A Deterministic Database Replication Protocol Where Multicast Writesets Never Got Aborted

J.R. Juárez, J.E. Armendáriz, F.D. Muñoz, J.R. González de Mendívil, J.R. Garitagoitia

Universidad Pública de Navarra, 31006 Pamplona, Spain,
Instituto Tecnológico de Informática, 46022 Valencia, Spain

{jr.juarez,enrique.armendariz,mendivil,joserra}@unavarra.es, fmunyoz@iti.upv.es

Technical Report TR-ITI-ITE-07/15

A Deterministic Database Replication Protocol Where Multicast Writesets Never Got Aborted

J.R. Juárez, J.E. Armendáriz, F.D. Muñoz, J.R. González de Mendívil, J.R. Garitagoitia

Universidad Pública de Navarra, 31006 Pamplona, Spain,
Instituto Tecnológico de Informática, 46022 Valencia, Spain

Technical Report TR-ITI-ITE-07/15

e-mail: {jr.juarez,enrique.armendariz,mendivil,joserra}@unavarra.es,
fmunyoz@iti.upv.es

June 29, 2007

Abstract

Database replication protocols based on a certification approach are usually the best ones for achieving good performance when an eager update everywhere technique is being considered. 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 eager replication when performance is a must, and both of them need 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 need not be broadcast in our strategy. Thus, this new approach always reduces the communication traffic and also achieves a good transaction response time (even shorter than those previous strategies in some system configurations).

1 Introduction

There have been presented several approaches for the full replication of data in distributed databases [18]. One of the most outstanding database replication techniques [9] is the eager update everywhere (all updates are propagated to all replicas before the commit of a transaction) [9, 17] with constant interaction (a fixed number of messages is exchanged per transaction) [17]. Under this approach, the execution flow of a transaction can be split into two different main phases: the first one, all operations are entirely executed at the delegate replica of the transaction; and followed by the second phase, started when the transaction requests its commit, all updates are collected and grouped (denoted as writeset) at the delegate replica and sent to all replicas. The commitment or abortion of a transaction is decided in this last phase maybe with an additional message round with the outcome of the transaction and, hence, ensuring the constant interaction feature.

The practical implementation of this technique is by way of the total-order multicast delivery service provided by group communication systems [5] that lead to a family of database replication protocols presented in [18, 12, 13]. Among all of them, the two foremost, in performance terms, are [18]: certification-based; and, weak-voting protocols. Both will be outlined in the following.

Certification-based protocols store 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 they were delivered at all available replicas. Each writeset is certified, according to some predefined rule that depends on the isolation level [7],

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

against the information contained in the log in order to abort or commit the delivered transaction. In the former case, the writeset is discarded (except at its the delegate replica where the transaction gets aborted) and, in the latter case, it will be applied and ensured its commitment at the remote replicas (while straightly committed at its delegate replica). On the other hand, the weak-voting protocols upon the commit request of a transaction send the writeset using the total-order multicast too. When a writeset is delivered to a replica, it is *atomically* applied so it may cause the abortion of other transactions. If the aborted transaction has already sent its writeset the protocol will abort the transaction and multicast an abort message (using a weaker multicast delivery service [5]) to the rest of replicas saying the transaction must be aborted. 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 is committed and an additional message is multicast saying that the transaction must be committed [12]. From the previous description, it should be clear that certification-based protocols just need one total-order message round per transaction whereas weak-voting ones need an additional round. Thus, the first one presents a better behavior in terms of performance but with higher abortion rates [18].

Recently, due to the use of Database Management Systems (DBMS) providing Snapshot Isolation (SI) [3] –since it allows that read-only transactions never block, using multi-version concurrency control–, we have found several certification-based protocols issued to achieve this isolation level in a replicated setting [7, 14, 19, 15] while quite a few weak-voting ones [11, 10]. If we consider the previous certification-based protocols [7, 14, 19, 15] one of their key differences is the way in which the application and commitment of an already certified writeset is handled by the given replication protocol. It is important to note that a certified writeset must be committed at all replicas just to ensure the atomicity of a transaction. Hence, some protocols will make use of re-attempt mechanisms [14] whilst others will take advantage of the block detection mechanism already provided by almost every DBMS [15].

As it is well-known, database replication ensures higher availability and performance of accessed data. Hence, it will be important to study other interesting alternatives to the already presented replication protocols; unfortunately, it is very difficult to find an alternative to those based on the total-order multicast. This 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. Moreover, from our point of view total-order based replication protocols offer two key properties: a) they reliably send the writeset to all replicas; and, b) they provide the same scheduling of transactions and, thus, all replicas reach the same decision for each transaction in the replicated setting.

If we go further in the same-scheduling property we may derive a replication protocol where a certification-based and a weak-voting protocol converge: establishing a deterministic and "*a priori*" known scheduling policy of all transactions in the system. The ideal case for a certification-based protocol will consist in that all delivered writesets coming from a certain replica are known in advance to be successfully certified and, hence, the use of the log of previously certified transactions does not make any sense. Moreover, notice that this is also the ideal case of a weak-voting replication protocol where there is no need to multicast the additional message of the outcome of the transaction (a commit message in this case) since all delivered writesets must be committed. Nevertheless, this ideal replication protocol must ensure that all transactions that need to be aborted due to conflicts with remote ones are local, i.e. being executed at their respective delegate replicas.

