Implementing Replication Protocols in the MADIS Architecture[*]

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

F.D. Muñoz-Escoí,
L. Irún-Briz
Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Camino de Vera s/n
46022 Valencia
{fmunyoz,lirun}@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 which it is not an easy task. In this paper, we introduce a description of two one-copy-serializable eager update everywhere replication protocol. The preliminary results of their implementation in the MADIS middleware architecture is 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; 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.

1. Introduction

O2PL [4] was one of the first concurrency control algorithms specially designed for replicated databases. It showed several advantages when compared with other general concurrency control approaches (such as distributed 2PL, basic timestamp ordering, wound-wait, or distributed certification):

- 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.
- It only needs constant interaction [10], 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.
- 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 [7, 8, 9] based on total order broadcast [1], 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 O2PL concurrency control discussed above, and implemented in a middleware called MADIS [6]. 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 [7]. On the other hand, the resulting system will be easily portable to other DBMS's.

In both protocols we have eliminated the need of lock management at the middleware layer. To this

[*]Supported by the Spanish Government under research grant TIC2003-09420-C02 and Generalitat Valenciana grant Grupos05/067.

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) using priorities: each time a transaction completes one of the phases that will be described later, its priority is increased; in case of conflicts, the transaction with highest priority is completed, and the other is rolled back.

Our first protocol needs only a uniform reliable broadcast, but requires two communication phases in order to commit a transaction, propagating the updates and requesting the locks in the first round of the first phase. It is also able to manage unilateral aborts; i.e., those raised in a given replica due to some error in such a transaction.

The second protocol replaces the first communication phase with one total order broadcast, simplifying thus the replication protocol. 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 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 group communication system [1]. We assume a group communication system providing reliable channels among nodes, featuring the next group communication primitives: basic and total order multicast. This group communication system includes the membership service with the virtual synchrony property [1].

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 [3]. 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. All operations over logical objects are firstly performed over the physical copy of that object on this site; afterwards, when the user wishes to commit, only all write operations are propagated to the rest of sites. We follow a *read-one-write-all* (ROWA) policy, hence we assume no failures.

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 ANSI serializable [2]. 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 conflicts 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 [3] atomic commitment protocol and the other one is based in group communication primitives [7].

The first replication protocol, called BULLY, is an adaptation of the *Optimistic 2PL* protocol proposed by Carey et al. in [4]. The second replication protocol, called TORPE, is an adaptation of the SER-D algorithm proposed by Kemme et al. in [7]. Both have been slightly modified and adapted to our middleware architecture. Thus, in the former we have not implemented any kind of locks at the middleware layer, and in the latter protocol we permit write operations on the local copy of objects being modified (i.e. we do not have deferred write operations). 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, a formal description of the signature and steps for a site i for the BULLY and TORPE algorithms are respectively introduced. 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 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.

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 ANSI serialization [2]. This assumption is not enough to prevent distributed deadlock cycles formation [3]. We have avoided this problem using a deadlock prevention schema based on priorities, rather than the usage of total order broadcast primitives as in [7]. 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 transaction master site. Once all ready messages are collected from all available sites the transaction master site commits ($end\_commit_i(t, m)$) and multicasts a commit message to all available nodes. The reception of this message commits the transaction at the remainder sites ($receive\_commit_i(t)$).

