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

Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia (Spain)
{rsalinas,jbgisber,armendariz,fmunyoz}@iti.upv.es

Technical Report ITI-ITE-07/03
SIRC-Rep, a Multiple Isolation Level Protocol for Middleware-based Data Replication Architectures

Instituto Tecnológico de Informática, Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, Spain
E-mail: {rsalinas, jbgisber, armendariz, fmunyoz}@iti.upv.es

Technical Report: ITI-ITE-07/03
February 6, 2007

Abstract

One of the weaknesses of replicated protocols, compared to centralized ones, 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 the SIRC-Rep, a protocol that is able to execute concurrently both Generalized Snapshot Isolation (SI) and Read Committed (RC) 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.

1 Introduction

Nowadays, applications often require support to multiple isolation levels. 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 [1, 2] but, as far as we know, none of them have culminated in the implementation of a protocol in a real environment.

In this paper we present a protocol able to concurrently execute GSI [3] and RC transactions. We have chosen these isolation levels since SI semantics is quite close to that of the serializable level and it fits better modern replication techniques, whereas RC is normally enough for non-critical transactions and, indeed, is the default level in many DBMSs.

This protocol has been designed following the instructions described in [2] consisting in 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 [4] because it was already implemented in our middleware [5, 6, 7]. 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 our middleware MADIS, to prove that our protocol reduces the abortion rate (and hence the response time), specially in histories having a high rate of RC transactions.

This paper is structured as follows. In the next section, the model being used in this paper is described. In section 3 the SIR-SBD protocol is presented as the basis of our SIRC-Rep with support for two isolation levels, outlined in section 4. In section 5 some performance analysis of this protocol are shown. Section 6 is focused in a light correctness demonstration and, finally, section 6 includes the conclusions.

2 System Model

The MADIS middleware provides the necessary architecture to execute database replication protocols. It is developed in Java and has a JDBC interface to communicate with external applications. MADIS is also linked with the Spread group communication system which provides a total order multicast.

In this paper we also suppose that the underlying DBMS in every node supports locally the execution of RC and SI transactions concurrently, and that the replication protocols use a ROWAA approach [8].

In the protocol SIR-SBD is also assumed that the underlying database includes a block detection procedure, as explained in [4].

In the rest of this article, we will write “SI” meaning “GSI”, and by “RC” we will refer to a kind of replicated RC, which is not necessarily equivalent to a single-copy Read Committed level, since transactions are not blocked at their beginning [9].

3 SIR-SBD protocol

This protocol, presented in [4], was developed to take advantage of the block detection procedure presented in the same paper.

This procedure is an asynchronous mechanism meant to detect local blocks between transactions. In order to do that, every running transaction must have a priority assigned by the protocol, which can be modified during the transaction execution. When the procedure detects a block between two transactions, the one with lowest priority is aborted. If both have the same priority and only one of them is local then this will be aborted. If its writeset has been already broadcast, it will continue its path and will be applied if no conflicts are detected. If the writeset has not been broadcast, the transaction must be relaunched by the client side. Otherwise, the block detector will do nothing, and then the commit or abort decision is delegated to the protocol transaction validation process.

SIR-SBD simplifies the SI-Rep protocol proposed in [10] by taking advantage of the block detection schema to avoid deadlocks between the middleware and the local DBMS [10] which can appear in some DBMSs (like PostgreSQL or Oracle) due to the use of locking techniques in their SI protocols. This solution also 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.

In this protocol (shown in Figure 1 if the lines in boldface are not considered), a transaction has initially the lowest priority (0). Once the application requests its commit, its priority increases to 1 before its writeset is broadcast (step I). Once a writeset is delivered (step II) it must be validated against previously validated concurrent transactions and aborted if some write conflict appears. If not, the writeset is enqueued in the validated writesets list (ws_list) and in every node list of writesets waiting to commit (to_commit). The writesets of the to_commit lists are applied asynchronously and in order. Once a writeset has been applied, the block detection procedure will abort every conflicting local transaction (step III). As
we explained before, a local transaction must be re-
launched if got aborted before its writeset was broad-
cast.

4 SIRC-Rep protocol

This protocol (shown in Figure 1) has been con-
structed to allow for the concurrent execution of
transactions with two possible isolation levels: gen-
eralized snapshot isolation or read committed. This fits
better than current replication solutions that kind of
problems that do not need a high isolation level for
all of its transactions.

To construct SIRC-Rep we have followed the
schema presented in [2] consisting in four steps.

