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

TECHNICAL REPORT

Abstract

The increasingly pervasive use of clusters makes replication a central element of modern information systems. Replication, however, must nowadays play a dual functionality: it must increase both the availability and the processing capacity of the application. Most existing data replication protocols cannot do this as they improve availability at the cost of scalability. In this paper we present a protocol that achieves this dual goal. The contribution is to demonstrate that data replication does not need to severely affect the overall scalability and that it can be efficiently implemented in a middleware layer. A key feature of this layer is that it only requires from the databases underneath, two commonly implemented services.

Keywords: replication, transactions, group communication, database clusters, fault-tolerant distributed systems.

1 Introduction

The increasingly pervasive use of clusters as the preferred platform for large information systems makes replication a central element of modern system architectures. Clusters offer the possibility to use off-the-shelf components to build more powerful information systems by simply adding more sites. This *scale out* approach (adding more elements) contrasts with the more traditional *scale up* approach (using more powerful elements) in that inexpensive elements can be used to increase the system capacity. From the

*Extended version of the paper published at 22nd IEEE Int. Conf. on Distributed Computing Systems 2002 (ICDCS'02).

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

hardware point of view, a system can only scale up to a certain limit and the cost is exponential with the scale up factor. For a cluster system, in principle there are no limits to how many sites can be added. Unfortunately, things are not that straightforward. In a scale up approach, the software architecture does not play a big role: the better the hardware, the faster the application. In a scale out approach, if one is not careful with the software design, all the advantages can be quickly lost. Replication, for instance, is one of the software aspects that require particular attention since an improper design will turn a large cluster into a very reliable but very expensive version of a single machine.

This problem is particularly acute in information systems like those running web sites for electronic commerce, on-line auctions, or large data warehouses. Most of these 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 thereby increase both the processing capacity and the availability of the system. For this to work, data needs to be efficiently replicated across all servers (for availability) and the load partitioned so that each server can take care of part of it (for performance). The latter requirement holds in many systems. The limiting factor today is that existing data replication protocols are entirely inadequate for cluster environments.

Conventional, text book data replication protocols [BHG87] have proven to be impractical for all intents and purposes: they do not scale and create huge overheads [GHOS96]. More modern approaches, like those that combine total order multicast with transactional properties, are for the most part theoretical constructs never implemented in a real system. The ones implemented, like Postgres-R [KA00a], require to modify the underlying database, something that is not always feasible. Ideally, what is needed is both a protocol that can provide replication for both performance *and* availability as well as an implementation that is not intrusive.

In this paper we present such a protocol and describe how it can be implemented 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 allowing a degree of scalability that is much better than what is currently possible with existing commercial systems. The contribution of this work is to demonstrate that data replication can be implemented without severely limiting the scalability of the cluster and that this does not require to modify existing tools but can efficiently be done as an additional middleware layer. Performance and availability gains aside, this is one of the most attractive features of the system we propose.

The paper is structured as follows. In Section 2, we formally analyze the concept of replication and show

how naive designs can reduce the scalability to essentially none. Section 3 presents the replication protocol. Section 4 discusses the implementation details of the middleware layer. The performance and scalability results are presented in Section 5 and Section 6 concludes the paper.

2 Motivation

This section presents an analytical model to illustrate the limitations of replication protocols. We concentrate on protocols that follow a read-one/write-all approach and maintain consistency at all times (i.e., eager protocols [GR93]).

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 one site. Each site is responsible for coordinating 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. Read-only transactions are executed locally, while update transactions are executed at all 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 processes 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 transactions, x, plus whatever operations arriving at other sites, $x \cdot (n-1)$, happen to be writes, $w \cdot (n-1) \cdot x$). Solving for x we obtain:

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

The relative throughput of a site i, or scale-out (so_i), is the actual work it performs, x, divided by its nominal capacity, t, that is:

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

From here, the scale-out factor for the entire system, so, can be obtained as the sum of the scale-out factor of each site:

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

This expression represents how much of the nominal capacity of the system, n, remains after replication has been taken into account. For instance, in a replicated system performing only updates, $w = 1$, the scale

out factor is 1, indicating that the overall capacity is the same as that of a single site (i.e., the system does not scale at all). Similarly, in a system performing only queries, $w = 0$, the scale out factor is n, indicating the system has linear scalability. These results are intuitive and correspond to what can be empirically observed. In a system with only queries, each site acts independently so the more sites we add, the more capacity there is. In a system with only updates, all sites are doing exactly the same work. Therefore, independently of how many sites there are, the overall capacity is the same as that of a single site. Thus, in order to have some scalability, w must be smaller than one. 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 of w have a significant effect on the scalability. The conclusion from here is that for replication to scale, we need a load with a sufficiently large proportion of read operations and somehow reduce the amount of redundant work done in the system.

To reduce the amount of redundant work, we need to examine what happens to write operations. At the local site each write operation must be parsed and executed. In naive replication protocols, this price is paid by all sites but this does not need to be case. The local site could, instead, send the physical updates to the other sites so that the other sites 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. In those cases all sites pay the same price for an update operation and, essentially, all sites are doing the same work. When consider the writing overhead, the scale-out factor becomes:

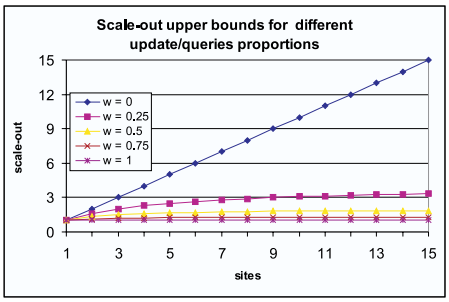
$$so = \frac{n}{1 + wo \cdot w \cdot (n-1)}$$

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 scalability as more sites are added.

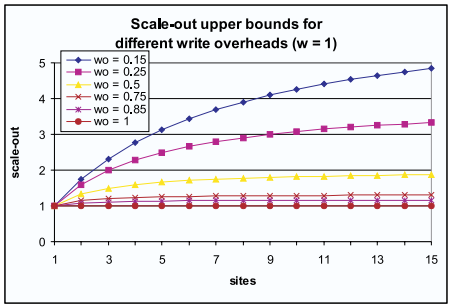
From this analysis, we conclude that, to obtain scalability, two aspects are important. First, there needs to be a sufficient number of read operations (thereby reducing the value of w). Second, the cost of executing a remote update transaction on behalf of other site has to be less than the cost of executing a local transaction (thereby reducing wo). This is particularly relevant for cluster based information systems that are accessed, e.g., through a web server. Nowadays transactions are no longer the short transactions typical of financial applications. Transactions often involve, for instance, invoking programs through triggers in the database

and the dynamic generation of web pages (i.e., lt is quite large). In those cases it is crucial for this work to be done only once and not repeated at every site in the system. If we manage to do this while still preserving consistency, then $rt \ll lt$ and the writing overhead will be small enough to allow the cluster to scale.

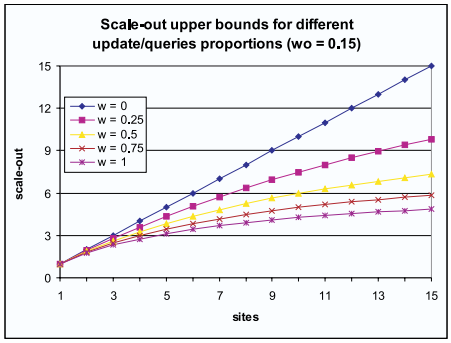
It could be argued that quorums are another way to reduce the amount of redundant work in the system. Quorums are quite misleading in terms of performance gains [JPAK01]: they introduce redundant work not only for write operations but also for read operations. Unlike write operations, read operations in databases tend to involve a lot of data (e.g, a select statement over an attribute with no index). Performing the read on several copies will result in a significant overhead because the I/O penalty will have to be paid by all sites accessed.



(a) Scale-out for different w values



(b) Scale-out for different wo values



(c) Scale-out for $wo = 15$ and different w

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

2.2 Related Work

Previous work on reliable middleware has mainly concentrated on replicated ORBs. Examples of these approaches are [Maf95, MSEL99, NMMS99, FGS99]. In contrast, this paper focuses on a novel middleware layer for database replication.