In case of a remote update failure while being

```
Signature:
  {$\forall i \in N, t \in T, m \in M, op \in OP$: $\mathbf{create}_i(t)$, $\mathbf{begin\_operation}_i(t, op)$, $\mathbf{end\_operation}_i(t, op)$, $\mathbf{begin\_commit}_i(t)$,
    $\mathbf{end\_commit}_i(t, m)$, $\mathbf{local\_abort}_i(t)$, $\mathbf{receive\_remote}_i(t, m)$, $\mathbf{receive\_ready}_i(t, m)$, $\mathbf{receive\_commit}_i(t, m)$,
    $\mathbf{receive\_abort}_i(t, m)$, $\mathbf{receive\_rem\_abort}_i(t, m)$, $\mathbf{discard}_i(t, m)$}.
States:
  $\forall i \in N \wedge t \in T$: $status_i(t) \in$ {idle, 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$)$.
  $\forall i \in N \wedge \forall t \in T$: $participants_i(t) \subseteq N$, 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:
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 ≡ $status_i(t) \leftarrow$ active;
    if $node(t) \neq i$ then
      $sendRUnicast(\langle ready, t, i\rangle)$ to $node(t)$;
      $status_i(t) \leftarrow$ pre_commit.

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

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

receive_ready$_i(t, m)$ // $t \in T \wedge node(t) = i$ //
pre ≡ $status_i(t) =$ pre_commit $\wedge$ $\|participants_i(t)\| > 1 \wedge$
      $m = \langle ready, t, source\rangle \in channel_i$.
eff ≡ $receive_i(m)$; // Remove m from channel
    $participants_i(t) \leftarrow participants_i(t) \setminus \{source\}$.

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 $node(t)$.

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 ≡ $receive_i(m)$; // Remove m from channel
    $DB_i.commit(t)$; $status_i(t) \leftarrow$ committed.

receive_remote$_i(t, m)$ // $t \in T \wedge node(t) \neq i$ //
pre ≡ $status_i(t) \neq$ aborted $\wedge$
      $m = \langle remote, t, WS\rangle \in channel_i$.
eff ≡ $receive_i(m)$; // Remove m from channel
    $conflictSet \leftarrow DB_i.getConflicts(WS)$;
    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 transaction
          // has the highest priority or no conflicts
      $\forall t' \in conflictSet$:
        $DB_i.abort(t')$;
        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.

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

receive_rem_abort$_i(t, m)$ // $t \in T \wedge node(t) = i$ //
pre ≡ $status_i(t) \neq$ aborted $\wedge$ $m = \langle rem\_abort, t\rangle \in channel_i$.
eff ≡ $receive_i(m)$; // Remove m from channel
    $sendRMulticast(\langle abort, t\rangle, \mathcal{V}_i.availableNodes \setminus \{i\})$;
    $DB_i.abort(t)$; $status_i(t) \leftarrow$ aborted.

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

$higher\_priority(t, t') \equiv node(t) = j \neq i \wedge$
      $status_i(t) =$ 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 replication protocol.

applied in the DBMS, the $local\_abort_i(t)$ action is responsible for sending the *remote abort* to the transaction 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 conclu-

Signature:

$\{\forall\, i \in N, t \in T, m \in M, op \in OP$: $\mathbf{create}_i(t)$, $\mathbf{begin\_operation}_i(t, op)$, $\mathbf{end\_operation}_i(t, op)$, $\mathbf{begin\_commit}_i(t)$, $\mathbf{end\_commit}_i(t, m)$, $\mathbf{local\_abort}_i(t)$, $\mathbf{receive\_remote}_i(t, m)$, $\mathbf{receive\_commit}_i(t, m)$, $\mathbf{receive\_abort}_i(t, m)$, $\mathbf{discard}_i(t, m)$, $\mathbf{commit\_remote}_i(t)\}$.

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$)$.
$\forall\, i \in N$: $channel_i \subseteq \{m : m \in M\}$, initially $channel_i = \emptyset$.
$\forall\, i \in N \wedge t \in T$: $committable_i(t) \subseteq T$, initially $committable_i = \emptyset$.
$\forall\, i \in N$: $\mathcal{V}_i \in \{\langle id, availableNodes\rangle : id \in \mathbb{Z} \wedge availableNodes \subseteq N\}$, 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 ≡ if $node(t) = i$ then $status_i(t) \leftarrow$ active
      else $status_i(t) \leftarrow$ pre_commit.

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

$\mathbf{end\_commit}_i(t, m)$ // $t \in T \wedge node(t) = i$ //
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.
      $sendRMulticast(\langle commit, t\rangle,$
        $\mathcal{V}_i.availableNodes \setminus \{i\})$.

$\mathbf{local\_abort}_i(t)$
pre ≡ $status_i(t) =$ blocked $\wedge$ $DB_i.notify(t, op) =$ abort.
eff ≡ $status_i(t) \leftarrow$ aborted;
      $DB_i.abort(t)$.

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

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

$\mathbf{receive\_remote}_i(t, m)$ // $t \in T \wedge node(t) \neq i$ //
pre ≡ $status_i(t) \neq$ aborted $\wedge$
          $m = \langle remote, WS, t\rangle \in channel_i$.
eff ≡ $receive_i(m)$; // Remove m from channel
      $conflictSet \leftarrow DB_i.getConflicts(WS)$;
      $\forall\, t' \in conflictSet$:
        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.

$\mathbf{receive\_abort}_i(t, m)$ // $t \in T \wedge node(t) \neq i$ //
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.

$\mathbf{commit\_remote}_i(t)$ // $node(t) \neq i$ //
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 Total Order Replication Protocol Enhanced (TORPE), where updates are propagated using the total order semantics provided by the group communication system primitives.