In the first one, we need to select a protocol pro-
viding the strictest level we need to support. This
protocol will be modified to allow more relaxed iso-
lation levels. In our case study, the highest isolation
level we want to provide is SI so we have selected the
SIR-SBD protocol presented in section 3.

In the second step, the selected protocol is modified
to ensure that we know the isolation level of every
transaction being executed. To do that, we have to
forward the isolation level change request to the local
DBMS. The level change must appear before any read
or write once the transaction starts (last line of step
I.1.a).

In the third step, we have to ensure that every
writeset has its transaction isolation level attached
(already done in step I.1.a). This is necessary to ap-
ply the correct validation rules, once the writeset is
delivered in every node, according to the transaction
isolation level. This level had been written into the
data structure $T_i$ at the moment the client set it, so
that we will have access to it all along the transac-
tion’s lifetime.

In the fourth and last step, the writeset validation
process must by modified to consider the isolation
level of all transactions being checked. In our pro-
tocol, a SI writeset must be validated against previ-
ous concurrent transaction writesets, like in the orig-
inal SIR-SBD protocol. On the other hand, a RC
writeset will be directly validated since the writeset
application order, based on writeset total-order de-
livery, avoids dirty writes (defined in [11], a dirty
write appears when a transaction modifies a value
updated by a not-yet-committed transaction). Since
local DBMSs ensure that their local transactions’ iso-
lation guarantees will be respected, the only possi-
ble write-write conflicts will appear between a local
transaction and a remote one. Since the block de-
tection procedure will abort every local transaction
blocking a remote one (III.c), the consistency is en-
sured, and the deadlocks between the middleware
and the DBMS are prevented.

5 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 perfor-
mance.

For the performance tests we have used a cluster
consisting of 4 nodes. Each node runs a MADIS
server working on the top of a PostgreSQL 7.3 DBMS.
A client is run on every server node. Every client per-
forms exactly the same amount of work. Each client per-
forms exactly the same amount of work. Each client
creates some threads that execute concurrent trans-
actions on the local MADIS server. The access pat-
tern consists of a certain number of random updates
on a table having 10000 rows. Every thread chooses
a random row of this table and updates a number
of consecutive rows. The transaction rate has been
bounded in our experiments to one transaction ev-
ery 4 seconds in each thread. If a thread has some
free time at the end of a work cycle, it waits. If a
transaction is aborted, it is retried until its commit-
ment, without waiting. Specifically, each launched
thread in our experiments has performed 15 updates
on 3 different tables. An inter-update delay summing
up 1 second 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. We have tested SIRC-Rep
under different loads, namely 2, 4 and 6 client threads
per node, which respectively correspond to a global
system load of 2, 4 and 6 transactions per second.

As we see in Figure 2, the abortion rate can be
Initialisation:
1. lastvalidatedtid := 0
2. lastcommittedtid := 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 := lastcommittedtid
      - T_i.priority := 0
      - T_i.IL := getil();
   b. execute operation at R_k and return to client
2. else /* commit */
   a. T_i.WS := getwriteset(T_i_k) 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 ∃ T_j ∈ ws_list : T_i.start < T_j.end ∧ T_j.WS ∩ T_i.WS ≠ ∅ ∧ T_j.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 := ++lastvalidatedtid
   b. append T_i to ws_list
   c. append T_i to tocommit_queue_k
   d. release wsmutex
III. T_i := head(tocommit_queue_k)
1. if T_i is remote at R_k
   a. begin T_i_k at R_k
   b. apply T_i.WS to R_k
   c. ∀ T_j : T_j is local in R_k ∧ T_j.WS ∩ T_i.WS ≠ ∅ /* if T_j has broadcast its WS, */
      /* it must follow its path */
      - abort T_j
   2. commit T_i_k at R_k
   3. ++lastcommittedtid
   4. remove T_i from tocommit_queue_k

Figure 1: SIRC-Rep algorithm at replica R_k

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 date 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 [12]. In a centralized setting, this does not happen since the local concurrency control blocks such concurrent accesses and all conflicting transactions can commit.

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.

6 Correctness Discussion

In order to prove the correctness of this kind of algorithms it is necessary to prove that every isolation requirement of every transaction is guaranteed. Therefore, the demonstration has been split up into two parts. First, we demonstrate that the algorithm fulfills the isolation requirements of a SI transaction.
Secondly, we demonstrate the same for the case of RC transactions.

Given a SI transaction $T_i$, we must guarantee that:

1. $T_i$ sees every data written by transactions that committed before the start moment of the former.