The limitations of replication pointed out in the formal model of the previous section are well known

among database designers. Gray et al. [GHOS96] provided an empirical evaluation of them and concluded that conventional replication protocols [BHG87] are unfeasible in practice. In an effort to resolve these limitations, several attempts have been made to develop data replication using group communication primitives. The resulting protocols can be classified in different groups. First, there are a number of protocols that demonstrate the potential of the idea but focus more on the formal aspects than on the practical issues [PGS97, KA00b, PF00]. A second group of protocols focuses on circumventing a single particular problem of replication (e.g., deadlocks) [HAA99, HAA00]. The conclusions drawn from these protocols are based on simulations and largely corroborate the results of the analytical model presented above. The third group of protocols exploits several optimizations to reduce the communication overhead [KPAS99]. Finally, a prototype implementation exists that focuses on how to combine group communication with database engines [KA00a]. This implementation requires modifying the database engine and, although it has demonstrated good performance, it is not clear how it can be used in a more general setting (i.e. without modifying the database). How complex this can be has recently been shown by a study on parallel database systems using off-the-shelf software and hardware [BGRS00]. The results of this study prove the limitations of existing cluster replication technology and the difficulties in reaching a reasonable scalability.

In this paper we borrow and expand ideas from these previous efforts. The starting point is a protocol recently proposed that has the potential to reduce the replication overhead in a clustered system [PJKA00]. This protocol exploits existing optimization techniques [KPAS99] to conceal the time required to establish the total order behind the execution of the transactions. Unlike previous work, this paper focuses on replication that can be performed on a middleware layer atop existing databases. In here, we are interested in an implementation at the middleware level and not within the database system as was done in previous work. Working on the middleware layer allows us to connect to any type of database system and without requiring any changes to the databases provided they export two commonly implemented services to get and apply the updates of a transaction. However, previous approaches [KA00a, KA00b] are useless for this purpose as they require to modify the whole database. The main goal is to offer an alternative that can be used in all those cases where it is not possible to modify the database or where heterogeneous databases are involved.

3 Replication Protocol

The system we present in this paper is based on an existing protocol [PJKA00]. For reasons of space, in this section we only briefly mention the main characteristics of the protocol and refer the interested reader to reference [PJKA00] for the correctness proofs and possible extensions to the basic protocol.

In what follows, we will assume 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 [BHG87].

3.1 Execution Model

Transactions are executed following a *read-one/write-all available* approach, where read operations are executed locally at one site and write operations are applied at all sites. Queries (read only transactions) are executed at their local site using snapshot isolation [BBG+95]. Snapshot isolation is based on the idea of obtaining a committed projection of the database at the time the query starts. The query will read data from this snapshot and will not see data committed after the query started. Snapshot isolation has the significant advantage of separating queries and updates as read only transactions do not interfere at all with updating transactions. This makes it very attractive in situations where conflicts can be a problem such as heterogeneous databases [SW00] or replication [KA00b]. Snapshot isolation is widely used in commercial database systems (e.g., it is the default isolation level in ORACLE [Ora97]).

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 class $\{C_x, C_y\}$. 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 is processed as follows. At the start of T, T is broadcast to all sites. However, it will only be executed at its master site. 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 the transaction (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, 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. This is a simplified version of a lock table [GR93] (using conflict classes instead of individual data items) and has been previously used in [KPAS99].

3.2 Communication Primitives

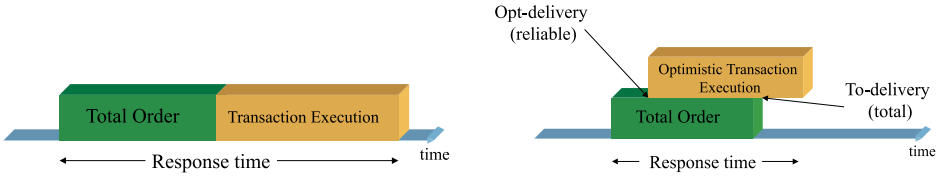
Update transactions are propagated to all sites using group communication primitives [HT93, Bir96, CKV01]. The properties of interest in this paper are *total order*, which ensures that messages are delivered in the same order at all the sites, and *reliability*, which ensures that a message is delivered at all available sites or to none in case the sender crashes. We assume a virtual synchronous system [Bir96], 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 [KPAS99], of the optimistic total-ordered broadcast presented in [PS98]. Optimistic delivery has been applied successfully to total order multicast in the context of replication [PJKA00] and to uniform multicast in the context of non-blocking atomic commitment [JPAA01].