On the other hand, it is not intuitive to set up in advance an appropriate scheduling of transactions so that the given pattern does not penalize certain transactions while maintaining 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 an eager update-everywhere replication protocol [9] that follows 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. Actually, this is a round-robin policy based on replica identifiers that is unique and known by all replicas. In this deterministic protocol, a transaction is firstly executed at its delegate replica, once it requests for its commit, its updates are stored in a data structure that will be committed when it reaches the turn of the delegate replica. The writesets will be multicast at the time of the slot and 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 where all writesets are applied in the same order. If we assume that

the underlying DBMS at each replica provides SI the deterministic protocol will provide Generalized SI [7] (GSI). The atomicity and the same order of applying transactions in the system have been shown in [8] as sufficient conditions for providing GSI. Hence, we provide some discussion about this fact in this paper. We have simulated a scenario where we compare this approach with a typical distributed certification protocol and verify that the abortion rate is reduced while maintaining the response time, specially in LAN configurations. Finally, we provide some outlines about how to overcome 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 assumed for presenting the deterministic protocol 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

We assume a partially synchronous distributed system where message propagation time is unknown but bounded. Such 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. So, full replication is being assumed. 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 [5] (GCS) that provides a reliable multicast communication service without any other ordering assumption rather than the reliable delivery of messages. Clients access to the system by way of its delegate replica; the way its delegate replica is chosen is totally transparent to the behavior of the protocol. Details about the failure and re-join of a replica will be depicted in Section 6.

Initialization:
1. ws_run := **false**
2. lastcommitted_tid := 0
3. WS_list := $\emptyset$
4. to_work := $\{\langle \mathbf{order}, (n \mod N), n, \emptyset \rangle, n \in \mathbb{N}\}$

I. Upon operation request for T_i from local client
1. **if** UPDATE, INSERT, DELETE **then**
   a. **if** ws_run = **true then**
      - wait until ws_run = **false**
   b. execute operation at R_k and return to client
2. **else if** SELECT **then**
   a. execute operation at R_k and return to client
3. **else** /* COMMIT */
   a. T_i.WS := get_writeset(T_{ik}) from local R_k
   b. **if** T_i.WS = $\emptyset$, **then** commit and return
   c. T_i.pre_commit := **true**
   d. WS_list := WS_list $\cdot \langle T_i \rangle$

II. First $\langle \mathbf{order}, k, n, \emptyset \rangle$ in to_work
1. **if** WS_list = $\emptyset$ **then** R_multicast($\langle \mathbf{next}, k, n, \emptyset \rangle$)
2. **else** R_multicast($\langle \mathbf{to\_commit}, k, n,$ WS_list$\rangle$)

III. Upon receiving m */* first */*
/* m is either $\langle \mathbf{next}, id, n, \emptyset \rangle$ */
/* or $\langle \mathbf{to\_commit}, id, n,$ seq_txns$\rangle$ */
1. Substitute $\langle \mathbf{order}, id, n, \emptyset \rangle$ by m

IV. First $\langle \mathbf{next}, id, n, \emptyset \rangle$ in to_work
1. Remove $\langle \mathbf{next}, id, n, \emptyset \rangle$ from to_work

V. First $\langle \mathbf{to\_commit}, id, n,$ seq_txns$\rangle$ in to_work
1. **while** seq_txns $\neq \emptyset$ **do**
   a. T' := first in seq_txns
   b. **if** T'.replica = k **then**
      - commit T'
      - remove $\langle T' \rangle$ from seq_txns
   c. **else** */* T' is remote */*
      - ws_run := **true**
      - apply T'.WS to local R_k
      /* T' may be reattempted */
      - commit T'
      - remove $\langle T' \rangle$ from seq_txns
2. Remove $\langle \mathbf{to\_commit}, id, n, \emptyset \rangle$ from to_work
3. ws_run := **false**

VI. Upon block between T_1 and T_2
/ T_1.replica $\neq k$ */*
/ T_2.replica $= k$, i.e., local */*
1. abort T_2
2. **if** T_2.pre_commit = **true then**
   a. remove $\langle T' \rangle$ from WS_list

Figure 1: Determ-Rep algorithm at replica R_k

3 Deterministic Protocol

This Section is devoted to explain the Determ-Rep protocol executed by the middleware at a replica R_k (shown in Figure 1). All operations of a transaction T_i are submitted to the middleware (abort operations are ignored for simplicity) corresponding to its delegate replica. The middleware maintains a list (to_work)

that determines the same scheduling of transactions in the system, i.e. each replica stores the same copy, though not shared, of to_work. Here, for the sake of understanding, it is assumed a round robin scheduling based on replica identifiers in Figure 1. In other words, to_work is in charge of deciding which replica can send a message or which writeset has to be applied respectively. Initially, it is filled with infinite tuples (just for algorithm presentation purposes) of the form $\langle order, n \mod N, n, \emptyset \rangle$ with $n \in \mathbb{N}$, sorted by n.

All operations are forwarded to the local database replica but the commit operation (step I of Figure 1). Besides, the replica maintains a list (WS_list) storing local transactions (T_i.replica $= R_k$) that have requested their commit. Hence, when a transaction requests its commit the writeset is retrieved from the local database replica, if it is an empty one the transaction will be committed, otherwise the transaction (along its writeset) will be stored in WS_list.

Concurrent to this, the replica must wait for its turn, i.e. the $\langle \mathbf{order}, k, n, \emptyset \rangle$, in its own to_work (II). If there are no transactions stored in WS_list it will make advance the turn to the next middleware replica, via sending the $\langle \mathbf{next}, k, n, \emptyset \rangle$ to all replicas. Otherwise, it will multicast (using the reliable service) all the writesets contained in WS_list and its content is emptied in a $\langle \mathbf{to\_commit}, k, n, \text{WS\_list} \rangle$.

