k-Bound GSI: A Flexible Database Replication Protocol

J.E. Armendáriz-Iñigo
Instituto Tec. de Informática
Campus de Vera s/n
46022 Valencia, Spain
armendariz@iti.upv.es

J.R. Juárez, J.R. G. de Mendívil
Universidad Pública de Navarra
Campus de Arrosadía s/n
31006 Pamplona, Spain
{jr.juarez, mendivil}@unavarra.es

H. Decker, F.D. Muñoz-Escoí
Instituto Tecnológico de Informática
Campus de Vera s/n
46022 Valencia, Spain
{hendrik, fmunyoz}@iti.upv.es

ABSTRACT

Several previous works have proven that there is no way of guaranteeing a snapshot isolation level in symmetrical replicated database systems without blocking transactions when they are started. As a result of this, the generalized snapshot isolation (GSI) level was defined, relaxing a bit the freshness of the snapshot being taken when a transaction is initiated in its local replica. This enhances performance, since transactions do not need to get blocked, but in some cases will increase the abortion rate. This paper proposes a flexible protocol that is able to bound the degree of snapshot outdateness from a relaxed GSI to the strict one-copy equivalent SI. Additionally, it proposes an optimistic solution where transactions do not block, and only need to be re-initiated when their optimistic start fails. Such re-initialization is made very soon and only rolls back the first transaction accesses, without waiting for the transaction completion. Finally, if 1CSI is not enough, this protocol is also able to manage transactions with *serializable* isolation, if such a level is requested.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed databases; H.2.4 [**Systems**]: Distributed databases; H.2.4 [**Systems**]: Transaction processing; H.3.5 [**Online Information Services**]: Data sharing.

General Terms

Database Replication.

Keywords

Replicated databases, isolation levels, eager database replication, scalability, performance, transaction scheduling.

1. INTRODUCTION

The *snapshot* isolation level (or SI, for short) was defined in [1] in order to identify a practical isolation level that has been supported for years in many DBMSs based on multi-version concurrency control (MVCC). However, those DBMSs provided it as an implementation of the *serializable* isolation level, and [1] and other papers proved that there were some isolation anomalies that could not be prevented using such level. In spite of this, the SI level has several advantages when it is compared to a strict *serializable* one thanks to its MVCC origin: read accesses never block, being able to complete read-only transactions very fast; and, in a replicated setting no read-set management is needed for any transaction. These features are very attractive for achieving good performance in a replicated database system. Additionally, the consistency anomalies that may arise using this level (when compared to a *serializable* one) are easily identifiable and avoidable.

Our aim in this paper is to design a flexible replication protocol able to manage both the *serializable* level and the SI level. However, some recent works have proven that a strict one-copy equivalent SI level [3] is quite hard to achieve in a replicated setting [4, 6], since this requires that all transactions are blocked on their start, waiting for the writeset arrival for all prior transactions. So, instead of such strict SI level we provide the mechanisms needed to select the "outdateness" of the snapshot being taken when a transaction is started. Thus, our work has been designed taking as its basis several recent database replication protocols [4, 11, 12] that provide the *generalized snapshot* isolation (or GSI, for short) level [4]. This generalized level relaxes the freshness of the snapshot gotten when a transaction starts, as described in [4].

Due to these limitations, the protocol to be presented here makes possible that the user selects which kind of *snapshot* isolation compliance is needed by each transaction, ranging from a default GSI to a strict one-copy SI. The novelty of our approach consists in being optimistic; i.e., even for the SI case, transactions do not block, and they only need to be re-initiated when conflicts that prevent such transaction to be continued arise. However, such conflicts are detected very soon and lead to a cancelation that only rolls back a few operations, re-initiating later the whole transaction. Moreover, such conflicts seldom arise.

Additionally, as proven by many current applications, in order to be *practical* and easily applicable to different contexts, a replication support must be able to manage multiple isolation levels, since different transactions in a single application may have different isolation requirements. See the TPC-C benchmark application as an example. Due to this, and as it has been already proposed in other papers [2], this

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'07 March 11-15, 2007, Seoul, Korea
Copyright 2007 ACM 1-59593-480-4 /07/0003 ...$5.00.

protocol is able to support the two most demanded isolation levels on top of MVCC DBMSs.

