IMPLEMENTING DATABASE REPLICATION PROTOCOLS BASED ON O2PL IN A MIDDLEWARE ARCHITECTURE

J.E. Armendáriz, J.R. Juárez, J.R. Garitagoitia, J.R. González de Mendívil
Dpto. de Matemática e Informática
Universidad Pública de Navarra
Campus Arrosadía s/n, 31006 Pamplona, Spain
email: {enrique.armendariz, jr.juarez, joserra, mendivil}@unavarra.es

F.D.Muñoz-Escoí
Instituo Tecnológico de Informática
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, Spain
email: fmunyoz@iti.upv.es

ABSTRACT
Database replication is a way to increase system performance and fault-tolerance of a given system. The price to pay is the effort needed to guarantee data consistency, and this is not an easy task. In this paper, we introduce a description of two 1-Copy-Serializable (1CS) [1] eager update everywhere replication protocols. The preliminary results of their implementation in the MADIS middleware architecture [2] are also presented. The advantage of these replication protocols is that they do not need to re-implement features that are provided by the underlying database. The first one does not rely on strong group communication primitives [3]; distributed deadlock is avoided by a deadlock prevention schema based on transaction priorities (whose information is totally local at each node). The second one manages replica consistency by the total order message delivery featured by Group Communication Systems (GCSs) [3].

KEY WORDS
Database replication, middleware, group communication systems, ROWAA, eager, two-phase commit.

1 Introduction

O2PL [4] was one of the first concurrency control algorithms specially designed for replicated databases. O2PL showed several advantages when compared with other general concurrency control approaches (such as distributed 2PL, basic timestamp ordering, wound-wait, or distributed certification): (a) As many of them, it does not need to propagate readsets in order to detect concurrency conflicts. Read locks are only locally managed, using the support provided by the underlying DBMS. (b) It only needs constant interaction [5], delaying all remote write-lock requests until commit time, being thus an optimistic variation of the distributed 2PL approach. This ensures a faster transaction completion time than those protocols based on linear interaction. (c) Its use of locks, although optimistic, guarantees a lower abortion rate than that of the timestamp-based approaches [4].

The principles of O2PL have been used in many modern database replication protocols [6, 7, 8] based on total order broadcast [3], removing thus the need of using the 2PC protocol in order to terminate the transactions, and improving in this way the protocol outlined in [4].

We propose two new replication protocols directly based on the ideas discussed above, and implemented in a middleware called MADIS [2]. These protocols are formally presented as state transition systems [1]. They do not follow the replication policy established in [9]. A middleware-based implementation has to necessarily add some collection and management tasks that reduce the performance of the resulting system, at least when compared to one built into the DBMS core [6]. On the other hand, the resulting system will be easily portable to other DBMSs. In both protocols we have eliminated the need of lock management at the middleware layer. To this end, we rely on the local concurrency control, adding some triggers that will be raised each time a transaction is blocked due to a lock request. Additionally, deadlocks are also prevented in these protocols (this was one of the main problems in the original O2PL algorithm) either by using priorities, the first one, or by the total order delivery guarantees of the GCS [3].

Our first protocol needs only a uniform reliable broadcast, but requires two communication phases in order to commit a transaction, since all its operations are firstly executed at a given site. The first phase comprises: update propagation, priority check among current active transactions at the rest of sites, and applying the updates on the local DBMS. Once this process is finished at each remainder site, it sends a message saying it is ready to commit. The second phase starts when all sites are ready to commit. The site where the transaction was originated multicasts a commit message to the rest of sites (the same may be applied for an aborted transaction). Therefore, it is also able to manage unilateral aborts; i.e., those raised in a given replica due to some error in such transaction execution.

The second protocol replaces the first communication phase with one total order broadcast, thus it simplifies the replication protocol. Since if the site where the transaction was executed sees this message as the first one to be applied, it sends a commit message, otherwise, it multicasts an abort message. However, it is not able to manage unilateral aborts. Both protocols will be compared in Section 4, providing some interesting figures about in what conditions each one provides the best results.

