Improving the Scalability of Fault-Tolerant Database Clusters

R. Jiménez-Peris, M. Patiño-Martínez*
School of Computer Science
Technical University of Madrid (UPM)
Madrid, Spain
Ph./Fax: +34-91-3367452/12
{rjimenez, mpatino}@fi.upm.es

B. Kemme
School of Computer Science
McGill University
Montreal, Canada
Ph./Fax: +1-514-3988930/3883
kemme@cs.mcgill.ca

G. Alonso
Department of Computer Science
Swiss Federal Institute of Technology (ETHZ)
Zürich, Switzerland
Ph./ Fax: +41-1-6327306/1172
alonso@inf.ethz.ch

Abstract

Replication has become a central element in modern information systems playing a dual role: increase availability and enhance scalability. Unfortunately, most existing protocols increase availability at the cost of scalability. This paper presents architecture, implementation and performance of a middleware based replication tool that provides both availability and better scalability than existing systems. Main characteristics are the usage of specialized broadcast primitives and efficient data propagation.

1 Introduction

Many information systems are based on clusters where a number of servers simultaneously share the load and act as a backup to each other. This dual functionality is increasingly important as the system grows: additional sites can increase both the processing capacity and the availability of the system. For this to work, however, the data needs to be replicated across all servers (for availability) and the load partitioned to use all available resources (for performance). Most existing data replication protocols have failed to provide these requirements: they do not scale and create huge overheads [7].

What is needed is a replication protocol that can provide both performance and availability as well as an implementation that is not intrusive. In this paper we present such a protocol and its implementation atop existing systems. The applications we have in mind are clusters of data servers where requests can be divided into disjoint categories, a typical situation in e-commerce applications. The replication protocol we propose guarantees consistency at all times so that replicas can be used as backups for other replicas. It is also designed to minimize the processing overhead of replication, thereby providing a higher scalability than what is currently possible with existing systems. The main characteristics of our approach are as follows. Transactions are executed locally and only their changes are sent to the other sites reducing the execution overhead of replication. The usage of a special form of total order multicast (in which transaction execution overlaps with the time needed to determine the total order of messages) reduces transaction respone time by minimizing message latency. Furthermore, it allows for concurrent execution of transactions without sophisticated mechanisms. The system is implemented as a middleware layer requiring only little functionality from the underlying data sources. This shows that efficient replication is feasible in layered and component-based systems.

*This work has been partially funded by the Spanish Research Council (*CICYT*), contracts *TIC98-1032-C03-01* and *TIC2001-1586-C03-02*.

2 Motivation

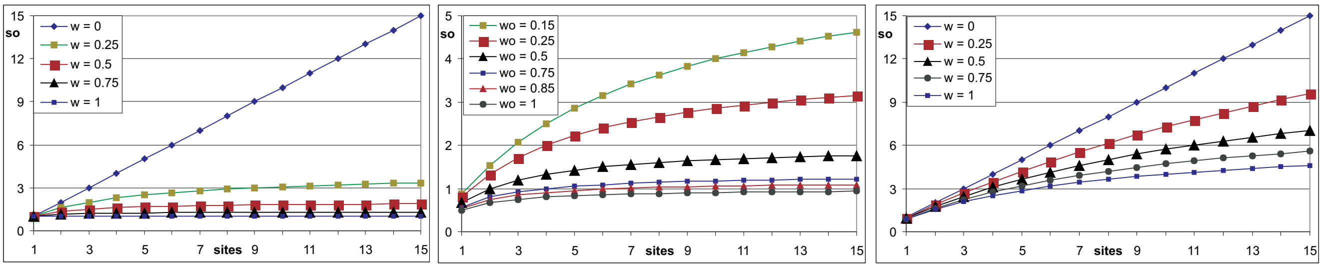
2.1 Limitations of Data Replication

A replicated database consists of a group of nodes (sites). Each site has a copy of the database. Clients interact with the system by submitting transactions to any site. Each site coordinates the execution of the transactions submitted to it. A transaction is called *local* at the site it is submitted to, and *remote* at the other sites. Nearly all commercial systems use the *read-one write-all available* (ROWAA) approach in which queries (read-only transactions) are executed locally, while update transactions are executed at all currently available sites. The write-all policy has important implications since each site has not only to process its local transactions but also the updates of remote transactions.