The rest of this paper is organized as follows. Section 2 describes the protocol supporting the GSI, SI, and *serializable* isolation levels. Later, Section 3 justifies how these levels are supported. Finally, Sections 4 and 5 outline some related works and conclude this paper.

2. A FLEXIBLE PROTOCOL

To implement our flexible Read One Write All Available protocol [7] we assume that all replicas have a DBMS that executes transactions providing SI and each one stores a copy of the database. Besides, we need to distinguish two logical timestamps associated to the start of a transaction T, as proposed in [4]: the *start* timestamp ($T.start$) corresponds to the "local" logical time of the first operation being executed by transaction T, and its *begin* timestamp ($T.begin$) corresponds to the "global" logical time that would be assigned in a hypothetical one-copy execution of such transaction. In order to translate $T.begin$ from such hypothetical time to a "concrete" one, T sends by atomic broadcast [8] a message that contains its identifier to all replicas. We refer to this message as T.ID. Such T.ID message will be delivered using the same destination group as those writeset messages multicast at the end of each transaction, using the constant interaction approach proposed in [14]. As a result, if some writesets are delivered between the sending of T.ID and its delivery, those writesets might invalidate the snapshot taken by transaction T at time $T.start$, at least if they have updated some of the objects already (or intended to be) read by T. Such kind of collisions is not a problem in GSI, but however violates a strict definition of a one-copy SI level. On the sequel, we refer to these writesets as *prior conflicting writesets* (or PC-WS, for short).

As it has been previously introduced, our replication protocol main goal is to avoid blocking the starting point of transactions with the SI level. The T.ID message approach introduces a limitation to our protocol: the dynamic collection of readsets during the execution of transactions does not prevent the "outdateness" of the snapshot taken at $T.start$. In other words, it may be possible for an active read-only transaction T_1 to read a value x contained in PC-WS, after its validation against PC-WS, as no restriction is imposed to read operations. Hence, we may circumvent this problem by the definition of a set of tables, items or restrictions which its outdateness a transaction wants to be aware of. This is not a weird assumption, in fact the TPC-W benchmark defines the tables to be read on each web interaction.

Our aim is to define a function that measures the distance between $T.start$ and $T.begin$ ($d(T.start, T.begin)$) being able to bound such distance for each transaction; i.e., each transaction T should be able to specify a value k such that $d(T.start, T.begin) < k$. Obviously, with an infinite value for k we achieve the most relaxed GSI whilst with a zero value we get SI. Note that a SI schedule is trivially a GSI schedule [4]. We also overload the value of this bound in order to select the isolation level. So, the meaning of its different values is:

$k > 0$: Bound values for the GSI level, according to the description given above.

$k = 0$: SI level.

$k = -1$: Serializable level.

$k = \perp$: GSI level.

Table 1: The list of attributes for a transaction T_i

aborted	It is a boolean initially set to false. It gets a true value when a remote writeset arrives and such writeset forces the rollback of the local transaction according to the distance and bounds set for it. However, the local transaction is not immediately reinitiated. It will be when its T.ID message is delivered.
conflicts	The number of conflicts found with incoming writesets, according to the getConflicts() function explained below.
end	A logical time that corresponds to the commit time for this transaction. It is needed for certification purposes.
decision	It holds the decision on the transaction termination (either *commit* or *abort*). This attribute can only be set in the replica that has started the transaction and is only needed when the k attribute has a -1 value (serializable level).
id	A global identifier for transaction T_i. It can be built using the identifier of the local node and the transaction identifier being used by the underlying DBMS. It is needed for building the T.ID message.
k	This attribute establishes the bound on the $d(T_i.s, T_i.b)$ as defined above.
tables_read	The set of tables or rows that T_i is interested in managing its outdateness freshness.
si	A boolean that is set to true when a transaction T_i receives its own T.ID message and all writesets previously received have been applied. Once this happens, T_i has obtained the proper snapshot version for reading.
start	This attribute gets its value from the local last-committed_tid, holding then the start timestamp for its associated transactions.
WS	The writeset of transaction T_i.
RS	The readset of transaction T_i (needed for local transactions with serializable isolation level at its delegate replica).