The rest of the paper is organized as follows. Section 2 introduces the system model. Replication protocols are formally described in Section 3. Section 4 presents some preliminary experimental results on the MADIS architecture. Finally, conclusions end the paper.

2 System model

The distributed system considered in this paper is composed of N sites. Each site contains a copy of the database (fully replicated). Sites communicate with each other by message exchange using a GCS [3]. We assume a GCS providing reliable channels among nodes, featuring the next group communication primitives: basic and total order multicast. This GCS includes the membership service with the virtual synchrony property [3].

Since the objective of the system is a middleware architecture providing database replication, clients access database by means of SQL statements through a client application with no modifications, using a standard interface like JDBC. These applications access the data repository via transactions, through the middleware layer where replication is managed. A transaction defines a partially ordered set of read and write operations [10]. Two or more transactions may concurrently access the same data item and may provoke a conflict among transactions provided that at least one of the conflicting transactions issues a write operation. Clients access the system through their closest site to perform transactions. Each transaction identifier includes the information about the node where it was originally created ($node(t)$). It allows the different replication protocol instances to know if a given transaction is a local or a remote one. Client applications access the system through their closest site to perform transactions following a JDBC style. When the application wishes to commit, only all write operations are propagated to the rest of sites. We follow a *read-one-write-all* (ROWA) policy.

Each site includes a database management system (DBMS) storing a physical copy of the replicated database. We assume that the DBMS ensures ACID properties of local transactions; transactions are serializable as in [11]. The DBMS gives standard actions such as: beginning a transaction; submitting an operation; and, finishing a transaction (either commit or abort). We have added a set of functions which are not provided by DBMSs but may easily be programmed as database procedures or functions so as to know the object written by a given transaction and the set of conflicting transactions between a write set and current active transactions at a given site (i.e. $t' \in getConflicts(WS(t)) \iff (WS(t') \cup RS(t')) \cap WS(t) \neq \emptyset$). As a final remark, we also assume that after the successful completion of a submitted operation by a transaction, it can be committed at any time. In other words a transaction may be aborted by the DBMS only when it is performing a submitted operation ($submit(t, op)$).

3 Description of the protocols

We propose two different replication protocols: the first one is based in the Two Phase Commit (2PC) [10] atomic commitment protocol and the other one is based in group communication primitives [6]. Both protocols need a recovery subprotocol that is not described in this paper, but that has been already designed. These recovery subprotocols are based on the one described in [12] for BULLY, and on that described in [13] for TORPE.

The first replication protocol is an adaptation of the *Optimistic 2PL* protocol proposed by Carey et al. in [4], where all lock management has been avoided at the middleware level. This algorithm is called BULLY due to the fact that updates performed at remote nodes rollback all conflictive active (neither committed nor aborted) transactions whose priority is lower than its own. The second replication protocol, called Total Order Replication Protocol with Enhancements (TORPE), is an adaptation of the SER-D algorithm proposed by Kemme et al. in [6], where write operations are not deferred until the total order delivery of the message containing the updates. Both have been adapted to our MADIS middleware architecture [2]. We are very interested in the comparison of both since BULLY supports unilateral aborts whilst TORPE does not need to wait for the updates to be applied at the remainder sites, more precisely, to the slowest one. Both replication protocols behave in a similar way, except for update propagation to the rest of available sites. Hence, we will firstly explain the common part of both protocols and afterwards we will introduce the differences.

Each time a client issues a transaction (*local transaction*), all its operations (i.e. all reads and writes) are locally performed on a single node called the *transaction master site*. The remainder sites enter in the context of this transaction when the user wants to commit. All write operations are grouped and sent to the rest of available sites, at this moment is when the two protocols differ, since the former uses the basic service and the latter employs a total order. Updates are applied in the rest of sites in the context of another local transaction (*remote transaction*) on the given local database where the message is delivered.

