provided that there are more than a half replicas connected; otherwise, it is usually forced to shutdown until it becomes part of the primary partition.

Related to this is the notion of uniform and same view delivery [6]: if a message is delivered by a replica (faulty or not), it will be eventually delivered to all replicas that install the next view in the former view. All these features let us know which writesets have been applied between failures or joins of nodes and, thus, define what to do in these cases. This will be outlined in the following, keeping in mind the protocol shown in Figure 1.



(a) Deterministic (0% RO).
(b) Certification-based (0% RO).
(c) Deterministic (80% RO).
(d) Certification-based (80% RO).

Figure 3. Abortion rates.

6.1. Replica Failure and Recovery Process

As said before, the failure of a replica R_j involves firing a view change event. Hence, all nodes will install the new view with the excluded replica. The most straightforward solution is that each alive replica R_k to silently discard the position of towork associated to R_j.

However, we should be more careful about missed writesets by the faulty replica R_j until the view change reports its failure. This is not a very difficult task thanks to the round robin nature of our protocol. Each replica has an auxiliary queue where delivered messages are stored. This queue will be pruned each time a new round is started, i.e. when its turn arrives. Hence, when a replica crashes it is only necessary to store the content of this queue. This information will be transferred when it rejoins the system again.

After a replica has crashed, it will eventually rejoin the system firing a view change event. This *recovering* replica has to apply the possible missed writesets on the view it crashed and the writesets while it was down. Thanks to the strong virtual synchrony, there is at least one replica that completely contains all the system state. Hence, there is a process to choose a *recoverer* replica among all alive nodes; this is an orthogonal process and we will not discuss it here, hence assume that there exists a *recoverer* replica.

Upon firing the view change event, like in the previous case, we need to rebuild the towork queue including the *recovering* replica position. The *recoverer* will wait for its turn to send the missed information to the *recovering* replica. Meanwhile, the *recovering* will send $next$ messages until it finishes applying the missed updates and keep on discarding messages coming from other available replicas. It is worth noting that the set of missed updates can be inferred quite easily, it is only needed to store the transaction identifier of the last committed transaction before the *recovering* replica crashed.

Thanks to some metadata tables present in some commercial DBMS, such as PostgreSQL, it is possible to infer the set of registers updated since that transaction and to transfer their current state. Concurrently to this, every alive

replica will store all writesets delivered that will be compacted [17] in an additional queue called pending_WS. Once the recovering is done applying missed updates, it will send a *pending* message. The delivery of this message to the next replica in towork will send the compacted writesets stored in pending_WS to the *recovering* and, thus, finish the recovery.

As it may be seen, we have followed a two phase recovery process very similar to the one described in [2]: the first phase consists in transferring the missed updates while the replica was crashed; and, the second one transfers the missed updates of the current view while the recovery process took place. This last phase serves while establishing a synchronization point with the rest of replicas to consider the *recovering* replica as alive.

7. Conclusions

Our deterministic database replication protocol proposal is able to inherit the best characteristics of both certification-based and weak-voting approaches. Thus, like a weak-voting protocol, it is able to validate transactions without logging history of previously delivered writesets, and like a certification-based protocol, it is able to validate transactions using only a single round of messages per transaction. Moreover, such a single round can be shared by a group of transactions already served at the same delegate replica.

The correctness of this new strategy has been justified. Additionally, its performance has been analyzed through simulation, providing a transaction completion time quite similar to that of a certification-based approach (the best one according to previous analysis [19]) in some configurations, and with a lower abortion rate.

Finally, a recovery strategy for this new kind of replication protocols has also been discussed. It can be easily matched with the regular tasks of this replication proposal.

Acknowledgments

This work has been partially supported by the European Union (EU FEDER funds) and Spanish Ministry of Science and Technology (research project TIN2006-14738-C02).

References

[1] C. Amza, A. L. Cox, and W. Zwaenepoel. A comparative evaluation of transparent scaling techniques for dynamic content servers. In *ICDE*, pages 230–241. IEEE-CS, 2005.

[2] J. E. Armendáriz-Íñigo, F. D. Muñoz-Escoí, J. R. Juárez-Rodríguez, J. R. González de Mendívil, and B. Kemme. A recovery protocol for middleware replicated databases providing GSI. In *ARES*, pages 85–92. IEEE-CS, 2007.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] P. A. Bernstein, D. W. Shipman, and J. B. R. Jr. Concurrency control in a system for distributed databases (sdd-1). *ACM Trans. Database Syst.*, 5(1):18–51, 1980.

[6] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

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

[8] S. Elnikety, F. Pedone, and W. Zwaenopoel. Database replication using generalized snapshot isolation. In *SRDS*. IEEE-CS, 2005.

[9] J. R. González de Mendívil, J. E. Armendáriz, J. R. Garitagoitia, L. Irún, F. D. Muñoz, and J. R. Juárez. Non-blocking ROWA protocols implement GSI using SI replicas. Technical Report ITI-ITE-07/10, Instituto Tecnológico de Informática, 2007.

[10] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Dep. of Computer Science, Cornell University, Ithaca, New York (USA), 1994.

[11] J. R. Juárez, J. E. Armendáriz, J. R. G. de Mendívil, F. D. Muñoz, and J. R. Garitagoitia. A weak voting database replication protocol providing different isolation levels. In *NOTERE'07*, 2007.

[12] J. R. Juárez, J. R. González de Mendívil, J. R. Garitagoitia, J. E. Armendáriz, and F. D. Muñoz. A middleware database replication protocol providing different isolation levels. In *EuroMicro-PDP. Work in Progress Session*, 2007.

[13] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Trans. Database Syst.*, 25(3):333–379, 2000.

[14] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD*. ACM, 2005.

[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 Computer Society, 2006.

[16] F. Pedone. *The database state machine and group communication issues (Thèse N. 2090)*. PhD thesis, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 1999.

[17] J. Pla-Civera, M. I. Ruiz-Fuertes, L. H. García-Muñoz, and F. D. Muñoz-Escoí. Optimizing certification-based database recovery. In *6th ISPDC*, pages 211–218, Hagenberg, Austria, 2007. IEEE-CS.

[18] D. Skeen and D. D. Wright. Increasing availability in partitioned database systems. In *PODS '84: Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 290–299, New York, NY, USA, 1984. ACM Press.

[19] M. Wiesmann and A. Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE TKDE*, 17(4):551–566, 2005.

[20] S. Wu and B. Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *ICDE*, pages 422–433. IEEE-CS, 2005.