SIRC, a Multiple Isolation Level Protocol for Middleware-based Data Replication

R. Salinas, J.M. Bernabé-Gisbert, F.D. Muñoz-Escoí
Instituto Tecnológico de Informática
Camino de Vera s/n
46022 Valencia, Spain
{rsalinas, jbgisber, fmunyoz}@iti.upv.es

J.E. Armendáriz-Iñigo, J. R. González de Mendívil
Universidad Pública de Navarra
Campus de Arrosadía s/n
31006 Pamplona, Spain,
{enrique.armendariz,mendivil}@unavarra.es

***Abstract*—One of the weaknesses of database replication protocols, compared to centralized DBMSs, is that they are unable to manage concurrent execution of transactions at different isolation levels. In the last years, some theoretical works related to this research line have appeared but none of them has proposed and implemented a real replication protocol with support to multiple isolation levels. This paper takes advantage of our MADIS middleware and one of its implemented Snapshot Isolation protocols to design and implement SIRC, a protocol that is able to execute concurrently both Generalized Snapshot Isolation (GSI) and Generalized Loose Read Committed (GLRC) transactions. We have also made a performance analysis to show how this kind of protocols can improve the system performance and decrease the transaction abortion rate in applications that do not require the strictest isolation level in every transaction.**

I. Introduction

Nowadays, database applications often require support to multiple isolation levels. Actually, this necessity has been included as one of the several standard benchmark applications proposed by TPC: TPC-C [1]. In such benchmark, its New-Order, Payment, Delivery and Order-Status transactions require the ANSI *serializable* level [2], [3], and the same set of transactions requires that other transactions accessing the same data (besides the Stock-Level one, that is also included in the benchmark) use the *repeatable read* level [2], [3], whilst its Stock-Level transaction only demands the *read committed* (RC) level [2]. Modern centralized Database Management Systems (DBMS) can deal with that kind of applications but replicated ones are normally one isolation level oriented or, in the best cases, can support multiple protocols with different isolation levels but only one of them can be used at a time.

Few studies have been made about how to construct replication protocols with multiple isolation level support [4], [5], [6] but, as far as we know, none of them have culminated in the implementation of such a protocol in a real system.

In this paper we present a protocol, called SIRC, to be run in MADIS [7], a middleware that currently uses PostgreSQL as its underlying DBMS. Since PostgreSQL only supports the SI and RC isolation levels, SIRC is able to manage both GSI [8] and Generalized Loose RC (GLRC) [9] levels. We have chosen these isolation levels since they are the natural extensions of SI and RC semantics to a replicated setting. It is important to note that both allow to read from older committed versions as opposite to the latest one as provided in most commercial (i.e., also centralized) DBMSs. Moreover, RC, as proposed in [2] makes use of *short* read locks for reading implying the potential blocking of read operations as opposed to SI where only write operations do block.

The SIRC protocol has been designed according to the instructions described in [4] which consist in following a few steps. In the first one, a protocol with the strictest required isolation level must be selected. We have taken the SIR-SBD protocol presented in [10] because it was already implemented in our MADIS middleware. In the next steps, this protocol has to be modified in order to propagate the isolation level throughout the system, to ensure that we always know the isolation level requested by every transaction. Finally, we need to modify the transaction validation rules to consider the isolation level of all transactions being checked. We have also implemented the protocol in MADIS, to prove that our protocol reduces the abortion rate, specially in histories having a high rate of GLRC transactions. In the same way, it is interesting to check whether this low abortion rate is achieved at the risk of introducing higher overheads that penalize the response time of transactions.

This paper is structured as follows: the model being used in this paper is described in Section II. In Section III the SIR-SBD protocol is presented as the basis of our SIRC with support for two isolation levels, outlined in Section IV. An analysis of its performance in MADIS, i.e. the transaction response time and abortion rate, is shown in Section V. Section VI provides an outline of the correctness of our approach. Finally, conclusions end the paper.