Assume we have n sites in the system, each being able to execute t transactions per second (tps). Each site executes x local transactions and a number of remote transactions. This number depends on the proportion of update transactions (w). That is, at each site $t = x + w \cdot (n-1) \cdot x$ (i.e., each site processes its local queries and updates transactions, x, plus the update transactions arriving at the other sites, $(n-1) \cdot w \cdot x$). The relative throughput of a site i, or scale-out (so_i), is the work it performs on local transactions, x, divided by its nominal capacity, t. From here, the scale-out factor for the entire system, so, can be obtained as the sum of the scale-out factors of each site:

$$so = \frac{n}{1 + w \cdot (n-1)}$$

The scale-out factor indicates how much of the nominal capacity of the system, n, remains after replication has been taken into consideration. It clearly varies with the percentage of updates in the load. If we perform only updates, $w = 1$, the scale out factor is 1, indicating that the overall capacity is the same as that of a single site. Similarly, with only queries, $w = 0$, the scale out factor is n, indicating the system has linear scalability. Fig. 1(a) shows the overall scalability for an increasing number of sites as a function of w. The interesting aspect is that even relatively small values



(A) Scaleout for different w values (B) Scaleout for different wo values (C) Scaleout for $wo = 0.15$ and different w

Figure 1: Scale-out for different values of w and wo

of w have a significant effect on the scalability. To further understand this behavior, we need to examine what happens to write operations. At the local site each write operation must be parsed and executed. In many existing replication protocols, this price is paid by all sites. Alternatively, the local site can, instead, send the physical updates to the other sites so that they only have to install the changes. With this idea, for any site, the cost of a remote transaction (denoted *rt*) is smaller than the cost of a local one (denoted *lt*). We capture this notion by introducing a *writing overhead* $wo = rt/lt$. The writing overhead is maximal, $wo = 1$, when the cost of executing the local part of a transaction is the same than the cost of executing its remote part. From here, the scale-out factor becomes:

$$so = \frac{n}{1 + wo \cdot w \cdot (n-1)}$$

Fig. 1(b) shows that for small values of the writing overhead, the scalability of the system can be improved even if the load consists of only update operations. This improvement is also significant when the load is not just updates but also includes read operations (Fig. 1(c)). The intuition behind this behavior is clear: the smaller wo, the lower the cost of remote transactions and the less work a site has to do on behalf of other sites. Since sites have more spare capacity, the overall processing capacity of the system increases accordingly and hence the increase in throughput as more sites are added. This means, in order to achieve scalability the workload must be query oriented (small w) and updates must be propagated and applied efficiently (small wo).

2.2 Related Work

The limitations of replication are well known. Gray et al. [7] provided an empirical evaluation of them and concluded that conventional replication protocols [4] are unfeasible in practice. In an effort to resolve these limitations, proposals have been made to develop data replication using reliable total order multicast primitives [17, 13, 12, 16, 10, 11, 2, 1].

In this paper we borrow and expand solutions proposed in these previous efforts. The starting point is a protocol that has the potential to reduce the replication overhead in a clustered system [15]. This protocol exploits existing optimization techniques [13] to conceal the time required to establish the total order behind the execution of the transactions. Unlike previous work [12], this paper focuses on replication that can be performed on a middleware layer atop existing databases. The architecture of our system is quite similar to [1] but the focus of the research is different. [1] analyzes how to provide availability even in wide area networks, our focus is to provide efficient, concurrent, and scalable execution in cluster configurations.

3 Replication Protocol

In what follows, we give a short overview of the protocol we use in our system. It was first proposed in [15]. The protocol assumes that sites communicate by exchanging messages and only fail by crashing. Each site contains a copy of the entire database and the correctness criterion is one-copy-serializability [4].

3.1 Execution Model

We follow the *read-one write-all available* approach of existing systems. Furthermore, we assume that queries are executed at their local site using snapshot isolation [3, 20, 14].

As explained above, load partitioning is an important aspect of cluster based systems. This partitioning can be best done when working at the middleware layer and it is a common strategy when working with web servers. The idea is that the designer of a site identifies parts of the expected load that can be processed independently (e.g., different categories of auctions) and allocates each part to a different site. Other sites have a copy of the data to act as a backup and to facilitate read only operations over that data. We capture this idea using the notion of *conflict class*.

The available data is initially partitioned into basic *conflict classes*. Basic conflict classes are disjoint and can be as small as a tuple or a selection over a table. These basic conflict classes are then grouped into *compound conflict classes*. Compound conflict classes do not need to be disjoint but they need to be distinct. The load is partitioned based on compound conflict classes. Each compound conflict class has a *master* or primary site. A transaction T can access any compound conflict class and we assume that the class it will access is known in advance (C_T). The same mechanisms used in complex web sites to forward a request to the appropriate site can be used to identify the compound conflict class accessed by a transaction.