In order to define such distance we have multiple possibilities: time, overall PC-WS –even not colliding–, number of colliding items in each writeset, number of times an item has been modified and so on. However, we only consider this one in the paper:

Colliding PC-WS. The number of writesets received in the interval ($T.start, T.begin$) having a non-empty intersection with the *intended* readset of transaction T.

As it has been outlined before, this replication protocol is able to provide serializable with SI replicas. It is possible to obtain transaction executing with serializable isolation level provided that it satisfies the *dynamic serializability condition* (DSC) stated in [4], in which a transaction T_i sees its read-write conflicts from the beginning of the transaction ($T_i.begin$) till its commit time ($T_i.commit$). Hence, if another conflicting transaction commits during that period of time ($[T_i.begin, T_i.commit]$) then T_i will be rolled back. It is very easy to implement the DSC condition, it is only needed to parse again the "`SELECT`" statement to convert it into a "`SELECT FOR UPDATE`" one; hence, read and write conflicts are governed by the *first-committer-wins* rule [1]. This would simplify the development of a serializable behavior of the replication protocol, provided that the write set is properly collected, i.e. those objects that have been really updated. Hence, the readset propagation is avoided since the commitment of a transaction is left to the replica where the transaction was firstly executed; it sends a message with the outcome of the transaction (*commit* or *abort*) to all replicas.

Considering all these issues, we propose the replication

protocol shown in Fig. 1. The system consists of a fixed number of replicas $R = \{R_1, \ldots, R_N\}$. Each replica contains a full copy of the database. The DBMS provides SI isolation level for transactions executed at each replica. We also consider a set of clients $C = \{C_1, \ldots, C_M\}$. Clients are the source of transactions (see Table 1 for the associated fields needed by the replication protocol); they specify the *colliding PC-WS* ($T_i.k$) and another parameter ($T_i.tables\_read$) that establishes the set of tables, or objects they are interested in maintaining their outdateness for reading. This is done to avoid the dynamic collection of the readset. In order to process a transaction T_i, a client C_i connects to a server R_i and submits transaction T_i to R_i. We call the server R_i the *delegate* for transaction T_i. A copy of the replication protocol runs on all servers. We assume the existence of an atomic and a reliable broadcast facility [8] to deliver the writesets for certification to all replicas and the weak voting decision for serializable transactions [13] respectively. Each replica contains two queues: *ws_list*, it contains the writesets already validated; and *tocommit_queue* a queue for managing the flow execution of transactions. Its algorithm is divided into four main parts after a small initialization. Each part can be executed by a different thread:

(I) Operations on the Delegate Replica. When the delegate replica R_n receives a transaction T_i from a client C_i, it executes transaction T_i but the first operation serves to establish the outdateness of the snapshot gotten by T_i. R_n sends a T.ID message by atomic broadcast to all replicas. It is important to note that if one operation is a "`SELECT`" and the requested isolation level for its transaction is serializable, the middleware, or the DBMS-core modified solution, must be able to convert such "`SELECT`" into a "`SELECT FOR UPDATE`". In fact, the MADIS middleware architecture, presented in [9], parses the SQL sentences if it is configured to do so, but these issues have not been detailed in the algorithm. When commit time is reached, the writeset of T_i is broadcast to all servers using the atomic broadcast.

(II) T.ID Message Delivery. This thread is devoted to listening to T.ID at the delegate replica of the transaction, otherwise it is silently discarded. This message will be appended to *tocommit_queue*.

(III) Writeset Message Delivery. When a replica receives a writeset, it applies the certification test (III.1 and III.2) checks T_i.WS against the writesets of its overlapping transactions T_j (T_i.start $< T_i$.end $\wedge$ T_i.WS $\cap$ T_j.WS$\neq \emptyset$) contained in *ws_list*. If the certification is passed T_i will be appended to *tocommit_queue* and *ws_list*; finally, its T_i.end field updated.

Some transaction conflict checks are done in this part by means of the getConflicts() function; mainly those needed for bounding the "outdateness" specified. This function returns an integer: 0, empty intersection between the incoming writeset and the specified *tables_read* (second argument) of a local transaction; or, 1, non-empty intersection. It is important to note that this function can be modified according to the different distance definitions we have previously outlined. All local transactions whose T.ID has not been processed yet are checked for its outdateness.