Optimistic delivery in total ordered multicast 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. With this, the execution of

a transaction overlaps with the calculation of the total order (Fig. 2 (b)), and response times are only then affected by the delay of the total order multicast if establishing the total order takes longer than executing the transaction. If the initial order is the same as the definitive order, the transactions can simply be committed. If the definitive order is different, additional actions have to be taken to guarantee consistency.



(a) Regular delivery **(b)** Optimistic delivery

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 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. In the networks used for clusters the probability or messages arriving out of order is small. If the conflict rate is within a reasonable range, the overall probability of having to abort a transaction because of reordering is very low (indeed, in the experiments below, the abort rate observed was negligible even for high loads).

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 or the commit has been submitted to the database, the transaction is removed from the queues. As shown in [PJKA00], this protocol guarantees 1-copy serializability.

3.4 Characteristics of the Protocol

Among all existing replication protocols based on group communication, the protocol outlined above has several advantages. These advantages can be summarized in terms of availability, reduction of redundant work, and the optimistic approach followed.

In terms of availability, the protocol ensures that all sites have the latest version of all data items. Although each compound conflict class has a primary site, the other sites in the system are acting as a *hot, 2-safe* backup [GR93]. As a result, any site in the system can immediately take over the work of other site in case of failures. This can be done by dynamically reassigning the compound conflict classes for which the failed site was the primary to some of the surviving sites. In addition, since transactions are broadcast to all sites, clients do not need to be reconnected and a transaction being executed at a failed site is also present at the site taking over and does not need to be resubmitted. The new master site can start executing those transactions as soon as it takes over. Consistency is guaranteed thanks to the virtual synchronous nature of the broadcast primitive used. Thus, if a commit message has not been received before the view change, the transaction should be re-executed at its new master.

In terms of redundant work, the big advantage of the protocol is that remote sites only install changes