Upon delivery of these **next** and **to_commit** messages they are substituted in their proper positions (which is reflected by n) of the to_work list of the delivered replica (III). It is important to note that, although it was the first position of the to_work contained in the message sender replica, all replicas run at different speed and there could be replicas still handling previous positions of their own to_work. Therefore, in the case of reaching the first position a **next** message it will be deleted so the next position of to_work can be executed (IV).

If we take a look on how writesets are applied, there can be several alternatives, though it is an implementation detail, to apply one by one transaction or all of them grouped in a single remote transaction. In Figure 1 it has been followed the former (V). We will distinguish two cases of writeset execution: at its delegate replica or at a remote replica. In the first case the transactions will be directly committed. Whereas in the other case a remote transaction is used to apply and commit the transaction. As it can be inferred, if we are not very careful it will be the case that this transaction can never progress, it may conflict with local transactions and being involved in a local database deadlock. To partially avoid this we stop the execution of write operations in the system (see step I.1.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 prevent the writeset abortion in the replica; the writeset can be involved in a deadlock with local transactions that already wrote in some data item that intersects with the writeset and be aborted and, hence, it must be re-attempted until its successful completion. This last feature is ensured since a pretty similar block detection mechanism as already presented in [15] can be used to abort all possible conflicting local transactions (VI).

4 Correctness Discussion

In this Section we will outline the discussion about the correctness of our replication protocol. In the following, we will 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 to_commit message has to be delivered and scheduled (i.e. reaches the first position of to_work). At that moment, it is submitted to the database (at the same time none local transactions is 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 from 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* $\langle \mathbf{to\_commit}, id, n, \mathit{seq\_txns} \rangle$ *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 different parts. Let us denote as $\mathsf{R}_{k'}$ with $k' \in N \wedge j \neq k'$ the replica to be analyzed.
Reception of $\langle \mathbf{to\_commit}, id, n, \mathit{seq\_txns} \rangle$ *at* $\mathsf{R}_{k'}$. 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_{id}) using the reliable channels between replicas.
The $\langle \mathbf{to\_commit}, id, n, \mathit{seq\_txns} \rangle$ *message reaches the first position of* to_work *at* $\mathsf{R}_{k'}$. 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 to_work list may contain items already processed by R_k. In particular, it contains different $\mathbf{to\_commit}$, $\mathbf{next}$ and $\mathbf{order}$ tuples. Let us denote as n' the first position of to_work. Hence, we must ensure that the distance between n and n' is decreased and the message will be processed (i.e. writesets contained in seq_txns will be applied and committed). If we start considering all the $\mathbf{order}$ messages, by the process carried out at R_k, each respective position will be filled by the proper $\mathbf{to\_commit}$ or $\mathbf{next}$ since they were reliably multicast and they will be eventually delivered. However, this does not decrease the distance, it only ensures that there will not be any $\mathbf{order}$ message. If we consider that the current position n' of to_work is a $\mathbf{next}$ message, it will be removed from to_work and, hence, the distance shortened. Otherwise, it is a $\mathbf{to\_commit}$ message and its associated writesets will be eventually committed, by Theorem 1, and they will be removed from to_work and the distance decreased. Therefore, the message will eventually reach the first position of to_work.
The $\langle \mathbf{to\_commit}, id, n, \mathit{seq\_txns} \rangle$ *is processed at* $\mathsf{R}_{k'}$. This is easily shown by Theorem 1. Once it reaches the first position of to_work 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 [8].

Theorem 3 (Same commit ordering 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 to_work is built. It is initially built with the same $\langle \mathbf{order}, id, n, \mathsf{seq\_txns} \rangle$ tuples for all R_{id} with $id \in N$ and $n \in \mathbb{N}$. When different $\mathbf{next}$ or $\mathbf{to\_commit}$ messages are delivered they preserve their associated n value and they replace (exactly once per position) the $\mathbf{order}$ tuple at that position by the respective message. On the other hand, writesets are sequentially applied from to_work 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 simulated our deterministic protocol, comparing it with a general certification-based one. To this end, the simulation parameters being considered have been summarized in Table 1. As it can be seen in such table, we have chosen very slow networks for two different configurations: either a LAN or a WAN. 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 seen as a way of implementing a total-order broadcast, as already discussed above. But this leads to a poor-performance way of implementing such a total-order broadcast when the network is slow.

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. 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 the the global loads shown in Table 1 refer to the transaction arrival rate.

Parameter	*Values*	*Parameter*	*Values*
Number of replicas	2, 4, 6, 8, 10, 15, 20	Minimal transaction length	100 ms
Global load	30, 100, 300 TPS	Writeset application time	30 ms
Database size	10000 items	WAN message delay	140 ms
Item size	200 bytes	LAN message delay	3 ms
Mean writeset size	15 items	Transactions/experiment	40000
Mean readset size	15 items	Read-only transactions	0%, 80%

Table 1: Simulation parameters.

In order to model resource contention we have used two different configurations. In the first one, each machine has 6 open connections for accessing the underlying database. This means that the local DBMS is able to serve 6 concurrent transactions at a time, leading to a contention scenario if the global load exceeds 45N TPS, being N the number of replicas. In the second simulation deployment we have simulated 12 open connections per node. This implies that we need twice the load of the previous scenario to saturate the system.