II. System Model

The MADIS middleware [7] provides the necessary support to implement a suite of replication protocols in an interchangeable manner and serve as a testbed for them. It is developed in Java and has a JDBC interface to communicate with external applications, so that they will remain unaware of the replicated system. MADIS is also linked with the Spread [11] group communication system which provides a total order multicast [12]. We assume a fully replicated system composed of N replicas

$(R_1, \ldots, R_N)$ where each replica has an underlying DBMS that stores a physical copy of the database; PostgreSQL [13] has been used. This DBMS is a multiversion one (i.e. a new database version is generated each time a transaction is committed, we assume the version number is stored in a local replica variable called lastcommitted_tid) that locally supports the concurrent execution of RC and SI transactions. To keep consistent copies of the database a replication protocol is executed; in our case we assume that it follows the Read One Write All Available (ROWAA) approach [14]: a transaction is firstly executed at its delegate replica and at commit time its changes (denoted as writesets) are propagated to the rest of sites. Finally, MADIS includes a block detection mechanism that will be very useful to apply remote writesets for the SIR-SBD and, hence, to the SIRC protocols. In the rest of the paper, we will use SI and GSI (respectively RC and GLRC) in an interchangeable manner when referring to transactions executed in a replicated setting. Hence, otherwise stated, SI, and RC, will refer to the isolation level provided by the DBMS and the replication protocol.

III. SIR-SBD Protocol

SIR-SBD is based on a ROWAA certification-based [15] replication protocol, called SI-Rep, proposed for a middleware replicated database architecture in [16]. When the transaction requests its commitment (for simplicity, we do not consider aborted transactions), its writeset is locally collected and sent to all replicas using the total-order multicast. Since all replicas maintain a log of delivered *validated* writesets, and using a given decentralized validation technique, they are able to certificate such incoming transaction. If the validation succeeds, the incoming writeset is added to the log, and the delivered writeset is applied and committed (the delegate replica will directly commit). Otherwise, the writeset is discarded and the transaction is aborted in all replicas (indeed, only its delegate replica needs to abort it).

This protocol, presented in [10], was developed to take advantage of the block detection procedure presented in the same paper. This procedure is a per replica polling mechanism meant to detect local blocks between transactions at each replica. In order to do that, every running transaction must have a priority assigned by the protocol, which can be dynamically modified during the transaction execution (i.e., while it is being executed at the delegate replica, has broadcast the updates, is a remote one, ...). When the block detector finds a block between two transactions, the one with lowest priority is aborted. If both have the same priority it will be up to the replication protocol to decide which transaction should be aborted. This will be explained in the following.

SIR-SBD simplifies the SI-Rep protocol by taking advantage of the block detection schema to avoid deadlocks in the latter between the middleware instance at a replica and its underlying DBMS [16]. This deadlock may appear, e.g. if we use PostgreSQL [13], due to the use of write locks in their SI schedulers. This approach increases the performance since conflicts will be normally detected earlier. The block detector can detect them during the transaction execution and not in the certification step, once the commit operation is requested.

Initialization:
1. lastvalidated_tid := 0
2. lastcommitted_tid := 0
3. ws_list := ∅
4. tocommit_queue_k := ∅

I. Upon operation request for T_i from local client
1. If select, update, insert, delete
   a. if first operation of T_i
      - T_i.start := lastcommitted_tid
      - T_i.priority := 0
      - ***T_i.IL := get_il();***
   b. execute operation at R_k and return to client
2. else /* commit */
   a. T_i.WS := getwriteset(T_{ik}) from local R_k
   b. if T_i.WS = ∅, then commit and return
   c. T_i.priority := 1
   d. multicast T_i using total order

II. Upon receiving T_i in total order
1. obtain wsmutex
2. if $\exists\ T_j \in$ ws_list : T_i.start $<$ T_j.end $\wedge$ T_i.WS $\cap$ T_j.WS $\neq \emptyset$ $\wedge$ ***T_i.IL = SI***
   a. release wsmutex
   b. if T_i is local then abort T_i at R_k
   c. else discard
3. else
   a. T_i.end := ++lastvalidated_tid
   b. append T_i to ws_list
   c. append T_i to tocommit_queue
   d. release wsmutex