In Figure 1 and 2, BULLY and TORPE are shown as state transition systems [1], introducing their respective steps and actions for a site i. Each action is subscripted by the node at which it is executed. Transactions created at node i (*local transactions* at i) follow a sequence, for both protocols, initiated by $create_i(t)$ and followed by multiple $begin\_operation_i(t, op)$, $end\_operation_i(t, op)$ pairs actions in a normal behavior. However, the $local\_abort_i(t)$ action is possible if the underlying database cannot guarantee serializability [11] or by an internal deadlock resolution. Each active transaction at node i ($status_i(t) = active$) is committable by the DB_i at any time. The $begin\_commit_i(t)$ action sends the write-set and update statements of a transaction t to every site and this is the place where both protocols differ.

```
States:
  $\forall i \in N \wedge t \in T$: $status_i(t) \in$ {idle, delivered, start, active, blocked, pre_commit, aborted, committed},
    initially $(node(t) = i \Rightarrow status_i(t) =$ start$) \wedge (node(t) \neq i \Rightarrow status_i(t) =$ idle$)$.
  $\forall i \in N, \forall t \in T$: $participants_i(t) \subseteq N$, initially $participants_i(t) = \emptyset$.
  $\forall i \in N$: $channel_i \subseteq \{m : m \in M\}$, initially $channel_i = \emptyset$.
  $\forall i \in N$: $\mathcal{V}_i \in \{\langle id, availableNodes\rangle : id \in \mathbb{Z} \wedge availableNodes \subseteq N\}$, initially $\mathcal{V}_i = \langle 0, N\rangle$.
Transitions:

$\mathbf{create}_i(t)$ // $node(t) = i$ //
pre ≡ $status_i(t) =$ start.
eff ≡ $DB_i.begin(t)$; $status_i(t) \leftarrow$ active.

$\mathbf{begin\_operation}_i(t, op)$ // $node(t) = i$ //
pre ≡ $status_i(t) =$ active.
eff ≡ $DB_i.submit(t, op)$; $status_i(t) \leftarrow$ blocked.

$\mathbf{end\_operation}_i(t, op)$
pre ≡ $status_i(t) =$ blocked $\wedge$ $DB_i.notify(t, op) =$ run.
eff ≡ $status_i(t) \leftarrow$ active;
    if $node(t) \neq i$ then
      $sendRUnicast(\langle ready, t, i\rangle)$ to $node(t)$;
      $status_i(t) \leftarrow$ pre_commit.

$\mathbf{begin\_commit}_i(t)$ // $node(t) = i$ //
pre ≡ $status_i(t) =$ active.
eff ≡ $status_i(t) \leftarrow$ pre_commit;
    $participants_i(t) \leftarrow \mathcal{V}_i.availableNodes \setminus \{i\}$;
    $sendRMulticast(\langle remote, t, DB_i.WS(t)\rangle,$
        $participants_i(t))$.

$\mathbf{end\_commit}_i(t)$ // $t \in T \wedge node(t) = i$ //
pre ≡ $status_i(t) =$ pre_commit $\wedge$ $participants_i(t) = \emptyset$.
eff ≡ $DB_i.commit(t)$; $status_i(t) \leftarrow$ committed;
    $sendRMulticast(\langle commit, t\rangle,$
        $\mathcal{V}_i.availableNodes \setminus \{i\})$.

$\mathbf{receive\_ready}_i(t, m)$ // $t \in T \wedge node(t) = i$ //