5.1 Results with 6 Connections per Node

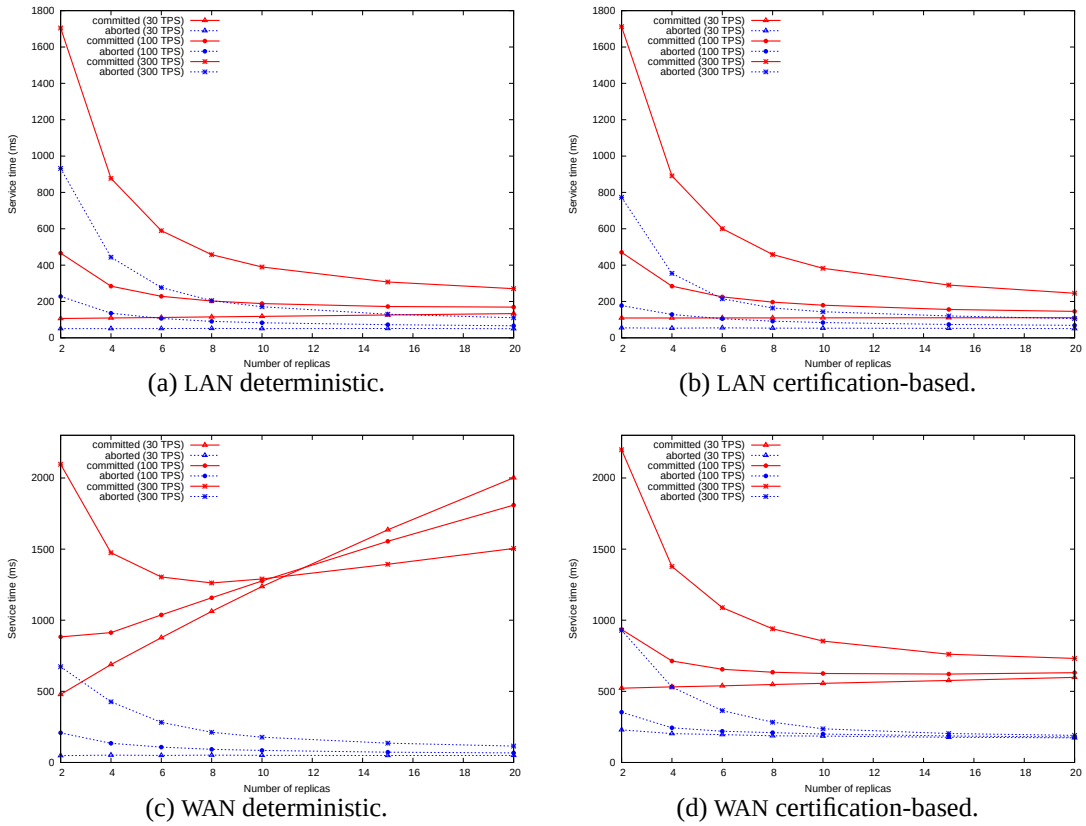
In this series of tests, we have used 6 connections per node, checking the protocols behavior both in LAN and WAN deployments and using a worst case load consisting only in read-write transactions and another with 80% of read-only transactions (the common case in many applications). Besides the transaction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

Thus, Figure 2 shows the results in the worst case being considered, i.e., when no read-only transactions are included in the simulated load. In the LAN case, 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).

Differences are much bigger in the WAN deployment (Figure 2.c and 2.d). Although the deterministic protocol is slightly better when only two replicas are considered, its transaction completion time has an increasing trend in all cases when more replicas are added to the system (except with 300 TPS, where adding more replicas has a favorable impact due to its contention reduction). Moreover, the lower is the load, the bigger is the increasing trend. It is worth noting that the deterministic protocol depends a lot on the size of the logical ring. If we assume a system with more than 10 nodes and a very light load, there are few transactions being multicast when the sending privilege arrives to each node, and this leads to a lengthy transaction service time. Indeed, for the lightest load being considered in our simulation (30 TPS), the completion time is almost 4 times bigger when 20 replicas are used. However, the transaction abortion time is shorter in the deterministic protocol than in the certification-based one, in all cases (both when the load and the number of nodes are varied). Note that in the deterministic protocol such transaction abortion times are almost equal in both kinds of network (the single difference is for a deployment with only two replicas with the heaviest load, where resource contention has caused an increase in the abortion time in the LAN network) due to its very light network traffic. However a certification-based protocol is quite penalized when the network is slow, recall that it requires an atomic broadcast for each completed transaction, and this equally affects the completion time for committed and for aborted transactions.

Figure 3 provides the same set of results for a more realistic load where 80% of transactions are read-only. Results for a LAN deployment (Figures 3.a and 3.b) follow the same trend shown above, i.e., there are no significant differences between both protocols.

In the WAN deployment (Figures 3.c and 3.d), the differences found when all transactions were read-only have been highly reduced in this case. Now, the deterministic protocol provides better or comparable results when less than 7 replicas are considered, whilst in the previous case this only happenned with 2 replicas. Moreover, when many replicas have been used, the differences are still big, but lower than in the previous case. Thus, for the worst case (20 replicas and 30 TPS) the mean transaction completion time (for committed transactions) is 413.11 ms in the deterministic protocol and 199.12 ms in the certification-based



(a) LAN deterministic. (b) LAN certification-based.

(c) WAN deterministic. (d) WAN certification-based.