III. T_i := head(tocommit_queue)
1. if T_i is remote at R_k
   a. begin T_{ik} at R_k
   b. apply T_i.WS to R_k
   c. $\forall\ T_j$: T_j is local in R_k $\wedge$ T_j.WS $\cap$ T_i.WS $\neq \emptyset$
      /* if T_j has broadcast its WS$_j$ */
      /* it must follow its path */
      - abort T_j
2. commit T_{ik} at R_k
3. ++lastcommitted_tid
4. remove T_i from tocommit_queue

Fig. 1. SIRC algorithm at replica R_k

In this protocol (shown in Figure 1 without considering boldfaced lines), a transaction has initially the lowest priority (0) and gets a snapshot from its delegate replica (T_i.start = lastcommitted_tid). Once the application requests its commit, its priority increases to 1 before its writeset is broadcast (item I in Figure 1) using the total-order primitive. Once a writeset is delivered (item II) it must be validated against previously validated concurrent transactions. The validation process for SI transactions consists in applying the *First Committer Wins* rule, i.e. there is an empty writeset intersection between concurrent, though already validated, transactions (T_j.end $\neq \perp \wedge$ T_i.start $<$ T_j.end: $T_j \in$ ws_list); otherwise, the transaction got aborted, at the delegate replica or silently discarded,in the rest of replicas. If it succeeds the lastvalidated_tid will be increased and assigned to T_i.end and the writeset will be enqueued in ws_list and in another list (to_commit_queue). The latter governs the asynchronous application and commitment of the validated transaction at each replica. As mentioned before, the writeset application will abort all possible local transactions thanks to the block detector (item III). Finally, the lastcommitted_tid got increased.

IV. SIRC PROTOCOL

This protocol (introduced in Figure 1) has been designed to support the concurrent execution of transactions with two possible isolation levels: GSI [8] or GLRC [9]. Recall, as we have mentioned before, that this fits better than current replication solutions to applications that do not need a higher isolation level for all of their transactions. We have followed the four step process described in [4] to construct SIRC.

The first step consists in selecting a replication protocol providing the strictest level we need to support. This protocol will be modified to allow more relaxed isolation levels. In our case study, the highest isolation level we want to provide is GSI so we have selected the SIR-SBD protocol presented in Section III.

In the second step, the selected protocol is modified to ensure that it is able to know the isolation level of every transaction being executed. SIRC includes an additional field in the transaction ($T_i.IL \in \{SI, RC\}$ see item I.1.a in Figure 1) that is filled at transaction start time. Hence, the transaction is also executed at each replica with the proper isolation level.

In the third step, it must be ensured that every writeset broadcast has its transaction isolation level attached which it has been already done in item I.1.a. This is necessary to apply the correct validation rules for each isolation level, once the writeset is delivered to a replica. Recall that the isolation level had been written into T_i at the moment the client set it; hence, as it is multicast with the writeset, the replication protocol has access to it at any time during the lifetime of T_i.

Finally, the fourth step, the writeset validation process must be modified to consider the isolation level of all transactions being checked. In our protocol, a SI writeset must be validated as in the original SIR-SBD protocol. On the other hand, a delivered RC writeset will be directly validated since the total-order delivery avoids dirty writes [2]. The clue is that local conflicting transactions running under RC will get aborted (item III.1.c in Figure 1), no matter whether they have multicast their writesets or not. In the case of a local RC transactions still in their read and write phase it will be aborted and the user will get notified by this fact and, hence, re-attempted. Otherwise, when the transaction writeset has already been broadcast, the transaction is silently rolled back, i.e. the user does not get notified, since such transaction must be successfully applied (and committed) at the moment of its total-order delivery [12]. The correctness of this behavior will be argued in Section VI-B. Note that in RC a transaction T_1 will be able to overwrite the updates of another concurrent one (T_2) if T_1 has always read committed values and T_2 has committed.