pre ≡ $status_i(t) =$ pre_commit $\wedge$ $participants_i(t) \neq \emptyset \wedge$
        $m = \langle ready, t, source\rangle \in channel_i$.
eff ≡ $receive_i(m)$; // Remove m from channel
      $participants_i(t) \leftarrow participants_i(t) \setminus \{source\}$.

$\mathbf{local\_abort}_i(t)$
pre ≡ $status_i(t) =$ blocked $\wedge$ $DB_i.notify(t, op) =$ abort.
eff ≡ $status_i(t) \leftarrow$ aborted; $DB_i.abort(t)$;
    if $node(t) \neq i$ then
      $sendRUnicast(\langle rem\_abort, t\rangle)$ to $node(t)$.

$\mathbf{discard}_i(t, m)$ // $t \in T$ //
pre ≡ $status_i(t) =$ aborted $\wedge$ $m \in channel_i$.
eff ≡ $receive_i(m)$.

$\mathbf{receive\_commit}_i(t, m)$ // $t \in T \wedge node(t) \neq i$ //
pre ≡ $status_i(t) =$ pre_commit $\wedge$
        $m = \langle commit, t\rangle \in channel_i$.
eff ≡ $receive_i(m)$; $DB_i.commit(t)$;
      $status_i(t) \leftarrow$ committed.

$\mathbf{receive\_remote}_i(t, m)$ // $t \in T \wedge node(t) \neq i$ //
pre ≡ $status_i(t) \neq$ idle $\wedge$
        $m = \langle remote, t, WS\rangle \in channel_i$.
eff ≡ $receive_i(m)$; $status_i(t) \leftarrow$ delivered;
      $conflictSet \leftarrow DB_i.getconflictSet(WS)$;
      if $\exists t' \in conflictSet : \neg(higher\_priority(t, t'))$ then
        $status_i(t) \leftarrow$ aborted;
        $sendRUnicast(\langle rem\_abort, t\rangle)$ to $node(t)$
      else // The delivered remote has the highest priority
        $\forall\ t' \in conflictSet$:
          $DB_i.abort(t')$;
          if $status_i(t') =$ pre_commit $\wedge$ $node(t') = i$ then
            $sendRMulticast(\langle abort, t'\rangle,$
                $\mathcal{V}_i.availableNodes \setminus \{i\})$;
          $status_i(t') \leftarrow$ aborted;
        $DB_i.begin(t)$; $DB_i.submit(t, WS)$;
        $status_i(t) \leftarrow$ blocked.

$\mathbf{receive\_abort}_i(t, m)$ // $t \in T \wedge node(t) \neq i$ //
pre ≡ $status_i(t) \notin$ {aborted, committed}
        $\wedge\ m = \langle abort, t\rangle \in channel_i$.
eff ≡ $receive_i(m)$; $DB_i.abort(t)$; $status_i(t) \leftarrow$ aborted.

$\mathbf{receive\_rem\_abort}_i(t, m)$ // $node(t) = i$ //
pre ≡ $status_i(t) \neq$ aborted $\wedge$
        $m = \langle rem\_abort, t\rangle \in channel_i$.
eff ≡ $receive_i(m)$; $DB_i.abort(t)$; $status_i(t) \leftarrow$ aborted;
      $sendRMulticast(\langle abort, t\rangle,$
          $\mathcal{V}_i.availableNodes \setminus \{i\})$.
ç
$\diamond$ function $higher\_priority(t, t') \equiv node(t) = j \neq i \wedge$
      $status_i(t) =$ delivered $\wedge (a \vee b \vee c)$
(a) $node(t') = i \wedge status_i(t') \in$ {active, blocked}
(b) $node(t') = i \wedge status_i(t') =$ pre_commit $\wedge$
      $t.priority > t'.priority$
(c) $node(t') = k \wedge k \neq j \wedge k \neq i \wedge status_i(t') =$
      blocked $\wedge$ $t.priority > t'.priority$
```