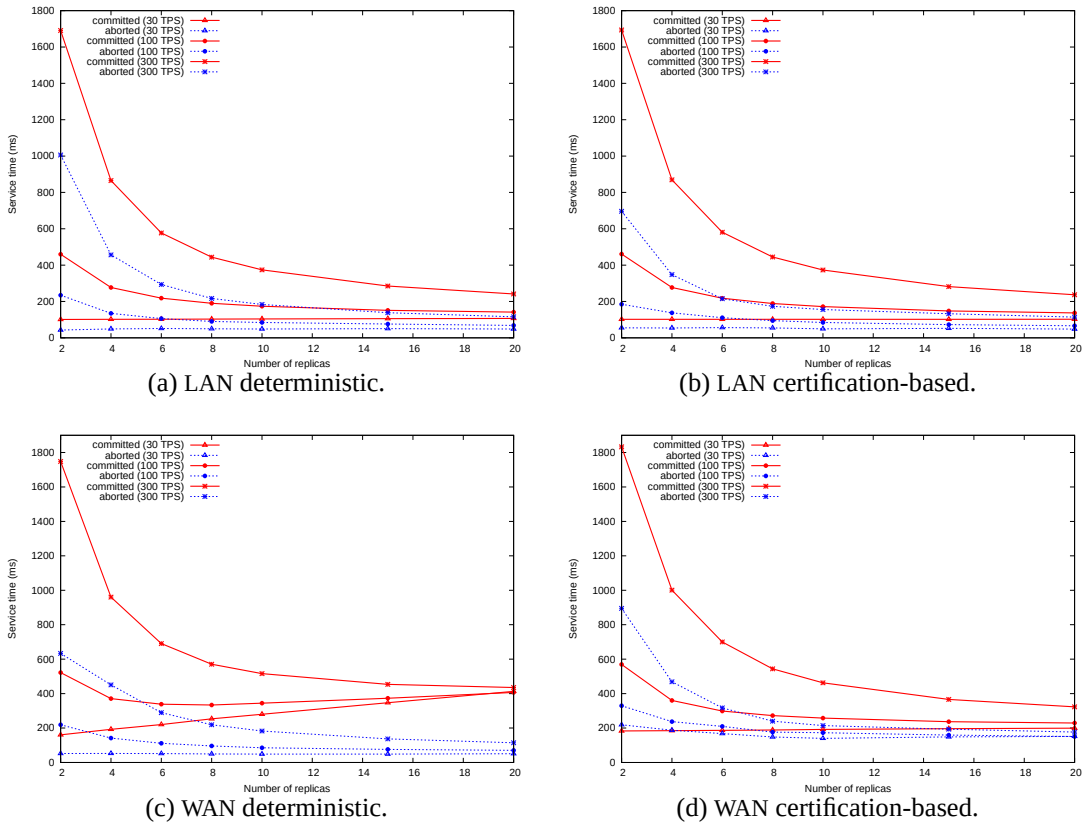
Figure 2: Transaction completion time with 0% RO-trans (6 conn/replica).

one; i.e., twice bigger whilst in the previous case it was four times bigger. As expected, the fast abortion time of the deterministic protocol is still maintained with this kind of load.

The abortion rates for both kinds of loads are summarized in Figure 4. With a load without read-only transactions, both protocols have a higher abortion rate when a slow network (WAN) is being used. In the deterministic protocol, such differences increase with the load. They are negligible with 30 TPS but exceed a relative 22% with 300 TPS (e.g., with 20 replicas, the respective abortion rates are 35.92% and 29.38%; and thus 35.92/29.38=1.222). On the other hand, in the certification-based protocol the differences between the WAN and LAN results are almost constant with light load (30 TPS), but with medium and heavy load practically disappear when few replicas are used and get progressively increasing when more replicas are considered.

Comparing both protocols, the deterministic one provides the best results for light and medium loads, with bigger differences in the WAN case, and always at least 20% better than the certification-based protocol. 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, both in the LAN and WAN case, whilst the certification-based protocol has its maximum with 20 replicas in the WAN case and with 10 replicas in the LAN one. Due to this, the deterministic protocol is better than the certification-based one when there are more than 10 replicas in the WAN case, and when there are more than 8 in the LAN one.

Finally, Figures 4.c and 4.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 load 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 without read-only transactions, but now the curves follow different trends.



(a) LAN deterministic. (b) LAN certification-based.

(c) WAN deterministic. (d) WAN certification-based.