We say that a transaction, T, is *local* to the master site of C_T and it is *remote* everywhere else. For example, assume there are two sites N and N' and two basic conflict classes C_x and C_y. N is the master of the compound conflict class $\{C_x\}$ and N' is the master of the compound conflict classes $\{C_x, C_y\}$ and $\{C_y\}$. A transaction updating C_x is local at N and remote at N', a transaction updating both C_x and C_y will be local at N' and remote at N. Queries over any of these basic conflict classes can be local to either N or N'.

With this, a transaction T is processed as follows. At its start, T is broadcast to all sites. Only T's master site executes T. After completing the execution, the master site broadcasts a commit message to all other sites and piggybacks to this message the result of all modifications performed by T (i.e., the *write set* of T). Upon receiving these modifications, a remote site proceeds to install the changes directly without having to execute the transaction.

For concurrency purposes, we use a simplified version of a lock table: each site has a queue CQ_x associated to each basic conflict class C_x. Upon delivery of a transaction T that accesses a compound conflict class C_T, each site (local or remote) adds T to the queues of the basic conflict classes contained in C_T.

3.2 Communication Primitives

Update transactions are propagated to all sites using group communication primitives [8, 5, 6]. We assume a virtual synchronous system [5], where all group members see membership changes at the same logical instant.

Two messages are broadcast per update transaction: a message containing the transaction itself and a commit message with the updates performed. Commit messages are just reliable broadcast, no ordering guarantees are needed. The broadcast used to send update transactions to all sites needs total order semantics. This order determines the serialization order of the transactions. We use an aggressive version, optimistic delivery total ordered broadcast [13], of the optimistic total-ordered broadcast presented in [18].

Optimistic delivery in total ordered broadcast takes advantage of the fact that in a local area network, messages are often spontaneously totally ordered. This optimistic delivery broadcast is defined by three primitives. *To-broadcast*(m) broadcasts the message m to all the sites in the system. *Opt-deliver*(m) delivers message m optimistically to the application (with no order guarantees). *To-deliver*(m) delivers m definitively to the application (in a total order). A sequence of opt-delivered messages to an application is a *tentative order*. A sequence of to-delivered messages to an application is the *definitive order* or total order. Messages can be opt-delivered in a different order at each site, but are to-delivered in the same order at all sites. The properties of optimistic broadcast ensure that every to-broadcast message is eventually opt-delivered and to-delivered by every site in the system. They also ensure that no site to-delivers a message before opt-delivering it.

By delivering a message in two steps (opt-delivery and to-delivery) we are able to overlap transaction processing with the time needed to determine the total order. In the traditional approaches (see Fig. 2 (a)), first the total order of a transaction is determined, then the transaction is executed. In our approach, a message is opt-delivered as soon as it is received from the network and before the definitive ordering is established. Transaction processing can start directly after the opt-delivery (Fig. 2 (b)).

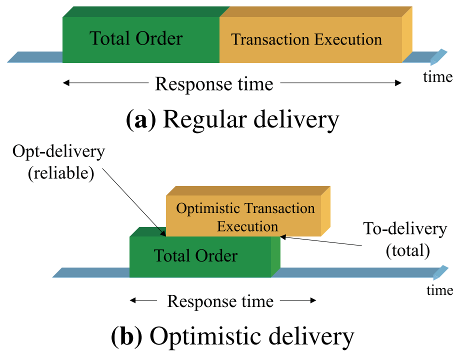


Figure 2: Optimistic execution

3.3 Replication Protocol

We describe the protocol (Fig. 3) according to different events that occur during the lifetime of a transaction T: T is opt-delivered, T is to-delivered, T completes execution, and T commits. One important issue is that a transaction can only commit when it has been executed and to-delivered. Each transaction has two state variables to ensure such a behavior. The *executed* variable is set when the execution of a transaction finishes. The *committable* variable indicates whether the transaction has been to-delivered.

When a transaction T is opt-delivered, it is queued in all the basic conflict classes it belongs to. This is done at all sites. At the master site of T, once T is the first transaction in all the queues, it will be submitted for execution.

At the time T completes its execution (this can only happen at its master), the to-delivery of T might have already taken place or not. If that is the case, T can be committed since it is the first transaction in all its queues and there cannot be a conflicting transaction ordered before T neither in the tentative order nor in the definitive order. The commit message (with the write set) is then broadcast to all sites. If the transaction has not been to-delivered, it is marked as executed. Waiting for the to-delivery before committing the transaction is necessary to avoid conflicting serialization orders at the different sites.