Figure 1. State transition system for the BULLY protocol.

3.1 BULLY replication protocol

Write operations are multicast to the rest of sites using the basic service. Once the remote transaction is finished it sends a message saying it is ready to commit the given transaction. When the reception of *ready* messages is finished, that is, all nodes have answered to the transaction master site, it multicasts a message saying that the transaction has been committed. BULLY relies for conflict detection on the mechanism implemented in the underlying DBMS which guarantees local serialization as stated in [11]. This is not enough to prevent distributed deadlock cycles formation [10]. We have avoided this problem using a deadlock prevention schema based on priorities, rather than the usage of total order broadcast primitives as in [6]. A global priority value for each transaction, based on the transaction state and a unique value, taking into account the transaction information (timestamp, objects read, written, etc.) along with its site identifier, is defined ($t.priority$).

The key action of the BULLY protocol is the $receive\_remote_i(m)$ action of Figure 1. Once the remote message is received at node i, the protocol action finds out in the local copy of the database the set of transactions conflicting with the received write set (WS). The remote updates, for that WS, will only be applied if there is not a conflictive transaction at node i having a higher priority than the received one. If there exists a conflictive transaction at i with higher priority, the remote message is ignored and sends a *remote abort* to the transaction master site.

Finally, if the remote transaction is the one with the highest priority among all at i then every conflictive transaction is aborted and the transaction updates are submitted for their execution to the underlying DBMS. The finalization of the remote transaction ($end\_operation_i(t, op)$), upon successful completion of $DB_i.submit(t, WS)$, is in charge of sending the ready message to the trans-

```
States:
  $\forall\, i \in N \wedge t \in T$: $status_i(t) \in$ {idle, start, active, blocked, pre_commit, committable, committed, aborted},
    initially $(node(t) = i \Rightarrow status_i(t) =$ start$) \wedge (node(t) \neq i \Rightarrow status_i(t) =$ idle$)$.
  $\forall\, i \in N$: $channel_i \subseteq \{m\colon m \in M\}$, initially $channel_i = \emptyset$.
  $\forall\, i \in N, \forall\, t \in T$: $committable_i(t) \subseteq T$, initially $committable_i(t) = \emptyset$.
  $\forall\, i \in N$: $\mathcal{V}_i \in \{\langle id, availableNodes\rangle : id \in \mathbb{Z} \wedge availableNodes \subseteq N\}$, initially $\mathcal{V}_i = \langle 0, N\rangle$.

Transitions:

create$_i(t)$ // $node(t) = i$ //
pre ≡ $status_i(t) =$ start.
eff ≡ $DB_i.begin(t)$;
    $status_i(t) \leftarrow$ active.

begin_operation$_i(t, op)$ // $node(t) = i$ //
pre ≡ $status_i(t) =$ active.
eff ≡ $DB_i.submit(t, op)$;
    $status_i(t) \leftarrow$ blocked.

end_operation$_i(t, op)$
pre ≡ $status_i(t) =$ blocked $\wedge\ DB_i.notify(t, op) =$ run.
eff ≡ if $node(t) = i$ then $status_i(t) \leftarrow$ active
    else $status_i(t) \leftarrow$ pre_commit.

begin_commit$_i(t)$ // $node(t) = i$ //
pre ≡ $status_i(t) =$ active.
eff ≡ $status_i(t) \leftarrow$ pre_commit;
    $sendTORMulticast(\langle remote, DB_i.WS(t), t\rangle,$
      $\mathcal{V}_i.availableNodes)$.

end_commit$_i(t, m)$ // $t \in T \wedge node(t) = i$ //
pre ≡ $status_i(t) =$ pre_commit $\wedge$
      $m = \langle remote, WS, t\rangle \in channel_i$.
eff ≡ $receive_i(m)$; // Remove m from channel
    $DB_i.commit(t)$;
    $status_i(t) \leftarrow$ committed.
    $sendRMulticast(\langle commit, t\rangle,$
      $\mathcal{V}_i.availableNodes \setminus \{i\})$.

local_abort$_i(t)$
pre ≡ $status_i(t) =$ blocked $\wedge\ DB_i.notify(t, op) =$ abort.
eff ≡ $status_i(t) \leftarrow$ aborted;
    $DB_i.abort(t)$.

discard$_i(t, m)$ // $t \in T$ //
pre ≡ $status_i(t) =$ aborted $\wedge\ m = \langle *, t, *\rangle \in channel_i$.
eff ≡ $receive_i(m)$. // Remove m from channel

receive_commit$_i(t, m)$ // $t \in T \wedge node(t) \neq i$ //
pre ≡ $m = \langle commit, t\rangle \in channel_i$.
eff ≡ $receive_i(m)$; // Remove m from channel
    $committable_i \leftarrow committable_i \cup \{t\}$.

receive_remote$_i(t, m)$ // $t \in T \wedge node(t) \neq i$ //
pre ≡ $status_i(t) \neq$ aborted $\wedge$
    $m = \langle remote, WS, t\rangle \in channel_i$.
eff ≡ $receive_i(m)$; // Remove m from channel
    $conflicts \leftarrow DB_i.getConflicts(WS)$;
    $\forall\ t' \in conflicts$:
      if $node(t') = i$ then
        $DB_i.abort(t')$;
        if $status_i(t') =$ pre_commit then
          $sendRMulticast(\langle abort, t'\rangle,$
            $\mathcal{V}_i.availableNodes \setminus \{i\})$;
        $status_i(t') \leftarrow$ aborted;
    $DB_i.begin(t)$;
    $DB_i.submit(t, WS)$; $status_i(t) \leftarrow$ blocked.

receive_abort$_i(t, m)$ // $t \in T \wedge node(t) \neq i$ //
pre ≡ $m = \langle abort, t\rangle \in channel_i$.
eff ≡ $receive_i(m)$; // Remove m from channel
    if $status_i(t) \neq$ aborted then
      $DB_i.abort(t)$; $status_i(t) \leftarrow$ aborted.

commit_remote$_i(t)$ // $node(t) \neq i$ //
pre ≡ $status_i(t) =$ pre_commit $\wedge\ t \in committable_i$.
eff ≡ $committable_i \leftarrow committable_i \setminus \{t\}$;
    $DB_i.commit(t)$; $status_i(t) \leftarrow$ committed.
```