(IV) Queue Processing. This last part governs the validation of the snapshot gotten by a local transaction and the commitment process of transactions. It is important to note that serializable transactions end differently, i.e. an additional message exchange is needed containing the decision (*abort* or *commit*).

```
Initialization:
  1. lastvalidated_tid := 0
  2. lastcommitted_tid := 0
  3. ws_list := ∅
  4. tocommit_queue := ∅
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 includes ⟨k, tables_read⟩ //
      - T_i.conflicts := 0
      - T_i.decision := commit
      - T_i.RS := ∅
      - T_i.aborted := FALSE
      - T_i.si := FALSE
      - T_i.start := lastcommitted_tid
      - multicast T_i.id in total order
    b. if T_i.aborted = FALSE
      - execute operation at R_n
    c. return to client
  2. else /* commit */
    a. if T_i.aborted = FALSE
      - T_i.WS := getwriteset(T_i) from local R_n
      - if T_i.WS = ∅, then commit and return
      - multicast T_i using total order
II. Upon receiving T_i.id
  1. if T_i is local in R_n
    a. append T_i.id to tocommit_queue
  2. else discard message
III. Upon receiving T_i
  1. if ∃ T_j ∈ ws_list : T_i.start < T_j.end ∧
     T_i.WS ∩ T_j.WS ≠ ∅
    a. if T_i is local then abort T_i at R_n else discard
  2. else
    a. T_i.end := ++lastvalidated_tid
    b. append T_i to ws_list and tocommit_queue
  3. ∀ T_j : T_j is local in R_n ∧ T_j.si = FALSE
     ∧ T_j.aborted = FALSE
    a. T_j.conflicts := T_j.conflicts + getConflicts(T_i.WS, T_j.tables_read)
    b. if T_j.k ≠ -1 then T_j.aborted := (T_j.conflicts > T_j.k)
IV. T_i := head(tocommit_queue)
  1. remove T_i from tocommit_queue
  2. if T_i is a T.ID message
    a. T_i.si := TRUE
    b. if T_i.aborted = TRUE
      - restart T_i /* All its operations must be restarted */
                    /* and this also includes step I.1.a.    */
    c. return
  3. if T_i is remote at R_n
    a. begin T_i at R_n
    b. apply T_i.WS to R_n
    c. ∀ T_j : T_j is local in R_n ∧ T_j.WS ∩ T_i.WS ≠ ∅
       ∧ T_j has not arrived to step III
      - abort T_j
    d. ∀ T_j : T_j is local in R_n ∧ T_j.k = -1 ∧ T_j.RS ∩ T_i.WS ≠ ∅
       ∧ T_j has not arrived to step IV
      - abort T_j
      - T_j.decision := abort /* The T_j.decision messages */
        /* are only sent if T_j arrives to step IV. */
  4. if T_i.k = -1 ∧ T_i is local in R_n then multicast T_i.decision // Reliable //
  5. if T_i.k = -1 then wait until T_i.decision delivered
    a. if T_i.decision = abort then
      - abort T_i
      - return
  6. commit T_i at R_n
  7. ++lastcommitted_tid
```

Figure 1: k-bound GSI algorithm at replica R_n

Let us start with the T.ID, this step serves to restart the transaction execution of an outdated transaction. We wait until the validated writesets are applied, hence it is not repeatedly rolled back. The second part deals with the writeset application at a remote replica. The conflict checks (according to the intended isolation level) are made in the substeps "c" and "d". These checks are implemented in an automatized way in MADIS, as described in [9]. Substep "c" aborts those local transactions that have not been already validated and conflict with the writeset to be applied.

Whilst substep "d" deals with local transactions executing in serializable isolation level that have not already multicast its decision message (on the contrary the writeset will get blocked).

The decision message is broadcast to all replicas (using the reliable service) by the delegate replica for serializable transactions, in a similar fashion to weak voting protocols [10, 13]. All replicas will be waiting for the delivery of this message. The rest of (GSI and one-copy SI) transactions are straightly committed.