When the to-delivery message of T is processed at T's master site, T can be already executed. In this case, T is the first transaction in all its queues and there is no mismatch between the optimistic and the definitive orders. With this, T can commit, and the commit message is broadcast to all sites. If the transaction has not yet been executed or is not

Upon **Opt-delivery** of T_i
  For each conflict class $C_x \in C_{T_i}$
    Append T_i to the queue CQ_x
  EndFor
  Once T_i is the first in all queues
    $\wedge$ Local(T_i)
      Submit T_i for execution

Upon **TO-delivery** of T_i:
  Mark T_i as committable
  If T_i is executed **then**
    Commit T_i
    Remove T_i from all C_{T_i}
    Broadcast commit (WS_{T_i})
  Else *(still active or not local)*
    For each $C_x \in C_{T_i}$
      If First(CQ_x) = T_j $\wedge$ Local(T_j)
        $\wedge$ $\neg$ Committable (T_j) **then**
          Abort T_j
      EndIf
      Schedule T_i before the first not
        to-delivered transaction in CQ_x
      EndFor
    EndIf

Upon **complete execution** of T_i
  If T_i is marked as committable **then**
    Commit T_i
    Remove T_i from all C_{T_i}
    Broadcast commit(WS_{T_i})
  Else
    Mark T_i as executed
  EndIf

Upon receiving **commit**(WS_{T_i})
  If $\neg$ Local(T_i) **then**
    Delay until T_i becomes committable
      $\wedge$ T_i first in all the queues
    Apply the updates of WS_{T_i}
    Commit T_i
    Remove T_i from all C_{T_i}
  EndIf

Figure 3: Replication protocol

local, the protocol checks for mismatches between the tentative and the definitive total order that would lead to incorrect executions. If any conflicting transaction T' was opt-delivered before T but not yet to-delivered (mismatch between opt- and to-delivery order), it is incorrectly ordered before T in the queues they have in common. Hence, T and T' must be reordered such that T is scheduled before T'. This reordering step might lead to aborting a transaction. This would happen if T' was already executing or had already completed its execution and was waiting to be committed. However, note that the abort only occurs at the site where T' is local (on all other sites T' is remote; thus, reordering only requires to switch the transactions in the queues). Moreover, the probability for this situation to occur is quite low as it requires that messages get out of order *and* that the messages that are out of order correspond to transactions that conflict *and* one of the transactions is being or has been executed at that site.

A local transaction commits by submitting a commit to the database. At remote sites, the updates received in the commit message are applied after the transaction has been to-delivered to ensure that the updates are applied following the total (serialization) order. When the updates are applied and the commit has been submitted to the database, the transaction is removed from the queues. As shown in [15], this protocol guarantees 1-copy serializability.

4 Implementation

4.1 Architecture

Each site has a replica manager (Fig. 4) running an instance of the replica control protocol. The replica managers are implemented as a middleware layer, located between the clients submitting transactions and the database.

The *transaction manager* implements the replication protocol. It maintains the conflict class queues and controls the execution of the transactions. It coordinates with the other sites (exchanging commit messages) and interacts with clients (receiving transaction request and sending results) through the communication manager, and it submits transac-

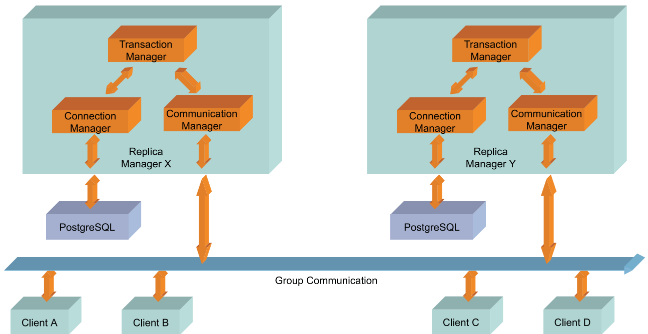


Figure 4: Main components

tions to the database through the connection manager.

The *communication manager* is the interface between the transaction manager and the group communication system (in our current system Ensemble [9]). Its task is to pipeline all messages (sending transactions and commits, opt- and to-delivering transactions, and delivering commits).

The transaction manager interacts with the database through the *connection manager*. In our current implementation, we use PostgreSQL [19], version 6.4.2, as underlying database system. The connection manager keeps a pool of processes available, each one of them with an open connection to the database. This acts as the connection pooling mechanisms found in modern middleware tools. Using this pool, transactions can be submitted and executed without having to pay the price of establishing a connection for each of them. At the same time, the processes can be used concurrently, thereby allowing the transaction manager to submit to the database several transactions at the same time.