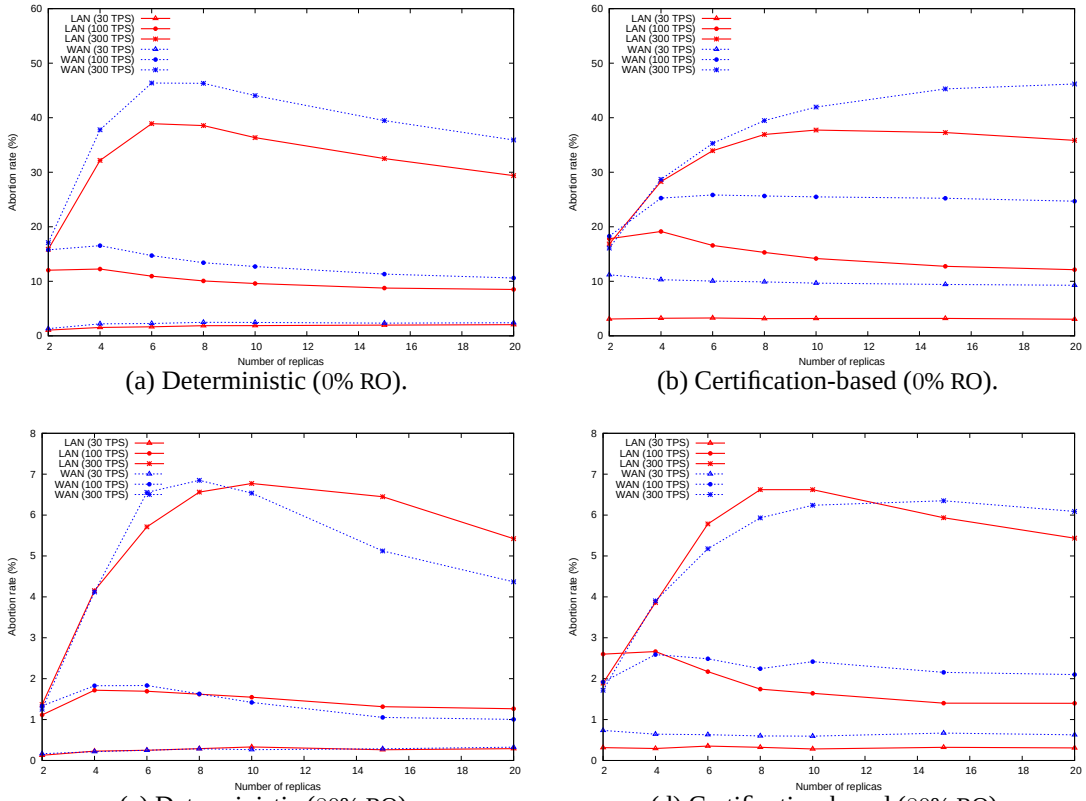
Figure 3: Transaction completion time with 80% RO-trans (6 conn/replica).

5.2 Results with 12 Connections per Node

For completeness purposes we also include in Figures 5, 6 and 7 the results with 0% and 80% RO transactions, and the abortion rates, respectively using 12 open connections per node. This simulates transaction servers with greater capacity, thus reducing resource contention. In all cases the curves follow the same trends than in the previous subsection, but with lower completion times and abortion rates, respectively.

6 Fault Tolerance Issues

Right now, we have not covered any issue about the failure of a replica which is something that is more likely to occur in a replicated database system. Moreover, it will be interesting to give a dynamic nature of the composition of replicas in the system (a partially synchronous one). Hence, replicas may fail, re-join or new replicas may come to satisfy some performance needs. The failure and recovery of a replica follows the crash-recovery with partial amnesia failure model [6]. Note that once a transaction has been committed, the underlying DBMS guarantees its persistence, but on-going transactions are lost when a replica fails. This provides a partial amnesia effect. Management of these issues are handled by the GCS thanks to the Membership Service [5]. This service provides the notion of view, the set of current connected and active nodes. In replicated databases it is important to work under the primary component assumption [5], i.e. a replica is allowed to continue processing transactions provided that there are more than a half replicas connected; otherwise, in general, it is forced to shutdown until it becomes part of the primary partition. The view concept may 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 [5], that provides as a report the number of already connected members. Moreover, this event is totally ordered for all replicas that install this new



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

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

Figure 4: Abortion rates (6 conn/replica).

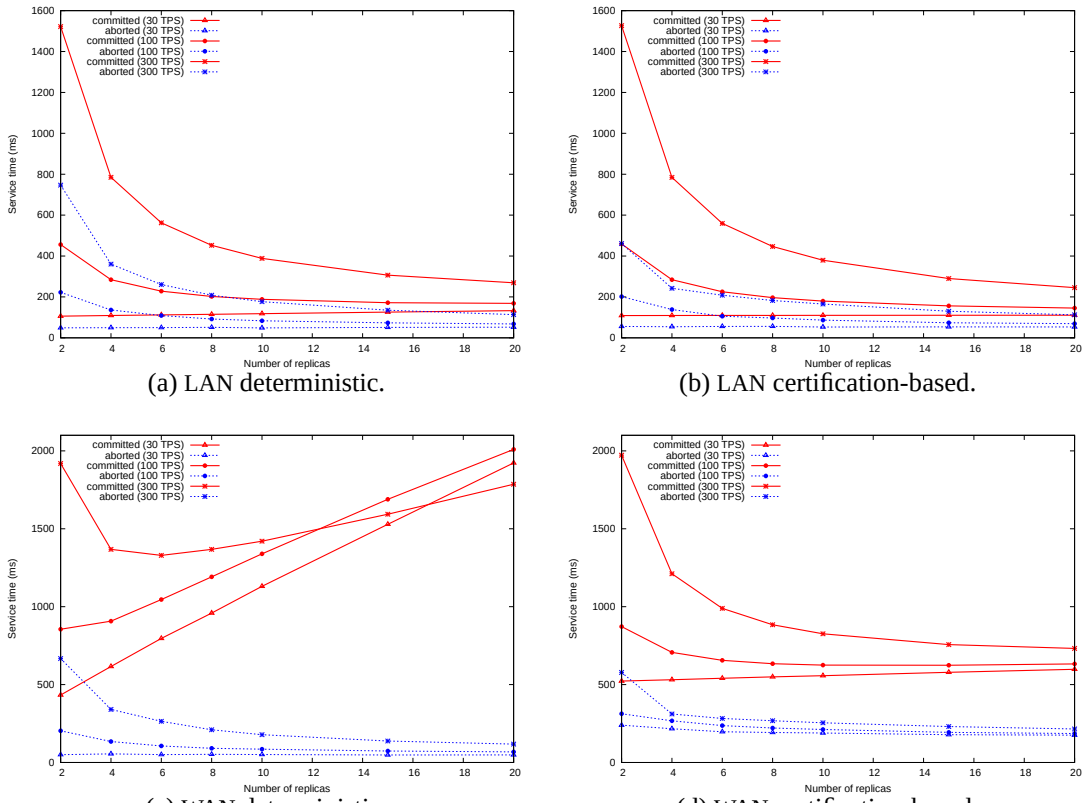
view and it also ensures that replicas contained in the former and the new views deliver the same set of messages; hence, giving the notion of view synchrony. Related to this is the notion of uniform and same view delivery [5] consisting in that if a message is delivered by a replica (faulty or not), 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 and joins of nodes and, thus, define what to do in case of a failure or join of a new replica that will be outlined in the following just keeping in mind the protocol shown in Figure 1.