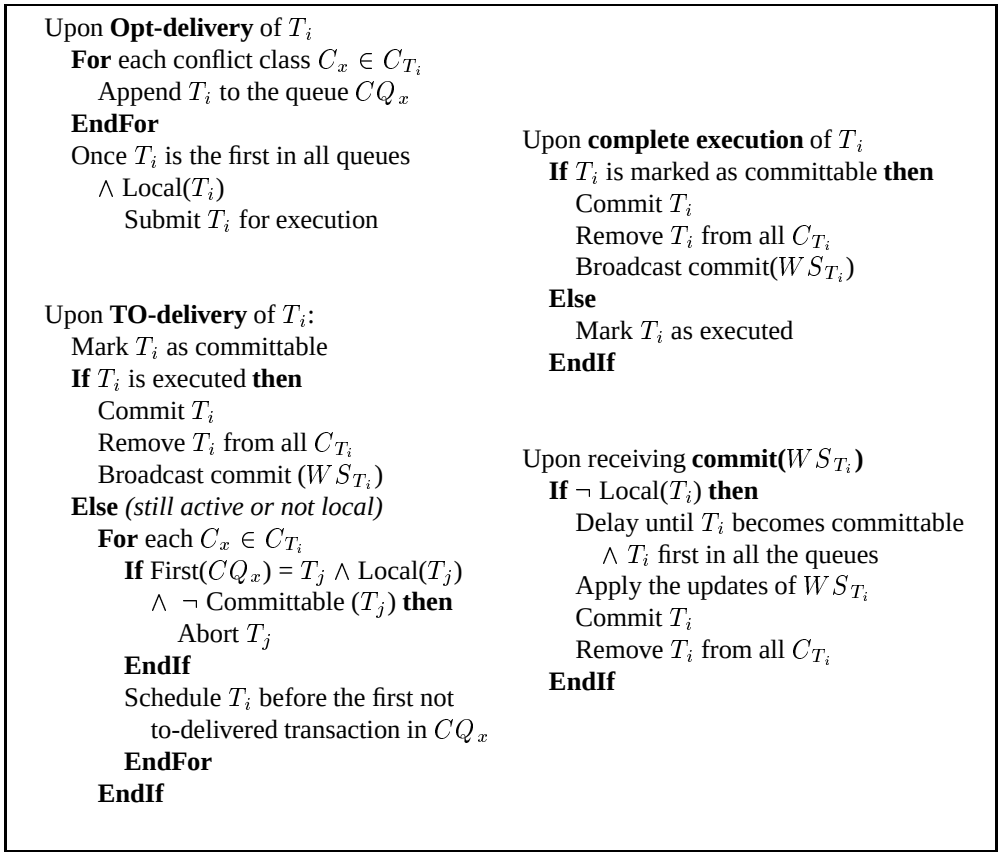


Figure 3: Replication protocol

instead of having to execute the whole transaction. As the experimental results will show, this greatly increases the scalability of the system. In addition, since queries can be executed at any site, the spare capacity of all sites is pooled together and also helps to increase the ability of the system to process more transactions if the load contains queries.

Finally, the protocol is based on the optimistic delivery of messages so that transaction execution does not need to wait until a total order is established. Although this might lead to having to abort transactions that were incorrectly executed, our experimental results show that the abort rate is very low. Thus, for most transactions, the optimistic delivery reduces the response time as the transaction can start executing right away and, in the meanwhile, the total order is established.

4 Implementation

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.

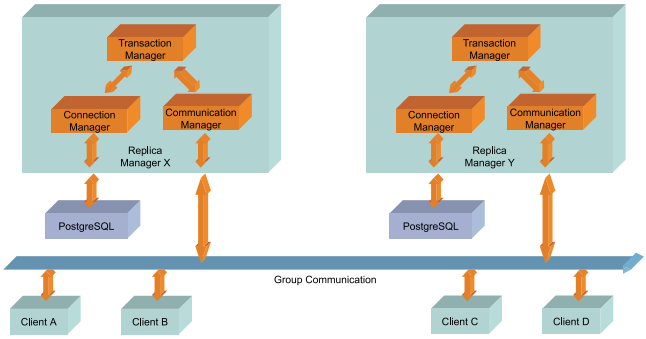


Figure 4: Main components

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 (sending and receiving commit messages) and interacts with clients (receiving transaction request and sending results) through the communication manager, and it submits transactions 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 implementation Ensemble [Hay98]). 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 [Pos98], 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.

4.1 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.

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 beginning of all its queues, it is to-delivered, and its commit message has been received. The transaction manager will then ask the connection manager to install the updates on the local copy. Once the transaction is completed, the transaction manager will remove it from all the queues.

4.2 Execution of Queries

Queries (read only transactions) are executed only at their local site. The idea is that queries will be 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.

This shortcut allows us to simulate snapshot isolation in a straightforward way and can be used with any other database management system that does not provide snapshot isolation (e.g., object oriented databases). It has the advantage that queries do not affect updates (unlike in [HAA99], a conflicting query does not abort an update transaction that is being executed) and do not need to limit their access to a single compound

conflict class. This is an interesting option for monitoring and analysis purposes since it allows queries to be run on a single site without any restrictions on the items they can access.

4.3 Optimistic Delivery

We have used Ensemble [Hay98] as communication layer. Ensemble is a group communication protocol stack providing virtual synchronous broadcast with different reliability and ordering properties.

In our protocol, update transactions use the total order broadcast with optimistic delivery. However, such a primitive is not available in any existing group communication system, including Ensemble. One way to implement this primitive is to modify the group communication protocol stack. As this was beyond the scope of this paper, we have implemented such a primitive on top of Ensemble. Although integrating the optimistic delivery into the protocol stack would have been more efficient, implementing it on top has served well our purposes.

In our implementation, each total order broadcast is performed 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.

4.4 Interaction with the Database

Previous work [KA00a] has explored how to combine data replication with group communication by implementing the protocol directly inside the transaction manager of the database. However, this approach most of the times is not feasible. In here, the protocol resides outside the database in a middleware layer.