Figure 2. State transition system for the TORPE protocol.

action master site. Once all ready messages are collected from all available sites the transaction master site commits (*end_commit*$_i(t)$) and multicasts a commit message to all available nodes. The reception of this message commits the transaction at the remainder sites (*receive_commit*$_i(t, m)$). In case of a remote update failure while being applied in the DBMS, the *remote abort* message is sent by the *local_abort*$_i(t)$ action to the master site. Once the updates have been finally applied the transaction waits for the *commit* message from its master site. One can note that the remote transaction is in the *pre_commit* state and it is committable from the DBMS point of view, since its status remains *active*. As a concluding remark, we will highlight that a delivered remote transaction has never a higher priority than other conflictive remote transactions at node i in the *pre_commit* state; this fact is needed to guarantee the transaction execution atomicity.

3.2 TORPE Replication Protocol

Write operations are multicast to the rest of sites using the total order service. If the message is delivered at the transaction master site, and it has not been aborted yet, it commits in the local database (*end_commit*$_i(t, m)$) and multicasts a commit message to the rest of sites using the basic service. The other case is when the message is delivered at the rest of sites (*receive_remote*$_i(m)$), this transaction aborts all local active conflicting transactions (those that are still in their acquisition phase and those that have sent their write operations but have not been yet delivered). Afterwards, the write operations are submitted to the database. On successful completion of the operation (*end_operation*$_i(t, op)$) it switches its state to *pre_commit* and waits for the commit message from its respective transaction master site.

Remote transactions are committed once their commit message has been received. However, these *commit* messages may come before the total order retrieval of the remote update; therefore, we need a data structure (*committable*$_i$) to store the commit of a given remote transaction that, probably, it has not been yet received.

Again, the TORPE replication protocol takes no responsibility for conflict detection as this task is managed by the underlying DBMS, which ensures serializability. Distributed deadlock is prevented by the total order of updates delivery, but, on the other hand, we have that remote updates can not be aborted by the database internals (unilateral aborts) and is limited by the total order delivery latency.

4 Experimental results on MADIS

We are currently ending the implementation of MADIS while implementing our replication protocols on the MADIS architecture [2]. The results presented here are preliminary ones and merely point out the comparison between these protocols in a middleware architecture.