2. $T_i$ does not see modifications carried out by transactions that have not finished before the starting moment.

3. $T_i$ aborts if its writeset overlaps with the writeset of some concurrent transaction that finished before it (First Committer Wins).

In SIRC-Rep, the timestamps used to keep the accounts of the the start and the end of a transaction are increasing counters. When $T_i$ starts in some node $R_k$, the middleware takes the last committed transaction tid in $R_k$ as $T_i$ start time. Upon writeset application, the counter referencing the last committed transaction is increased by one.

At the same time, the local DBMS in $R_k$ guarantees that $T_i$ sees all writes made by every transaction committed before $T_i$ started and do not see any modification made by any other transaction (that is, transactions that have not finished before $T_i$ start). This ensures the first two restrictions.

The third restriction is guaranteed thanks to II.2 and III.1.c steps of the protocol. II.2 ensures that a SI writeset $WS_i$ is aborted if a conflict is detected with any previous concurrent writeset $WS_j$, that is, if $WS_i \cap WS_j \neq \emptyset$ and $WS_j$ has finished after $WS_i$ start. On the other hand, III.1.c ensures that, given a node $R_k$ when a validated writeset $WS_j$ is going to be applied, every transaction $T_i$ local in $R_k$ which holds a lock conflicting with $WS_j$ will be aborted. If $T_i$ has already sent its writeset, its $WS_i$ will follow its path and finally aborts in validation step II.2. The $T_i$ local abort in $R_k$ will then be considered a "dummy" abort and will not be propagated. In other case, the local abort will be considered an effective abort and propagated to the user.

Given a RC transaction $T_i$, we must guarantee that:

- $T_i$ never sees uncommitted data.
- $T_i$ never overwrites uncommitted data.

As we said in previous sections, we suppose that every node local DBMS provides locally the isolation level needed by every transaction. This ensures that $T_i$ never will see uncommitted data in reads performed in its local node. Since our protocol is a Read One Write All Available, 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.

Notice that this is normally ensured by local DBMS using long read locks for both (RC and SI) kinds of transactions but short read local locks only for Read Committed ones. This implies that, with this solution, a remote transaction will never be blocked by a local read. But in the cases where this phenomenon would be possible, the local transaction will be aborted and relaunched if its writeset has not been yet broadcast or only locally "dummy" aborted until its WS can be delivered and applied.

Unlike reads, in a ROWAA algorithm like this, writes are propagated to all replicas so we need to ensure that a write never overwrites uncommitted reads in any replica. To prove that, we will first focus on the node local to $T_i$ and later on the nodes where $T_i$ is remote.

If $T_i$ is local to $R_k$ and another transaction $T_j$ (RC or SI) executes locally in $R_k$, again the local $R_k$ DBMS will ensure that $T_i$ never will see uncommitted updates made by $T_j$, normally by blocking $T_i$ until $T_j$ finishes. The same holds if $T_j$ is remote in $R_k$ and if $T_i$ tries to read a data modified by $WS_j$ because $T_i$ will be blocked until $WS_j$ is applied and committed.

But, what happens if $T_i$ is a remote transaction and it gets blocked by a local one $T_j$? Since total order multicast ensures that all writesets are delivered, and hence applied, in the same order in all replicas, $T_j$ writeset $WS_j$ will be applied after $WS_i$ since we are now applying $WS_i$ and $T_j$ is still in execution. But if $WS_j$ gets blocked by a $T_j$ lock, $WS_i$ never will be applied before $T_j$ finishes and hence a deadlock between de middleware and the DBMS appears.

To solve that we have to abort $T_j$ and this is made in the step III.1.c by the block detection procedure.
Again, if \( WS_j \) has been broadcast we only have to do a “dummy” abort. In other case, we need to relaunch \( T_j \).

Finally, if \( T_i \) and \( T_j \) are remote, their writesets will be applied in delivery order so the second restriction is always guaranteed.