6.1 Replica Failure Process

As it has been 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 positions of to_work associated to R_j. However, one should be more careful about missed writesets by the faulty replica R_j until the view change reporting its failure. However, this is not a very difficult task thanks to the round nature of our protocol. A replica has an auxiliary queue where delivered messages are stored. This queue is pruned each time a new round is started, i.e. it receives a new message that belongs to itself. Hence, when a node crashed it is only needed to store the content of this queue. This information will be transferred when it will re-join the system back again. Besides, for the normal scheduling of transactions in the system it is needed to rebuild the to_work queue.

6.2 The Process of Recovering a Replica

After a replica has crashed, it will eventually re-join the system firing a view change event about this fact. This *recovering* replica has to apply the (possibly) missed updates on the view it crashed and the updates



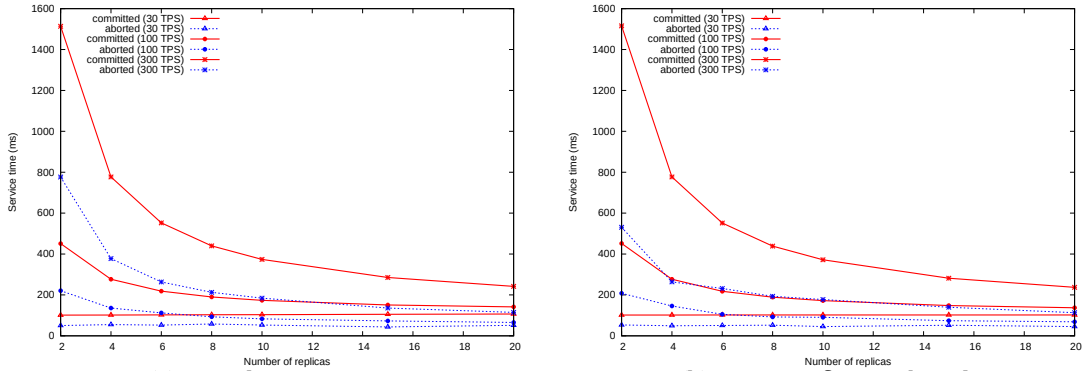
(a) LAN deterministic. (b) LAN certification-based.

(c) WAN deterministic. (d) WAN certification-based.

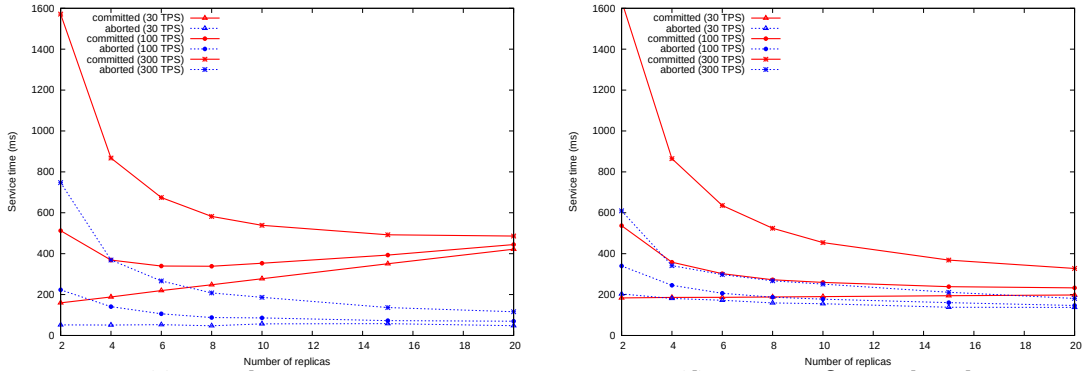
Figure 5: Transaction completion time with 0% RO-trans (12 conn/replica).

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 amon all alive nodes; this is an orthogonal process and we will not discuss here it any further, we will assume that there exists a *recoverer* replica.

Upon firing the view change event, like in the previous case, we need to rebuild the to_work queue including the *recovering* replica. The *recoverer* will wait for its turn to send the missed information to the *recovering* replica. Meanwhile, the *recovering* will send $next$ messages until its 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, infer the set of registers updated since that transaction and transfer their current state. Concurrently to this, every alive replica will store all writesets delivered that will be compacted [16] 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 to_work will send the compacted writesets stored in pending_WS to the *recovering* and, thus, finish the recovery process. 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.



(a) LAN deterministic. (b) LAN certification-based.



(c) WAN deterministic. (d) WAN certification-based.

Figure 6: Transaction completion time with 80% RO-trans (12 conn/replica).

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 needing a logged 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.