4.2 Execution of Update Transactions

Upon receiving a transaction from a client, the transaction manager checks whether it is a query or an update transaction. If it is a query, it will be executed locally. Otherwise it broadcasts the update transaction to all sites using the optimistic delivery total order broadcast. When the communication manager opt-delivers an update transaction T, it enqueues the transaction in the queues of the basic classes that correspond to the compound class the transaction wants to access. If the site is the master for that compound class, then T is local to that site. When T is at the head of all of its queues at its master site, the transaction manager sends the transaction to the connection manager that in turn issues a begin transaction (BOT) and starts submitting operations for execution at the database. Once execution is completed, the transaction manager will not commit T until its definitive order is confirmed (the transaction is to-delivered). If the definitive (to-delivery) and tentative order (opt-delivery) agree, the transaction manager will commit T by issuing a commit to the connection manager. The connection manager, in turn, will return the result of T and its write set to the transaction manager. The transaction manager will return the results to the client, send a commit message with the updates to all the sites through the communication manager, and remove T from the queues. If the definitive (to-delivery) and tentative order (opt-delivery) do not agree, the transaction manager establishes whether the ordering problem affects transactions that conflict. If this is not the case, the ordering problem is ignored (the transactions do not conflict; transitive closures for transactions accessing compound classes that overlap are also captured by the to-delivery). If the ordering mismatch affects transactions that conflict, the serialization order obtained so far (following the opt-delivery order) is incorrect (it should have followed the to-delivery order) and T must be aborted. This is done by issuing an abort to the connection manager and reordering T accordingly in the queues. No communication with the rest of the sites is needed, since they will not apply the transaction updates until the local site completes.

If the transaction is remote, the transaction manager waits until it is at the head of all its queues, it is to-delivered, and its commit message has been received. The transaction manager then asks the connection manager to install the updates on the local copy. Once the transaction is completed, the transaction manager removes it from all the queues.

4.3 Execution of Queries

Queries are executed only at their local site. The idea is that queries are executed using snapshot isolation so that they do not interfere with updates. However, and unlike commercial products, PostgreSQL does not provide snapshot isolation. We solve the problem by giving queries a preferential treatment. Since queries are not sent to all sites, only the local site sees the query. Thus, as long as the local site makes sure that the query does not reverse the serialization order of updating transactions, it can execute the query at any time. This can be easily enforced by queuing the query after transactions that have been to-delivered and before transactions that have not yet been to-delivered. By doing this, the site can be sure that, no matter what happens to the update transactions, their serialization order will not be altered.

4.4 Optimistic Delivery

We have used Ensemble [9] as communication layer. However, Ensemble or other available group communication systems do not support a total order broadcast with optimistic delivery. We bypass this limitation by implementing the total order broadcast in two steps. First, the client sends the message using ip-multicast. Immediately after that, the message is sent again using the Ensemble reliable total order broadcast. The delivery of the first message represents the opt-delivery, the delivery of the second represents the to-delivery. As ip-multicast is unreliable, a total order broadcast can be received without having received the optimistic message. In that case, before delivering the total order message (to-delivery), an opt-delivery is automatically triggered.

4.5 Interaction with the Database

In terms of direct interfaces to the database engine, our implementation requires two services from the API of the database engine. The first is a service to obtain the write set of a transaction (the new physical values of the modified tuples) and the second is a service that installs changes instead of executing a transaction. These two services exist in most commercial databases although they are not always directly accessible. Nevertheless it is possible to use the tools that database engines provide to extent their functionality to make these services visible to the outside. Thus, for all intents and purposes, the protocol we propose can be used with most commercial database engines.

5 Experimental Results

5.1 Parameters of the Experiments

All the experiments have been run in a cluster of 15 SUN Ultra-5_10 (440MHz UltraSPARC-IIi CPU, 2 MB cache, 256 MB main memory, 9GB IDE disk) connected through a 100Mbits Fast Ethernet network.

The database used for the experiments consists of 10 tables, each with 10,000 tuples. Each table has five attributes: two integers, t-id (which also acts as the primary key) and attr1, one 50 character string (attr2), one float (attr3) and one date (attr4). The only index on the table is an index on the primary key. The tuple size is slightly over 100 bytes, which yields a database size of more than 10MB.

The load in the database is divided among update transactions and queries. Since there is an infinite range of possibilities in terms of how many read and write operations a transaction can have, we have simplified the load to make the results better understandable. We will consider update transactions that do not perform any read operation (worst case). The percentage variation between read and writes in the load is controlled by varying the relative number of update transactions vs. queries in the load.