As an example, suppose two transactions \( T_i \) and \( T_j \) which read and modify \( X \), being \( T_j \) a Read Committed one (\( T_i \) can be Read Committed or Snapshot Isolation). If we execute them concurrently in a centralized DBMS like PostgreSQL and \( T_i \) is the first to get the lock on \( X \), \( T_j \) will get blocked until \( T_i \)’s commit. Then, \( T_j \) will be able to perform its update. Depending on when the reads are performed, there are two possible kinds of histories:

\[
\begin{align*}
(H1) & \quad b_i r_i(x_0) b_j w_i(x_1) c_r_j(x_1) w_j(x_{j1}) c_j \\
(H2) & \quad b_i r_i(x_0) b_j w_j(x_0) w_i(x_1) c_r_j(x_{j2}) c_j
\end{align*}
\]

In \( H1 \), \( T_i \) performs its writes before \( T_j \) reads. \( T_j \) would get blocked trying to read \( X \) until \( T_i \) commits and releases the lock. In \( H2 \), \( T_j \) reads before \( T_i \) writes \( X \) and does not get blocked until it tries to write over \( X \) and finds \( T_i \)’s write lock.

Let us now try to execute the same example using our SIRC-Rep algorithm, supposing that the underlying DBMS of every node is the centralized one used before. If \( T_i \) and \( T_j \) are local to the same node, their local DBMS will resolve the conflict as we have shown before. As \( T_i \) writes before \( T_j \), \( WS_i \) will be broadcast and applied before \( WS_j \) since \( T_j \) will be blocked until \( WS_j \) is applied locally and the lock released. The resulting history will be equivalent to \( H1 \) or \( H2 \) depending on if \( T_j \) tries to read \( X \) before or after \( T_i \) gets locally the lock on it.

In contrast, suppose now that \( T_i \) is local to node \( R_i \), \( T_j \) is local to node \( R_j \) and \( R_i \neq R_j \). Suppose also that \( WS_i \) is delivered before \( WS_j \) (in other case the histories will be equivalent to the centralized ones being \( T_j \) the first in committing). In this case, \( T_i \) will have the lock on the \( R_i \) copy of \( X \) and \( T_j \) will have the lock on \( X \) in \( R_j \). Since \( WS_i \) will be delivered first in both nodes, in \( R_i \) this writeset will be applied without problems because only \( T_i \) holds locks. These locks will be released upon \( T_i \)’s commit so \( WS_j \) will eventually be applied without problems in \( R_i \).

If \( T_j \) is executed in \( R_j \) after \( WS_j \) is applied in this node, the resulting history will be \( H1 \) in \( R_i \) since \( T_i \) would have seen \( WS_i \) updates. In other case, resulting history will be \( H2 \).

But in \( R_j \), since \( WS_j \) is the first writerset applied, two possible situations can occur. In the first one, \( T_j \) has not started when \( WS_i \) is applied or has started but has not tried to write \( X \) yet and hence does not hold the lock. In this case, \( WS_i \) will be applied without problems in \( R_j \) and eventually \( WS_j \) will do the same. The resulting history in \( R_j \) will be equivalent to \( H1 \) or \( H2 \) depending on the moment \( T_j \) has performed the read at, that is, after or before \( WS_i \) application in \( R_j \) (like in \( R_i \)).

In the second situation, \( T_j \) writes in \( R_j \) before \( WS_i \) is applied and this implies that \( WS_i \) get blocked when it tries to modify \( X \). If we do nothing, \( WS_i \) will remain blocked until \( T_j \) finishes its execution, but this cannot happen until \( WS_j \) is applied on \( R_j \) and, as we have shown before, this will never happen before \( WS_i \) is applied, so we have a deadlock between the DBMS and the middleware. The block detection procedure will solve that in III.1.c by aborting \( T_j \) in \( R_j \), thus allowing \( WS_i \) to finish. If \( T_j \) has not sent its \( WS_j \) when aborted then it will be executed again and see \( WS_i \) changes. The resulting history in \( R_i \) and \( R_j \) will be then equivalent to \( H1 \). If \( T_j \) has sent its \( WS_j \) then the local transaction will be dummy-aborted but \( WS_j \) will be delivered and applied. In this case, since \( T_j \) has not seen \( WS_i \) changes when \( WS_j \) is constructed, the resulting history will be equivalent to \( H2 \).

To conclude, we have seen that, under SIRC-Rep, both SI and RC transactions are run having the guarantees that their isolation level requires fulfilled. The example given illustrates how the combination of the local DBMS, the block detector and the certification step (II.2) ensure the correct behaviour, while avoiding incidental deadlocks between the middleware and the local DBMS.

7 Conclusions

In this paper we have presented an algorithm which supports the concurrent execution of SI and RC transactions. We have also shown that this solu-
tion noticeably reduces the abortion rate in applications with a high rate of RC transactions. We have also shown that it is possible to design algorithms with multiple isolation support which has been until now a weakness of replicated database management systems when compared with centralized ones.

References