V. PERFORMANCE RESULTS

We wanted to prove that, in histories containing transactions having different isolation requirements, using two levels - instead of just the highest one- leads to lower abortion rates, and thus to a better performance.

For the performance tests we have used a cluster consisting of 4 nodes. Each node runs a MADIS server [7] working on the top of a PostgreSQL 8.1 DBMS [13]. A client is run on every server node. Every client performs exactly the same amount of work.

Each client keeps a dynamic pool of working threads, that try to satisfy a targeted number of transactions per second.

Each transaction consists of 8 writes on a single table containing 10000 data items. The whole row set is divided into a hot-spot [17] and a non-conflicting region. The non-conflicting region is statically divided among the different clients, so that the conflict level can be accurately determined by means of the hot-spot parameters. In our case we have had one conflicting write out of 40.

If a transaction is aborted because of a concurrency problem, it is retried until its commitment, without waiting.

An inter-update delay summing up 500 ms per transaction has been introduced in order to extend the total transaction lifetime, which models some kind of workload at the client side, and increases the conflict rate without yielding a too heavy system load, which would tend to produce less stable time measurements. We have tested SIRC under different loads, namely a total of 2 and 4 transactions per second in the global system.

As we see in Figure 2.b, the abortion rate can be significantly decreased by using RC when we do not need a higher level. There is a low percentage of RC transactions that abort. Note that in a replicated system there can be multiple transactions that might concurrently update the same datum in different replicas and that lead to the abortion of some of these transactions when they are validated at commit time. This is an inherent problem of the certification-based replication protocols, as already explained in [15]. In a centralized setting, this does not happen since the local concurrency control blocks such concurrent accesses and all conflicting transactions can commit.

Additionally, the RC transaction response time is also better than the SI's one. In Figure 2.a, we can see that such response time ranges from 1040 ms for SI transactions to 740 ms for RC ones, with a global load of 2 TPS. So, if some application that could have used RC transactions is compelled to use the SI level due to the lack of support for more than one level in the common case for replicated architectures, it is paying around a 40% of unnecessary overhead in the response time or such transactions. Moreover, 2 TPS is a very light load. Things are still more favorable to multi-level support when load is increased. Thus, with 4 TPS the RC response time remains almost the same (765 ms) but the SI one is increased till 1295 ms. In this second case, such overhead reaches almost 70% of the RC time. Such overhead is partially explained by the abortion rate, that leads SI transactions to be retried. Using a transaction load with null abortion rate, the response time differences between the two considered isolation levels would not have been so big, but when there are any conflicts it is worth providing support for multiple isolation levels. For instance, with an 80% of RC transactions, there is a 4% abortion rate (caused entirely by the SI transactions) and the mean response time is still a 6% greater than for pure RC transactions (for both loads, 2 and 4 TPS). This still implies

that for such abortion rate SI transactions had a 30% bigger response time than RC ones, since they were a 20% of the overall transaction population. So, $0.3 * 0.2 = 0.06$ and this provides the justification of such 6% overhead.

As we expected, regardless of the system load, allowing for the use of a less strict isolation level yields a better performance, while transactions requiring a higher level still get their consistency requirements fulfilled.