Our approach has been to follow ideas from multilevel scheduling [Wei91] and composite systems [PA00] where an additional transactional scheduling layer is used to improve concurrency and to introduce higher order semantics as part of the scheduling procedures. The implementation described above combines, at the middleware layer, the replication protocol and a concurrency control mechanism based on the conflict classes defined. In fact, the protocol implements a simplified version of 2 Phase Locking [BHG87] as part of the queue handling mechanisms (simplified because transactions are received as a block and not progressively). To make sure that the database underneath does not reverse the serialization order, operations within one queue are submitted serially (equivalent to the way a data manager executes operations from a standard locking table); operations in different queues are submitted in parallel.

For reasons of space, we cannot get into the details of multilevel and composite transaction scheduling.

However, it is worth mentioning that the model presented can be extended to include operational semantics as part of the conflict classes so that the middleware layer and the database underneath perform scheduling at different semantic levels. This opens up very interesting possibilities for optimizations and make the system much more flexible.

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

In this section we analyze the scalability and overall performance of the protocol and the implementation we propose. It is important to emphasize that the absolute values of the results are not meaningful. They could be improved by simply using faster machines or by using a database other than PostreSQL. The important aspect of these results are the trends they show in terms of behavior as the number of sites and the load in the system increases.

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 that is maintained is an index on the primary key. The overall 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(1-1000)
```

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 $\frac{100,000}{16}$ data items 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. Needless to say, there are many other variations that could be used in terms of data organization or transactional load. However, we believe this setting is representative enough to test the scalability of the protocol. We purposefully did not use a standard database benchmark since the goal was to test the improvement achieved by the protocol we propose with respect the non-replicated database, instead of the database underneath, that was not very powerful, PostgreSQL. The purpose of our tests was to show that even with a freeware non-distributed database, it was possible to increase its throughput several times for small-medium cluster sizes using our middleware layer.