The structure of the transactions used in the experiments is as follows. Update transactions have one or more update operations of the type:

```
update table-i set attr1="randomtext",
attr2=attr2+4 where t-id=random(1-10000)
```

Queries are structured as operations that scan a whole table and perform operations over all the data they read:

```
select avg(attr3), sum(attr3) from tab
```

Transactions have conflict rates between 10 and 20% (i.e., transactions have a 10-20% probability of conflicting with another transaction when they execute). For simplicity in the experiments, we have not used compound conflict classes but coarse granule basic classes. Each basic class encompasses $\frac{10{,}000}{16}$ data items within each table (i.e. $\frac{100{,}000}{16}$ data items in total) and there is a total of 16 conflict classes. Such a setting reflects a typical partition of the load in a cluster based web site. Transactions are submitted at a rate that varies from experiment to experiment and are evenly distributed among all the replicas. Table 1 summarizes the parameters of the experiments.

Parameters	Exp. 1	Exp. 2	Exp. 3	Exp. 4
Database size	10 tables of 10,000 tuples each			
Tuple size	approx. 100 bytes			
# of servers	1-15			
% of update txn	100%	0-100%	0-100%	100%
# of upd op. in txn	5	8	8	1
txn per second	10	max.	10-110	20-260
# write set size	504	804		104

Table 1: Experiment Parameters

5.2 Other Eager Replication Solutions

A first question that needs to be addressed is whether the protocol we propose really solves the limitations of conventional replication protocols (e.g., those described in [4]). Gray et al. [7] showed that these conventional protocols do not scale and, in particular, that increasing the number of replicas would increase the response time of update transactions and produce higher abort rates. To test the protocol, we have compared the scalability in terms of response time of our solution with that of a commercial product that implements replication based on standard distributed locking (Fig. 5.2, borrowed from [12]). For this experiment we used a fixed load of 5 update transaction per second and increased the number of sites in the system from 1 to 5 (the reason for using such a low load is that distributed locking could not cope with anything higher).

The results for distributed locking reflect the behavior predicted by Gray et al. The behavior clearly corresponds to a system that does not scale: for a fixed load, the response time increases as the number of sites increases. In this experiment the long response times are mainly due to the fact that distributed locking has a significant amount of messages all within the boundaries of the transaction. Our system in comparison was quite stable. For the range of sites explored the response time did not vary, showing that the message overhead is not significant and that the system

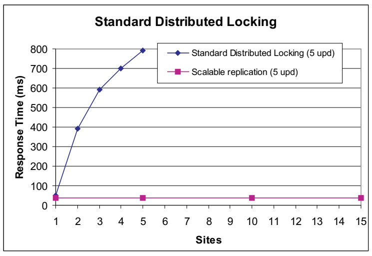


Figure 5: Comparison with Distributed Locking

is easily able to handle the small extra workload when more sites are added.

5.3 Throughput Scale-out

In the second test we examine how the processing capacity of the system (throughput) varies as more sites are added. We have run three sets of experiments to see how the system behaves when the load is read only, write only, and a mixture of both. The update transactions used perform 8 update operations each and have been designed to take about the same time as a query (to ease the comparison).

In the experiments, we first measured the maximum throughput delivered by a site. Then we used this 1-site throughput to determine how much the system scales for a given number of sites by measuring the maximum throughput for that number of sites. The number of sites varies from 1 to 15. The results are shown in Fig. 6.

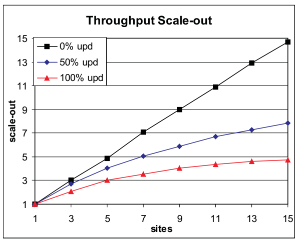


Figure 6: Scalability for different transaction loads

The linear scalability for read only loads (0% updates) is an obvious result of the protocol we propose. Since queries are executed only at one site and no information is propagated to the other sites, a system with N sites has a processing capacity that is N times larger than that of a single site. Note, however, that this scalability could have not been reached with a replication protocol based on quorums (where read operations do create redundant work). At the other extreme, for a write only load (100% updates) the scalability is limited but there is some scalability. The fact that, within the range tested, the scale-out is at least 30% of the number of sites is a significant gain over conventional protocols where the scalability would have been 0% [7]. In view of the analytical model, this proves that the protocol and implementation we propose indeed eliminates a fair amount of the redundant work done in the system (e.g., the writing overhead wo is around 0.15, a quite small value). Since the load we considered is only database load, the gain is relatively small. We are confident that once more complex transactions are involved (e.g., transactions that generate web pages with the results), this gain will be even more significant. The third experiment (50% updates) shows how the scalability improves as the proportion of queries increases (i.e., a decreasing w in the analytical model). This curve indicates how real systems (with a mixed load) will perform: the more queries, the more the scalability curve will resemble the upper one (0% updates); the more updates, the more the scalability curve will tend towards the lower one (100% updates).