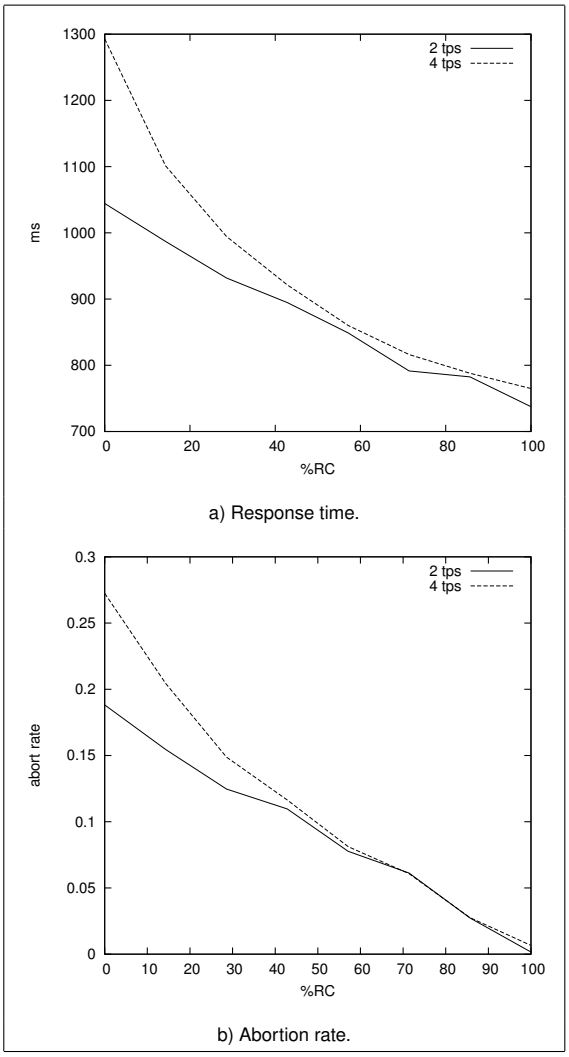

Fig. 2. Experimental results of SIRC.

VI. Correctness Discussion

In order to prove the correctness of this kind of algorithms it is necessary to prove that any transaction execution eventually finishes committing or aborting. Consistency must be ensured also, that is, once executed any given set of transactions, all active nodes must have the same values in their local database. Finally, we also need to prove that all transaction isolation level guarantees are ensured. During this section, some remarks, definitions and lemmas will be given in order to make the correctness process easier.

It is easy to see that all transactions execution will conclude since any local deadlock will be solved by local DBMSs and deadlocks between remote and local transactions are impossible since the block detector will abort any local transaction blocking a remote one. So, all transactions will be aborted locally in step 1, aborted in all nodes in the validation step or applied and committed in step 3.

Next, we need to prove that all nodes will reach the same state when executing a set of transactions. We know that the group communication system ensures that all nodes will deliver all writesets in the same order. Moreover, the algorithm validates writesets in delivering order so, if we prove that all nodes validate the same writesets and all validated writesets are applied in the same order then we will know that all nodes will eventually reach the same state. The first proof is solved in Lemma 3 and the second one is explained in Remark 1.

Since SIRC supports two isolation levels, SI and RC, the third correctness step is divided in two theorems. The first (Theorem 1) proves that all SI transaction guarantees are ensured and the second one (Theorem 2) do the same for RC transactions.

As explained before, every transaction T_i is initially executed at its delegate replica R_i in SIRC. The execution begins when the first operation of T_i arrives to R_i.

Definition 1 (Transaction Begin Operation)**.** *We define* $\mathbf{b_i}$ *as the first operation of transaction* T_i.

Once $\mathbf{b_i}$ arrives to R_i, lastcommitted_tid value is stored in T_i.start as the start timestamp of T_i. lastcommitted_tid is incremented in the algorithm every time a transaction is committed (item III.3) and its initial value is 0. Therefore, it has in every node the number of locally committed transactions. When $T_i.WS$ is delivered, if its validation process succeeds, lastvalidated_tid is stored in T_i.end as the validation timestamp. Hence, lastvalidated_tid has in every node the number of locally validated writesets.

Notice that all validated writesets are enqueued in to_commit_queue (item II.3.c of Figure 1) in the same order they are validated. In item III, the writesets are applied keeping the same order they have in the queue.

Remark 1 (Commit Ordering)**.** *Transactions are committed in the same order they are validated.*

It is important to note that when a transaction T_i is committed, its T_i.end will be equal to lastcommitted_tid. This is formally stated in the next lemma.

Lemma 1. *When some writeset* T_i*.WS is applied in some node* R_k*,* T_i*.end =* lastcommitted_tid *in* R_k.