ding remark, we will highlight that a delivered remote transaction has never a higher priority than other conflictive remote transaction 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, 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.

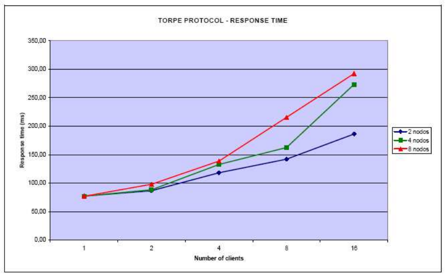
Remote transactions are committed once their commit message has been received. However, these $commit$ messages may come before the total order retrieval of the remote update; therefore, we need a data structure ($committable_i$) to store the commit of a given remote transaction that, probably, it has 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, in 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 and implementing our replication protocols on the MADIS architecture [6]. The results presented here are preliminary ones and merely points out the comparison between those protocols in a middleware architecture. The results have been measuren in 100 Mbps LAN composed by four computers **poner aquí los datos de las maquinas: procesador, sistema operativo y RAM**. We use PostgreSQL 7.4 as the underlying DBMS and Spread 3.17.3 as the group communication system for the TORPE protocol along with UDP multicast for the BULLY protocol (we are assuming no site failure).

The database is composed by 25 tables with 100 registers each one. The experimental results consist of executing a non-conflicting transaction composed of one update operation varying the number of clients. The first experiment consists of executing for a range of clients supporting different workloads 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. We may conclude, as expected, that TORPE behaves better than BULLY (even



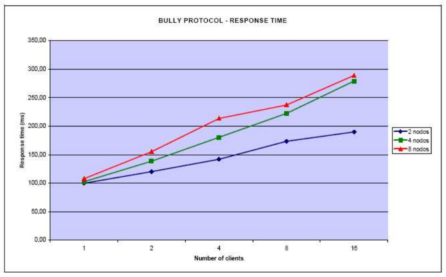
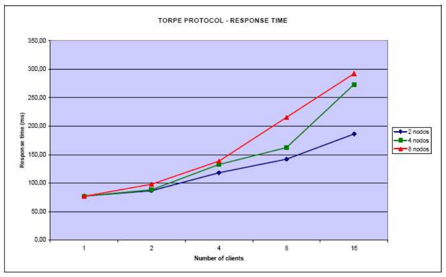


Figure 3: Performance Analysis: Response time with 1 operation per transaction in a MADIS architecture varying the number of transactions per second and the number of nodes.

though we have latencies of 40 ms with Spread), 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.

The second experiment is directly related to the scalability of the replication protocols [5]. We perform a proof varying the number of sites 2, 4 and 8 respectively. The number of clients range from 1 to 16 and the load remains constant to 6 TPS. We performed 200 transactions 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. BULLY grows linearly with the number of nodes and TORPE grows exponentially.

Txerra repasa esta parte que tú la controlas más. En la figura de escalabilidad cambia *nodos* por nodes. Entérate, o Paco, los datos de las máquinas como he puesto arriba



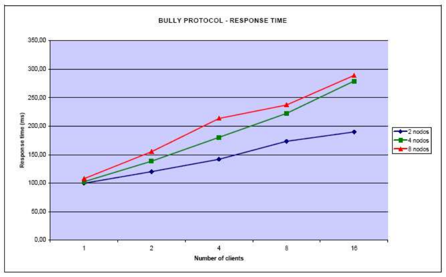


Figure 4: Scalability Analysis: Response time with 1 operation per transaction in a MADIS architecture varying the number of nodes.

5. Conclusions

We have introduced two eager update everywhere replication protocols for the MADIS middleware architecture [6], that ensures 1-copy-serializability, although we have not formally proof this assertion, provided that the underlying DBMS ensures serializability.

The first one is based on the two phase commit 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 table 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 in this paper must be carefully taken into account, since we are in a preliminary stage of the implementation of both: the middleware architecture and the replication protocols (in fact, we got latencies around 40 ms using Spread inside a LAN). We are currently performing several code optimizations of in our architecture. Nevertheless, these results compare for the first time, up to our knowledge, the O2PL with group communication based protocols.

As it can be derived from the figures showing the experimental results, the TORPE protocol has a more accurate performance (even with the Spread penalty) than the BULLY protocol. This is something that we expected since the two phase commit 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.

References

[1] A. Bartoli. Implementing a replicated service with group communication service. *Journal of Systems Architecture: The EUROMICRO Journal*, 50(8): 493-519, Elsevier North-Holland, August 2004.

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

[3] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, USA, 1987.

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

[5] J. Gray, P. Helland, P. O'Neil, and D. Shasha, The dangers of replication and a solution. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 173–182, Montreal, Canada, June 1996.

[6] 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. In *Proc. of the 2005 Int'l Euro-Par Conference*. *Accepted*.

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

[8] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. Strong replication in the globdata middleware. In *Proc. Workshop on Dependable Middleware-Based Sys. (Supp. Vol. of the 2002 DSN Conference), Washington D.C., USA*, pages G96–G104, June 2002.

[9] M. Wiesmann, F. Pedone and A. Schiper. A Systematic Classification of Replicated Database Protocols based on Atomic Broadcast. In *Proc. of the 3rd Europeean Research Seminar on Advances in Distributed Systems (ERSADS'99)*, Madeira Island, Portugal, 1999.

[10] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proc. of the 20th Int'l Conf. on Distributed Computing Systems (ICDCS)*, pages 464–474, Taipei, Taiwan, April 2000. IEEE-CS Press.