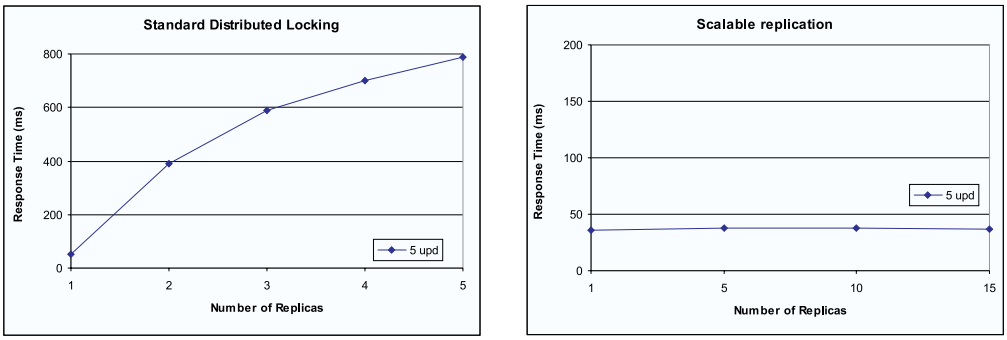
Parameters	Exp. 1	Exp. 2	Exp. 3	Exp. 4
Database size	10 tables of 10,000 tuples each			
Tuple size	approx. 100 bytes			
# of servers	1-15			
% of update txn	100%	0-100%	0-100%	100%
# of upd op. in txn	5	8	8	1
txn per second	10	max.	10-110	20-260
# write set size	504	804		104

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 [BHG87]). Gray et al. [GHOS96] 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 (Fig. 5.2(b)) with that of a commercial product that implements replication based on standard distributed locking (Fig. 5.2(a), borrowed from [KA00a]). 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).



(a) Standard distributed locking **(b)** Scalable replication

Figure 5: Comparison with Distributed Locking

Fig. 5.2(a) reflects the behavior predicted by Gray et al. and also captured by the analytical model presented in this paper. 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. This is due to the fact that distributed locking creates a significant amount of redundant work in the system, so much that adding extra sites becomes a liability more than an asset. Our system in comparison was quite stable. For the range of sites explored, the response time did not vary, indicating that the system is not adding any extra overhead when more sites are added.

5.3 Throughput Scale-out

The main motivation for this work is to provide a replication protocol that can scale in a cluster based system. 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.

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 scalability is at least 30% of the number of sites is a significant gain over conventional protocols where the scalability would have been 0% [GHOS96]. 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 (i.e., it decreases the writing overhead, wo, used in the analytical model). 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).

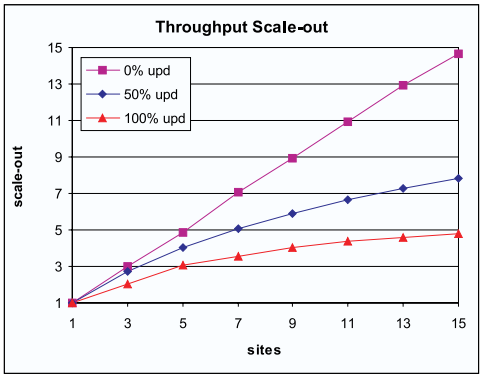
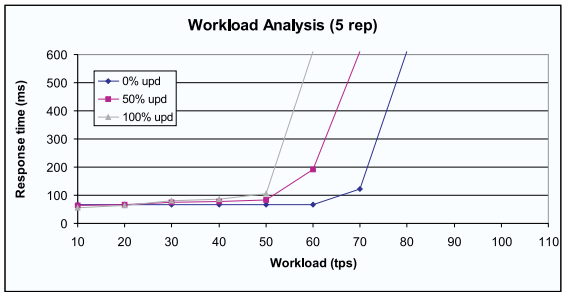


Figure 6: Scalability of the system for different transaction loads

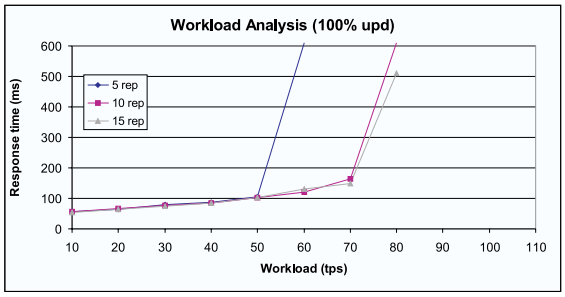
5.4 Response Time Analysis

In the first experiment we showed that the protocol we propose does not suffer from the limitations of conventional protocols. The second experiment proves that the protocol does indeed scale as more sites are added. This, however, does not mean that the response time of the entire system scales for all possible loads.

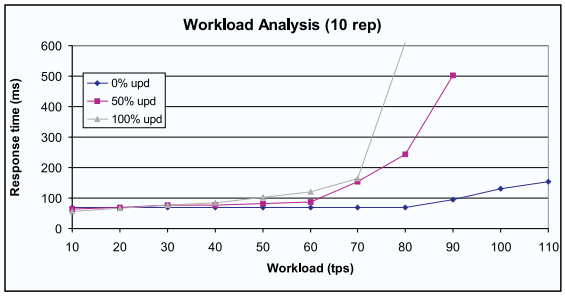
In this third experiment we try to find out the limits of the protocol by exploring when the response time becomes unacceptable. The idea is, given a system with a fixed number of sites, to increase the load while observing the response time until the response time is too high. As above, we consider 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 Figures Fig. 7(a-c).