Proof: When T_i is validated in some node R_k, lastvalidated_tid has the number of writesets validated until this moment (including T_i.WS) and its value is assigned to T_i.end. On the other hand, lastcommitted_tid has the number of transactions committed in R_k at every moment. As writesets

are committed in the same order they are validated and all validated writesets commit, the same set of transactions have committed when T_i commits than transactions have been validated when T_i is validated. So, when T_i commits, T_i.end = lastcommitted_tid. ■

Lemma 2 (Start and Commit Order)**.** *Given two transactions T_i and T_j, and being R_i the delegate replica of T_i, T_j commits in R_i before T_i starts iff T_i.start $\geq$ T_j.end in R_i.*

Proof: Proof 1: if T_j commits in R_i before T_i starts then T_i.start $\geq$ T_j.end in R_i. As previously said, when some transaction T_i starts at R_i, T_i.start = lastcommitted_tid. We also know that T_j.end = lastcommitted_tid when T_j commits. Since T_j commits before T_i starts and lastcommitted_tid is only incremented and never decremented in the protocol, T_i.start $\geq$ T_j.end in R_i.
Proof 2: if T_i.start $\geq$ T_j.end in R_i then T_j commits in R_i before T_i starts. Recall that T_i.start = lastcommitted_tid when $\mathbf{b_i}$ arrives to R_i, T_j.end = lastcommitted_tid when T_j commits (including the increment produced by T_j's commit) and lastcommitted_tid is only incremented when some transaction commits. Since T_i.start $\geq$ T_j.end, T_i sees at its start in R_i at least the same number of commits that T_j sees in R_i when commits, including its own commit. T_i.start must observe T_i's commit increment on lastcommitted_tid and, hence, T_j must have committed in R_i when T_i starts. ■

Notice that all nodes deliver writesets in the same order, due to total order broadcast, validate them in delivery order and validated ones are applied in validation order. Therefore, if for every writeset T_i all nodes reach to the same decision in its validation process, all nodes will commit the same writesets in the same order and, hence, all nodes will see the same database versions and the same lastcommitted_tid values.

A given transaction writeset T_i.WS validation process (item II.2 of Figure 1) depends on the following validation actions (i.e. T_i is aborted iff these 3 conditions are true):
1. Result of T_i.WS $\cap$ T_j.WS $\neq \emptyset$ for at least one writeset T_j.WS $\in$ ws_list.
2. Result of T_i.start $<$ T_j.end for at least one of the T_j that made Condition 1 true.
3. Isolation level of T_i is SI.

Lemma 3 (Validation Decision)**.** *Given a transaction T_i, all nodes arrive to the same decision about T_i.WS in the validation process.*

Proof: By induction over the length of ws_list.
• Induction base: The property holds for the initial state, i.e. ws_list $= \emptyset$, and T_i.WS is the first total-order delivered writeset. ws_list is empty when T_i.WS reaches at the validation process so the checking of conditions 1 and 2 fails in all nodes and T_i.WS is validated.
• Induction step: We assume that ws_list is the same for all the replicas when $(n-1)$-th writeset was validated (i.e. they have validated the same set of writesets) and T_i.WS is the n-th writeset being total-order delivered. Therefore, validation action 1 comparison will arrive to the same decision about T_i.WS in all nodes since it depends on the content of ws_list. Recall that all validated writesets are applied and validated in the same delivering order, so all T_j.WS $\in$ ws_list will have the same T_j.end. T_i.start is established by T_i delegate replica R_i and included in T_i.WS when it is broadcast, so all replicas see the same T_i.start value and T_i.start $<$ T_j.end comparison will produce the same result in all nodes. Finally, T_i isolation level is assigned also in delegate replica and propagated in T_i.WS, so again all replicas see the same value. Conclusion: all nodes arrives to the same decision in T_i.WS validation process. ■

Definition 2 (Concurrent Transactions)**.** *Two transactions T_i and T_j are concurrent if T_i.start $<$ T_j.end and T_j.start $<$ T_i.end.*

Definition 3 (Overlapping Transactions)**.** *Two transactions T_i and T_j overlap if T_i and T_j are concurrent and T_i.WS $\cap$ T_j.WS $\neq \emptyset$.*

Since all nodes apply the same writesets in the same delivering order (Lemma 3), all nodes will produce equivalent histories. Therefore, if every transaction isolation level guarantees are also ensured then our protocol must be correct.