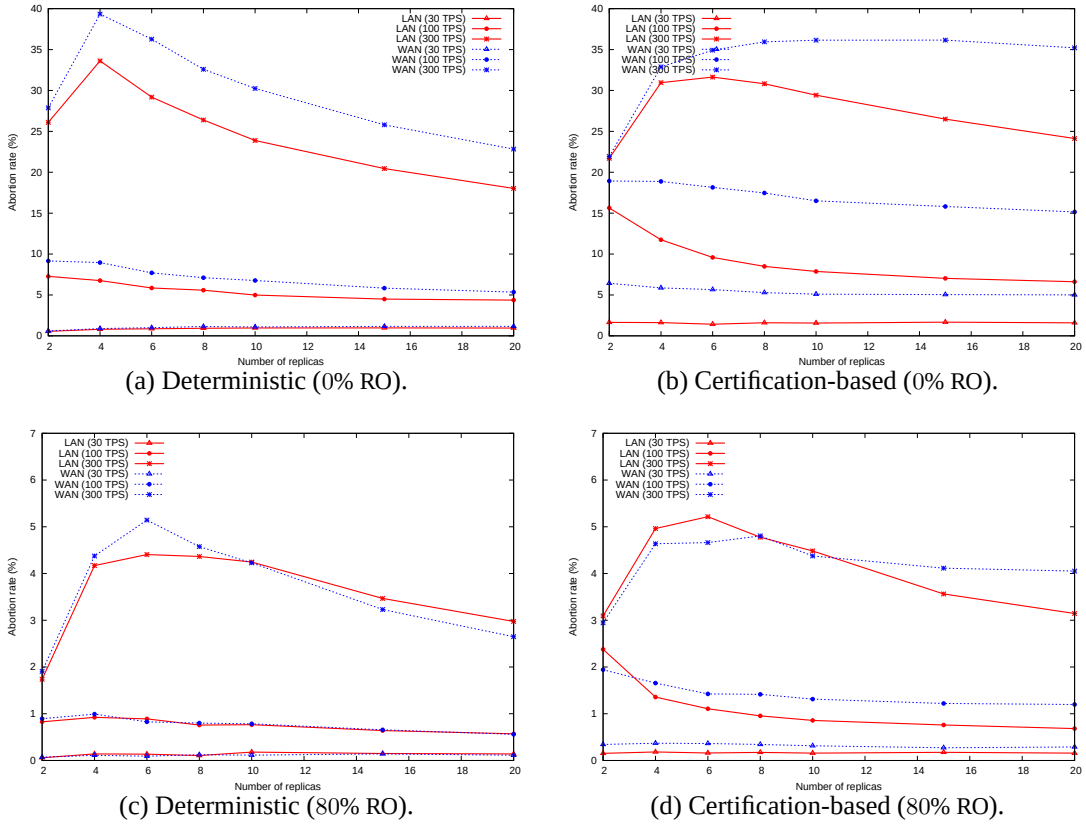
The correctness of this new strategy has been justified. Additionally, its performance has been analyzed through simulation, providing a transaction completion time quite similar to that of a certification-based approach (the best one according to previous analysis [18]) in some configurations, and with 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.

References

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

[2] José Enrique Armendáriz-Íñigo, Francesc D. Muñoz-Escoí, José Ramón Juárez-Rodríguez, José Ramón González de Mendívil, and Bettina Kemme. A recovery protocol for middleware replicated databases providing GSI. In *ARES*, pages 85–92. IEEE-CS, 2007.



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

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

Figure 7: Abortion rates (12 conn/replica).

[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] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

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

[7] Sameh Elnikety, Fernando Pedone, and Willy Zwaenopoel. Database replication using generalized snapshot isolation. In *SRDS*. IEEE-CS, 2005.

[8] José Ramón González de Mendívil, José Enrique Armendáriz-Íñigo, José Ramón Garitagoitia, Luis Irún-Briz, Francesc Daniel Muñoz-Escoí, and José Ramón Juárez-Rodríguez. Non-blocking ROWA protocols implement GSI using SI replicas. Technical Report ITI-ITE-07/10, Instituto Tecnológico de Informática, 2007.

[9] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *SIGMOD*. ACM, 1996.

[10] José Ramón Juárez, José Ramón González de Mendívil, José Ramón Garitagoitia, José Enrique Armendáriz, and Francesc Daniel Muñoz. A middleware database replication protocol providing different isolation levels. In *EuroMicro-PDP. Work in Progress Session*, 2007.

[11] José Ramón Juárez-Rodríguez, José Enrique Armendáriz-Íñigo, José Ramón González de Mendívil, Francesc Daniel Muñoz-Escoí, and José Ramón Garitagoitia. A weak voting database replication protocol providing different isolation levels. In *NOTERE'07: 7me NOuvelles TEchnologies de la REpartition*, 2007.

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

[13] Bettina Kemme, Fernando Pedone, Gustavo Alonso, André Schiper, and Matthias Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE Trans. Knowl. Data Eng.*, 15(4):1018–1032, 2003.

[14] Yi Lin, Bettina Kemme, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD*. ACM, 2005.

[15] Francesc D. Muñoz, J. Pla, María Idoia Ruiz, Luis Irún, Hendrik Decker, José Enrique Armendáriz, and J. R. Gonzalez de Mendivil. Managing transaction conflicts in middleware-based database replication architectures. In *SRDS*. IEEE-CS, 2006.

[16] J. Pla-Civera, M. I. Ruiz-Fuertes, L. H. García-Muñoz, and F. D. Muñoz-Escoí. Optimizing certification-based database recovery. In *6th ISPDC*, Hagenberg, Austria, 2007. IEEE-CS. Acc. for publication.

[17] M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso. Database replication techniques: A three parameter classification. In *SRDS*. IEEE-CS, 2000.

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

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