5.4 Response Time Analysis

In this third experiment we try to find out the limits of the protocol by exploring when the response time becomes unacceptable. The idea is, given a system with a fixed number of sites, to increase the load while observing the response time until the response time is too high. As above, we consider loads of 0%, 50%, and 100% updates. We consider systems with 5, 10, and 15 sites. The transactions used are the same as in the previous experiment. The results are shown in Fig. 7(a-c).

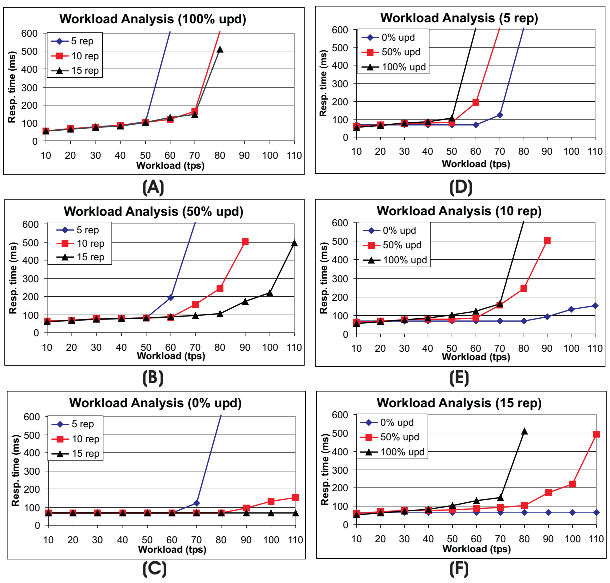


Figure 7: Response time for different transaction profiles and configurations

In all figures we observed a relatively flat evolution of the response time until the system is saturated and can no longer respond. As expected, the response time is essentially flat for read only loads but it does not increase significantly for write only loads. Also in all figures, the saturation point is reached at lower loads the higher the proportion of update transactions. With respect to each other, the saturation point moves to higher loads as the number of sites increases. This is also a good sign in terms of scalability as it indicates that for a given cluster, we can increase the throughput (experiment 2) by adding more sites without affecting the response time in any significant way. In addition, and interesting as well from the point of view of scalability, the growth in response time when the saturation point is reached is less explosive as the number of sites increases. This indicates that larger systems have a much more graceful degradation than smaller systems (a 50% update is a relatively high update rate; most systems have a larger proportion of queries and their response time curve will be somewhere between that for 50% updates and that of 0% updates). Finally, as the analytical model indicates, the response of any replication protocol is strongly determined by the proportion of updates in the load. For 100% update rates, the saturation point and the behavior of the system varies only slightly as the number of sites increase. This is due to the fact, seen in the previous experiment, that for high update rates, adding more sites does not significantly scale the system upwards.

This effect can be better observed in Figures 7(d-f) where the same results are grouped by the update rate rather than by system size. One interesting observation is the fact that for loads below the saturation point, the response time is exactly the same independently of the number of sites used (a similar result as that shown in experiment 1 but for much higher loads). The change in slope for the evolution of the response time for different update rates can be explained as it was done in the analytical model. At very low transaction rates, there is plenty of time to process write sets and transactions. As the transaction rates grow, the spare time diminishes and thus, the time devoted to process write sets starts to affect the transaction response time. For very high update rates, the amount of redundant work increases to the point that, as the load increases, there is less spare capacity in the system and, therefore, the response time grows as transactions have to wait longer to be executed.

5.5 Communication Overhead

When using group communication primitives, the system built can only scale as much as the underlying communication tool. In particular, our implementation requires two messages per transaction and it could be questionable whether the optimistic protocol we use is feasible in practice. In order to test this aspect of the system, we have performed a test with as many small update transactions as possible and observed how the system behaves. The transactions used contain a single update. The response time was measured for increasing loads and different configurations until the system was saturated. Fig. 8 shows a flat response time up to quite high transaction rates (200 transactions per second). This indicates that the communication does not become a bottleneck up to that point where the system saturates. At that stage, it does not matter what happens to the communication layer since the system is incapable of dealing with the load anyway. Thus, for the purposes of cluster based systems, the use of group communication primitives does not seem to be the limiting factor.