A. All SI transaction isolation guarantees are ensured in SIRC

Given a SI transaction T_i, we must guarantee that:
1. T_i sees every data written by transactions that committed before the start of the former.
2. T_i does not see modifications carried out by transactions that have not finished before its starting moment.
3. A SI transaction T_i aborts if it overlaps with some concurrent validated transaction T_j (*First Committer Wins*).

Lemma 4 (Snapshot 1)**.** *T_i sees every data written by transactions that committed before the start of the former.*

Proof: Suppose that R_i is T_i delegate replica. By Lemma 2, for all T_j committed in R_i before T_i starts, T_i.start $\geq$ T_j.end. Since the R_i local DBMS provides SI, T_i will see in its snapshot all updates made by T_j since it has committed before $\mathbf{b_i}$ arrives at the local DBMS. ■

Lemma 5 (Snapshot 2)**.** *T_i does not see modifications carried out by transactions that have not finished before the starting moment.*

Proof: In this case, by Lemma 2, for all T_j committed after T_i started, i.e. T_i.start $<$ T_j.end. Again, the local DBMS ensures that T_i does not see T_j if the last one has committed after T_i started. ■

Lemma 6 (First Committer Wins)**.** *A SI transaction T_i aborts if overlaps with some concurrent validated transaction T_j.*

Proof: If T_i overlaps with T_j and T_j is validated before T_i reaches the validation process, T_j.end $>$ T_i.start and T_i.WS $\cap$ T_j.WS $\neq \emptyset$. In T_i.WS validation process, T_j.WS will also be in ws_list (by definition, T_j is a validated writeset) so validation actions 1 and 2 are true. Since T_i is a SI transaction by definition, validation action 3 is also true and hence T_i.WS

will be discarded in all nodes and aborted at its delegate replica, as explained in item II.2 of Figure 1. ■

Theorem 1 (SI Theorem)**.** *All SI transaction isolation guarantees are ensured in SIRC.*

Proof: By Lemmas 3- 6, we have proved this theorem. ■

B. All RC transaction isolation guarantees are ensured in SIRC

In a T_i RC transaction the following conditions must be guaranteed:
1. T_i never sees uncommitted data of another transaction T_j.
2. T_i never overwrites uncommitted data of another transaction T_j.

Lemma 7 (Uncommitted Reads)**.** *T_i never sees uncommitted data of another transaction T_j.*

Proof: As we said in previous sections, every DBMS is supposed to provide locally the isolation level needed by every transaction. This ensures that T_i never will see uncommitted data in reads performed in its delegate replica. Since our protocol is a ROWAA one, all the reads of a given transaction are performed in the local node and hence the local DBMS ensures that this data belongs to a committed state and hence the first restriction is guaranteed. ■

Lemma 8 (Uncommitted Writes)**.** *T_i never overwrites uncommitted data of another transaction T_j.*

Proof: In this case, writes are performed in all nodes so we need to distinguish between local and remote transactions:
• T_i and T_j are local: the local DBMS resolves the conflict locally since it provides RC.
• T_i is local and T_j is remote: if both transactions do not overlap, none of them can overwrite uncommitted data of the other so, from now on, we suppose that T_i overlaps with T_j. If T_i tries to overwrite data written by T_j, local DBMS will block T_i until T_j finishes, so it will never overwrite uncommitted data. If T_j finds a T_i write, the local DBMS will block T_j and our block detection procedure in item III.1.c of Figure 1 will abort T_i. If this happens after T_i has sent its own writeset, T_j.WS will be eventually delivered and applied after T_j. In other case, i.e. when T_i is still in its read and write phase, T_i abortion will be forwarded to the client. In both cases, the uncommitted data overwrite is avoided.
• T_i is remote and T_j is local: is equivalent to the previous case.
• T_i and T_j are remote: T_i.WS and T_j.WS will be applied in delivery order and, hence, none of them can overwrite any uncommitted data from the other (since the overwritten data was already committed). ■