3. ISOLATION LEVELS COMPLIANCE

In this Section an outline of the isolation level compliance will be given in a failure free environment. As it has been shown in [6], a replication protocol provides GSI if the following three conditions are met: all DBMS replicas provide SI; all transactions are atomically committed at all available nodes; and, all transactions are committed in the same order at all available nodes.

To prove that this implementation is deterministic and obeys GSI rules [4, 6], we need to show two properties. The first property is that at the certification of T_i, all replicas have the same *ws_list* and *last_validated_tid*. Hence, every replica reaches the same decision on the certification process of T_i Thread III in Fig. 1). Atomic broadcast is used to deliver the certification request, which contains T_i.start and T_i.WS, to all replicas. Atomic broadcast guarantees two properties [8]: agreement (if a replica delivers message m, then every replica delivers m) and order (no two replicas deliver any two messages in different orders). It is easy to prove, by mathematical induction on the length of *ws_list*, that: *ws_list* is the same at all replicas as well as the *last_validated_tid*, since it is incremented each time a transaction is added. Next, we have to show that transactions are committed in the same order at all replicas. Writesets are also appended to the *tocommit_queue* in the same order as they are certified. By the flow execution of Thread IV in Fig. 1, the writesets are committed in the order established by the *tocommit_queue* queue.

A transaction T_i executed in k-bound GSI (i.e. $T_i.\mathsf{k} \geq 0$) must check the outdateness of its selected items (T_i.tables_read) which is checked at its delegated replica each time a writeset is certified. This process will continue until its respective T.ID message is delivered (Thread II) and processed in the *tocommit_queue* on Thread IV at Step 2.a. This last step defines its assigned global snapshot and if its distance with overlapping transactions is less than T_i.k, it is allowed to proceed and we are in the same case as in GSI. Hence, as it was previously shown writesets are atomically applied and their commit ordering is totally ordered.

In case of a *serializable* transaction T_i. It is executed at the delegate replica satisfying the DSC condition stated in [4] in which a transaction T_i sees its read-write conflicts from the beginning of the transaction till its commit time. Hence, read and write conflicts are governed by the *first-committer-wins* rule [1] and the ordering between conflicting transactions is governed by the atomic broadcast facility.

4. RELATED WORK

In [4], the notion of GSI is introduced. They present two replication protocols based on certification that satisfies GSI (*prefix-consistent* SI). The first proposal consists of a centralized certifier (master database) and a number of database replicas. The centralized certifier contains the most up-to-date version of data. Replicas communicate only with the master. This is a constant interaction protocol [14] where each transaction is firstly executed at a replica. At commit time, if it is a read-only transaction it will directly commit; on the contrary the replica will send a message containing the transaction's snapshot version and its associated writeset. They use a data structure similar to ours, *ws_list*, so that the master uses it in the certification process. The master checks whether the delivered writeset intersects with any of the already committed writesets since the snapshot version of the delivered transaction. If so, the transaction is aborted (the replica receives a message). Otherwise, the writeset is included in the *ws_list* and the version is increased. Since a centralized certifier is a single point of failure, they propose a distributed certification algorithm. They assume an atomic broadcast facility to exchange messages since all replicas execute certification. Hence, all replicas need to maintain the same data structure as the master database in the centralized certifier. The k-bound GSI protocol already provides the distributed certification protocol proposed in [4] when no k is provided. Moreover, k-bound GSI permits to refine the outdateness of the snapshot version gotten by a transaction (ranging from GSI to SI). Finally, we are able to provide *serializable* isolation level with the same replication protocol.