A last point to note regarding this experiment is the difference in scalability for short transactions (Fig. 8) and me-

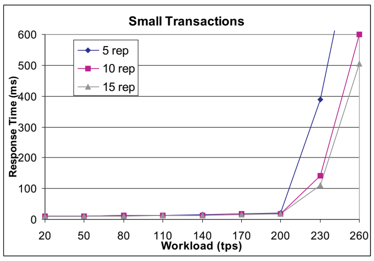


Figure 8: Small transactions

dium transactions (Fig. 7(d)). The reason is that for short transactions the constant overhead associated with processing a transaction remotely is quite large in relative terms. Thus, there is no that much redundant work to reduce. The longer the transaction the bigger the effect of reducing the writing overhead, wo, and the higher the scalability.

5.6 Aborts

A last aspect of the protocol we propose is the rate of aborted transactions it generates. Optimistic delivery of messages allows reducing the response time of transactions but some transactions might be aborted when there is a mismatch between the optimistic and definitive order for conflicting transactions. Unlike what has been observed in other replication protocols using group communication [10], in our experiments we observed a very low abort rate. We even conducted a set of experiments (varying the conflict rate and introducing hot spots, experiments not shown for reasons of space) to artificially increase the abort rate and it never went beyond 0.2%. This is due to the nature of the protocol, which in order to abort a transaction requires that the messages get out of order and that those transactions conflict and one of the transactions is local and is executing (or has been already executed). An interesting property of the algorithm is that the more sites in the system, the lower the probability of a transaction being local at a site, and thus, the lower the probability of an abort caused by messages arriving out of order. On the other hand increasing the number of sites increases the probability of messages getting out of order. These two opposed forces compensate each other when increasing the number of sites, and the experiments show that this compensation keeps the abort rate very low up to 15 sites.

6 Conclusions

In this paper we propose a middleware layer based on group communication primitives that exhibits a good scalability and circumvents the known limitations of existing protocols. One of the key features of this middleware layer is that it only requires two services from the database API that are implemented in most commercial databases, instead of modifying the whole database. The protocol allows designers to strike a reasonable balance between availability and scalability as it permits to add more sites to the system and yet improve both availability and scalability, without compromising consistency. The performance results we have obtained so far indicate that the protocol is a viable solution in many application scenarios in spite of running it as an additional middleware layer.

References

[1] Y. Amir, C. Danilov, M. Miskin-Amir, J. Stanton, and C. Tutu. Practical Wide Area Database Replication. Technical Report CNDS-2002-1, Johns Hopkins University, 2002.

[2] Y. Amir and C. Tutu. From Total Order to Database Replication. In *Proc. of Int. Conf. on Distr. Comp. Systems (ICDCS)*, July 2002.

[3] 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 SIGMOD*, pages 1–10, 1995.

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

[5] K. Birman. *Building Secure and Reliable Network Applications*. Prentice Hall, NJ, 1996.

[6] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computer Surveys*, 33(4):1–43, Dec. 2001.

[7] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *ACM SIGMOD'96*.

[8] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*, pages 97–145. Addison Wesley, 1993.

[9] M. Hayden. The Ensemble System. Technical Report TR-98-1662, CS Dept. Cornell University, Jan. 1998.

[10] J. Holliday, D. Agrawal, and A. E. Abbadi. The Performance of Database Replication with Group Communication. In *IEEE FTCS*, 1999.

[11] J. Holliday, D. Agrawal, and A. E. Abbadi. Using Multicast Communication to Reduce Deadlock in Replicated Databases. In *IEEE SRDS*, 2000.

[12] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, A new way to implement Database Replication. In *Proc. of Inf. Conf. on Very Large Databases, VLDB'00*, 2000.

[13] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *Proc. of IEEE ICDCS*, pages 424–431, 1999.

[14] Oracle. *Oracle 8 (tm) Server Replication*. 1997.

[15] M. Patiño Martínez, R. Jiménez Peris, B. Kemme, and G. Alonso. Scalable Replication in Database Clusters. In *Proc. of DISC'00, LNCS 1914*, pages 315–329, 2000.

[16] F. Pedone and S. Frolund. Pronto: A Fast Failover Protocol for Off-the-shelf Commercial Databases. In *Proc. of IEEE SRDS*, 2000.

[17] F. Pedone, R. Guerraoui, and A. Schiper. Transaction Reordering in Replicated Databases. In *IEEE SRDS'97*.

[18] F. Pedone and A. Schiper. Optimistic Atomic Broadcast. In *Proc. of DISC*, 1998.

[19] PostgreSQL. *v6.4.2*. http://www.postgresql.com, Jan. 1998.

[20] R. Schenkel and G. Weikum. Integrating Snapshot Isolation into Transactional Federations. In *5th Int. Conf. on Cooperative Information Systems*, Sept. 2000.