These results have been performed in two different environments, since the first one has not enough workstations to execute the second test. We firstly use a cluster of 4 workstations with full duplex Gigabit Ethernet (Mandrake 10.0, Pentium III 800MHz, 768MB main memory, 40GB SCSI disk) and secondly a cluster of 8 workstations with full duplex Fast Ethernet (Fedora Core 1, Pentium IV 2.8GHz, 1GB main memory, 80GB IDE disk). In all our tests, we use PostgreSQL 7.4 as the underlying DBMS (*www.postgresql.org*) and it provides serialization isolation level [11]. However, according to [11] the PostgreSQL is only Snapshot Isolation (SI), if its suggested classification is taken into account. Besides this refinement, it also adds new isolation levels: SI, cursor stability, etc. This DBMS ensures SI. Therefore, in the context of database replication, our replication protocols provide generalized SI [14]. Spread 3.17.3 (*www.spread.org*) is in charge of the group communication for the TORPE protocol along with TCP sockets for the BULLY protocol (failure-free assumption). The database is composed by 25 tables with 100 records each one. The experimental results consist of executing non-conflicting transactions composed of a number of update operations varying the number of clients.

In the first experiment, transactions averaging 4 updates are executed by a range of clients supporting different workloads using both replication protocols. Figure 3 shows the results for the BULLY and the TORPE protocols respectively. Results obtained in these figures determine the performance of both protocols with the same load of transactions. As shown in the figures, we may conclude, as expected, that TORPE behaves better than BULLY, due to the fact that BULLY has to wait for the application of updates at all nodes and the reception of the respective ready messages. TORPE has only to wait for the total order delivery of the remote message and is not affected by the remote nodes overhead.

The second experiment is directly related to the scalability of the replication protocols [9]. We perform a proof varying the number of sites from 2 to 8. The number of clients are distributed throughout the nodes ranging from 1 to 16 and the load introduced into the system remains constant to 8 TPS. We performed 500 transactions averaging 4 update operations each per client. Results introduced in Figure 4 show that TORPE behaves fine for a few number of clients and nodes but its results are comparable to those obtained by BULLY with a higher number of clients and sites. This is due to the fact that the latency of total order delivery grows with the number of nodes, as it takes more time to agree on the delivery order. BULLY grows linearly with the number of nodes and TORPE grows much faster.

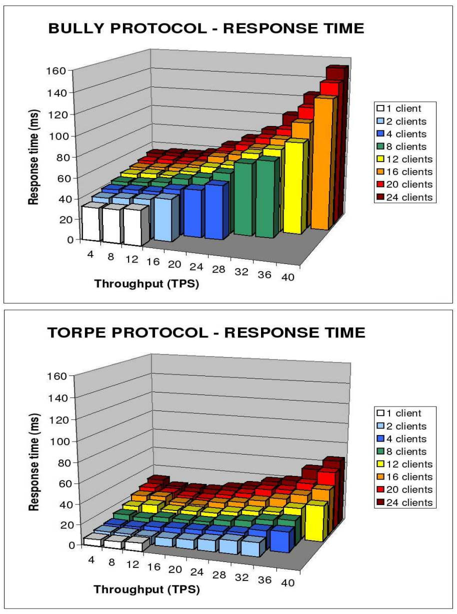

Figure 3. Performance Analysis: Response time with four operations per transaction in a system with four sites.

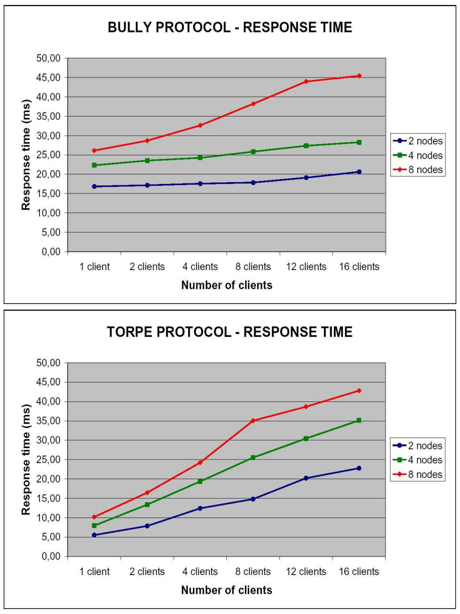

Figure 4. Scalability Analysis: Response time with four operations per transaction.