In [15] the Database State Machine Approach (DBSM) is revisited in order to avoid the readset propagation of update transactions to all sites for certification (called DBSM*). They propose three different replication protocols. The first one is the readsets-free certification protocol where each replica executes transactions according to strict 2PL [3]. As its own name states, it only checks writesets intersection and does not prevent the *write-skew* anomaly [1]. Their second proposal is SI DBSM*, where all replicas have a SI DBMS, where they claim that SI is obtained but it turns out that only GSI is achieved, actually their proposal is very similar to [4]. Their last proposal is One-copy Serializable DBSM*, they took the ideas of [5] about interference free transactions: two transactions executing at different sites are serializable in DBSM* if their writesets do not intersect or one does not read what is written by the other. Thus, they propose to split the database into a number disjoint logical sets where each replica is the master of one or more sets. Transactions are executed at each replica following the strict 2PL rule. The database schema is increased with a control table containing one dummy row for each logical set. This control table is for conflict materialization of transactions exclusively reading that logical set (they update the corresponding row). Additional processing is required for transactions updating two or more disjoint sets; they can be serialized if they are certified at the same node but it cannot be serialized with interfering transactions executing at different replicas. Our approach does not use readset propagation too but it does not use logical set (excepting read operations to check their outdateness). k-bound GSI is able to provide GSI, SI and serializable without changing the underlying DBMS or setting partitions in the database.

A definition of One-Copy SI is proposed in [11]. They present a replication protocol for a database replication middleware architecture. Each replica contains a DBMS providing SI. Transactions are locally executed at each replica and

at commit time the writesets are atomically broadcast for distributed certification. Once a writeset is certified, it must be applied in the DBMS; however, it may be involved in a deadlock with local transactions and become aborted by the underlying DBMS. Thus, it must be reattempted until it gets finally committed. They propose an optimization that permits the concurrent execution of certified transactions whose writeset intersection is empty. Nevertheless, this optimization provokes holes in the *ws_list* that blocks local transactions for performing read operations until there are no holes (therefore, loosing the main advantage of SI). In k-bound GSI, when the SI isolation level is chosen, read-operations never get blocked (their up-to-date version is checked with the delivery of the T.ID message) and we do not permit holes in the execution of writesets.

5. CONCLUSIONS

This paper has presented a single middleware database replication protocol able to support different degrees of compliance of the SI level on top of DBMSs supporting SI. It starts with a relaxed GSI (similar to the PCSI described in [4]), passes through a 1CSI implementation ([4] also describes one implementation of this kind, but pessimistic since it always blocks a transaction when it starts, while ours is optimistic, re-initiating those transactions that have seen an invalid snapshot), and is also able to ensure a serializable level. This flexibility is specially interesting when complex interactive applications should be written, since some of their transactions may require different isolation levels being SI and serializable two of the most requested ones.

6. ACKNOWLEDGMENTS

This work has been supported by the Spanish Government under research grants TIC2003-09420-C02 and TIN2006-14738-C02.

7. REFERENCES

[1] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *ACM SIGMOD International Conference on Management of Data*, pages 1–10, San José, CA, USA, May 1995.

[2] J. M. Bernabé, R. Salinas, L. Irún, and F. D. Muñoz. Managing multiple isolation levels in a database replication protocol. Technical report, ITI-ITE-06/05, Instituto Tecnológico de Informática, Valencia, Spain, July 2006.

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

[4] 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.

[5] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[6] J. R. González de Mendívil, J. E. Armendáriz, J. R. Garitagoitia, L. Irún, and F. D. Muñoz. Non-blocking ROWA protocols implement generalized snapshot isolation using snapshot isolation replicas. Technical report, ITI-ITE-06/04, Instituto Tecnológico de Informática, Valencia, Spain, July 2006.

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

[8] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Computer Science Department, May 1994.

[9] 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.

[10] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333–379, Sept. 2000.

[11] Y. Lin, B. Kemme, M. Patiño, and R. Jiménez. Middleware-based data replication providing snapshot isolation. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 419–430, Baltimore, Maryland, USA, June 2005.

[12] F. D. Muñoz, J. Pla, M. I. Ruiz, L. Irún, H. Decker, J. E. Armendáriz, and J. R. González de Mendívil. Managing transaction conflicts in middleware-based database replication protocols. In *25th IEEE Symposium on Reliable Distributed Systems*, Leeds, UK, Oct. 2006.

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

[14] M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso. Database replication techniques: A three parameter classification. In *19th IEEE Symposium on Reliable Distributed Systems*, pages 206–217, Oct. 2000.

[15] V. Zuikeviciute and F. Pedone. Revisiting the database state machine approach. In *Workshop on Design, Implementation, and Deployment of Database Replication (in VLDB 2005)*, Aug 2005.