Theorem 2 (RC Theorem)**.** *All RC transaction isolation guarantees are ensured in SIRC.*

Proof: It is proved with Lemmas 3, 7 and 8. ■

Notice that no validation process is needed to ensure RC in the algorithm and this is the reason why validation process conditions are only taken into account in SI transactions.

VII. Conclusions

In this paper we have presented an algorithm, SIRC, which is an evolution of SIR-SBD [10], originally suited for executing GSI transactions [8]. SIRC supports the concurrent execution of SI and RC transactions. SIRC reflects that it is possible to design a single replication protocol supporting different isolation levels. We have shown that this solution decreases the abortion rate in applications with a high rate of RC transactions in a replicated setting; reflecting a similar behavior to RC in a centralized environment. Moreover, SIRC does not increase the overhead introduced by SIR-SBD. Nevertheless, SIRC is a basic replication protocol that can include several optimizations. We have introduced it in this way because we wanted to emphasize the correctness of our approach in the definition of SIRC. Hence, it serves as a starting point for the development of new optimized replication protocols supporting these two isolation levels.

References

[1] Transaction Processing Performance Council, "TPC Benchmark C - standard specification. Version 5.8." 2007, http://www.tpc.org.
[2] 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.
[3] A. Adya, B. Liskov, and P. E. O'Neil, "Generalized isolation level definitions." in *ICDE*, March 2000.
[4] J. M. Bernabé, R. Salinas, L. Irún, and F. D. Muñoz, "Managing multiple isolation levels in middleware database replication protocols." in *ISPA*, ser. LNCS. Springer, 2006.
[5] 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.
[6] J. E. Armendáriz, J. R. Juárez, J. R. González de Mendívil, H. Decker, and F. D. Muñoz, "k-Bound GSI: A flexible database replication protocol," in *SAC*. ACM, 2007.
[7] L. Irún, H. Decker, R. de Juan, F. Castro, J. E. Armendáriz, and F. D. Muñoz, "MADIS: a slim middleware for database replication," in *Euro-Par.*, ser. LNCS. Springer, 2005.
[8] S. Elnikety, F. Pedone, and W. Zwaenepoel, "Database replication providing generalized snapshot isolation," in *SRDS*, 2005.
[9] J. M. Bernabé, J. E. Armendáriz, R. de Juan, and F. D. Muñoz, "Providing read committed isolation level in non-blocking ROWA database replication protocols," in *JCSD*, 2007, accepted for publication.
[10] F. D. Muñoz, J. Pla, M. I. Ruiz, L. Irún, H. Decker, J. E. Armendáriz, and J. R. G. de Mendivil, "Managing transaction conflicts in middleware-based database replication architectures." in *SRDS*, 2006.
[11] Spread, "The Spread toolkit." 2007, http://www.spread.org.
[12] G. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: a comprehensive study." *ACM Comput. Surv.*, vol. 33, no. 4, pp. 427–469, 2001.
[13] PostgreSQL, "PostgreSQL, the world's most advanced open source database." 2007, http://www.postgresql.org.
[14] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha, "The dangers of replication and a solution." in *SIGMOD*, 1996.
[15] M. Wiesmann and A. Schiper, "Comparison of database replication techniques based on total order broadcast." *IEEE TKDE*, vol. 17, no. 4, pp. 551–566, 2005.
[16] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris, "Middleware-based data replication providing snapshot isolation," in *SIGMOD*, 2005.
[17] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-R, a new way to implement database replication." in *VLDB*, 2000, pp. 134–143.