5 Conclusions

We have introduced two eager update everywhere replication protocols for the MADIS middleware architecture [2], that ensures 1CS, although we have not formally proven this assertion, provided that the underlying DBMS ensures serializability.

The first one is based on O2PL with a 2PC protocol (BULLY), whose replication strategy consists of setting transaction priorities, and the other one (TORPE) ensures database replication by means of total order delivery guarantees provided by a group communication system. The novelty of these replication protocols is that no extra database explicit operations must be re-implemented at the middleware layer (such as implementing a lock management or so). The main goal is to maintain data concurrency relying on the DBMS itself, and data replication is managed by the protocols described in this paper.

Experimental results presented here must be considered with caution, since we are in a preliminary stage of the system implementation (middleware architecture and replication protocols). We are currently performing several code optimizations in our architecture. Nevertheless, these results compare for the first time, up to our knowledge, the O2PL with GCS based protocols.

As it can be derived from the figures showing the experimental results, the TORPE protocol has a more accurate performance than the BULLY protocol. This is something that we expected since the 2PC must wait for all updates to be performed at the rest of nodes in order to commit a transaction. However, the TORPE replication protocol does not scale so well, although its performance is still comparable to BULLY. The preliminary results shown in this paper are for non-conflictive transactions. In the future, we plan to use ordinary TPC-W (`http://www.tpc.org`) benchmarks, since they combine read operations with update operations, and a non-negligible conflict rate.

Besides, the BULLY protocol may be enhanced if we assume that no unilateral aborts do occur. Under this assumption, we may send the $ready$ message before the operation submission to the underlying DBMS. Therefore, we do not have to wait for performing the update operation on the DBMS before sending the $ready$ message. Finally, as we have said before, we have not described a recovery protocol for these replication protocols. A recovery protocol for the BULLY may be based on the ideas introduced in [12]; respectively, a recovery protocol approach for the TORPE protocol would be based on [13].

Acknowledgments

This work has been supported by the Spanish Government under research grant TIC2003-09420-C02.

References

[1] A. Udaya Shankar, An introduction to assertional reasoning for concurrent systems, *ACM Comput. Surv.*, 25(3), 1993, 225–262.

[2] L. Irún-Briz, H. Decker, R. de Juan-Marín, F. Castro-Company, J.E. Armendáriz, and F.D. Muñoz-Escoí, Madis: A slim middleware for database replication, *Euro-Par*, 2005, Lecture Notes in Computer Science.

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

[4] M.J. Carey and M. Livny, Conflict detection tradeoffs for replicated data, *ACM Trans. Database Syst.*, 16(4), 1991, 703–746.

[5] F. Pedone, M. Wiesmann, A. Schiper, B. Kemme, and G. Alonso, Understanding replication in databases and distributed systems, *ICDCS*, 2000, 464–474.

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

[7] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, Using optimistic atomic broadcast in transaction processing systems, *IEEE Trans. Knowl. Data Eng.*, 15(4), 2003, 1018–1032.

[8] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso, Improving the scalability of fault-tolerant database clusters, *ICDCS*, 2002, 477–484.

[9] J. Gray, P. Helland, P.E. O'Neil, and D. Shasha, The dangers of replication and a solution, *SIGMOD Conference*, 1996, 173–182, ACM Press.

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

[11] H. Berenson, P.A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P.E. O'Neil, A critique of ANSI SQL isolation levels,*SIGMOD Conference*, 1995, 1–10.

[12] J.E. Armendáriz, J.R. González de Mendívil, and F.D. Muñoz-Escoí, A lock-based algorithm for concurrency control and recovery in a middleware replication software architecture, *HICSS*, 2005, 291.

[13] R. Jiménez-Peris, M. Patiño-Martínez, and G. Alonso, Non-intrusive, parallel recovery of replicated data, *SRDS*, 2002, 150–159.

[14] S. Elnikety, F. Pedone, and W. Zwaenopoel, Database replication using generalized snapshot isolation, *SRDS*, 2005, IEEE Computer Society.