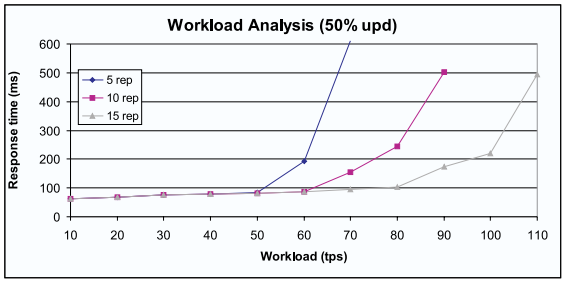
(a) 5 replicas



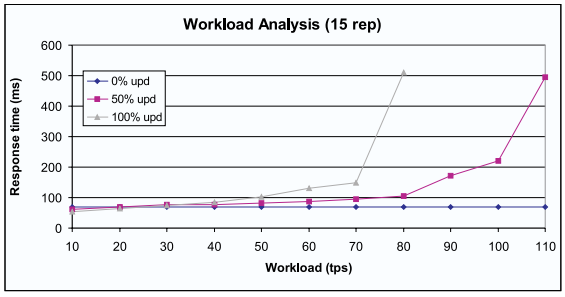
(d) 100% updates



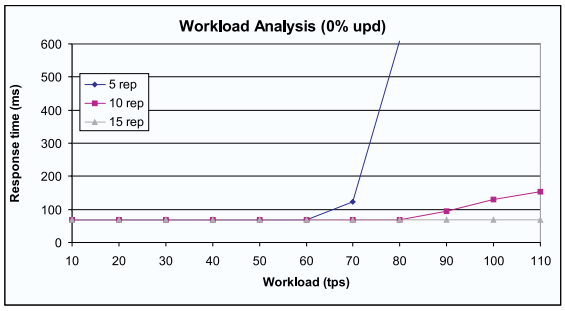
(b) 10 replicas



(e) 50% updates



(c) 15 replicas



(f) 0% updates

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 fact, one of the typical problems of conventional replication protocols is that they easily overload the network by generating too many messages (e.g., distributed locking generates one message per operation per transaction per site; a 10 site system running transactions of 10 operations at 50 transactions per second generates 5,000 messages per second).

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.

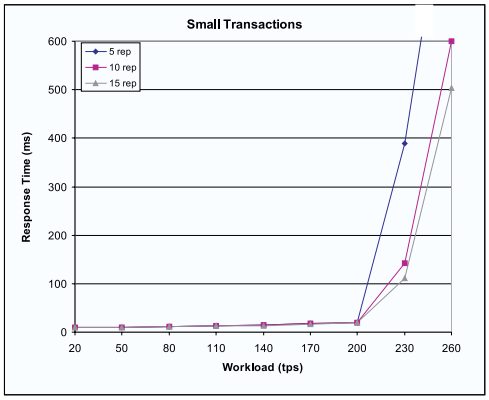


Figure 8: Small transactions

A last point to note regarding this experiment is the difference in scalability for short transactions (Fig. 8) and medium 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 [Kem00, HAA99], 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.

5.7 Empirical vs. Analytical Results

To validate the analytical model, we have compared its results with the values obtained during the experiments. We have found that they are well correlated (Fig. 9). The writing overhead factor was determined by curve fitting and set to $wo = 0.15$. The proportion of updates/queries was set to 1 for 100% updates and to 0.44 for 50% updates. This value comes from a slight difference in length between queries and updates (70ms and 55ms, respectively) that yields:

$$\frac{55}{70+55} = 0.44$$

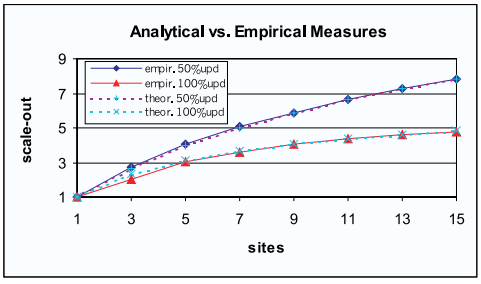


Figure 9: Comparison of Analytical and Empirical Results

6 Conclusions

High consistency has become a fundamental requirement for data cluster replication in many modern applications, like e-commerce. To achieve this level of consistency the only election are eager replication protocols. Unfortunately, conventional eager data replication protocols do not scale and are thus are unsuitable for many applications.

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

[BBG+95] 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, San Jose, USA, May 1995. ACM Press.

[BGRS00] K. Böhm, T. Grabs, U. Röhm, and H.J. Schek. Evaluating the Coordination Overhead of Replica Maintenance in a Cluster of Databases. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Proc. of of the 6th International Euro-Par Conference*, volume LNCS 1900, pages 435–444, Munich, Germany, September 2000. Springer.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, MA, 1987.

[Bir96] K.P. Birman. *Building Secure and Reliable Network Applications*. Prentice Hall, NJ, 1996.

[CKV01] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computer Surveys*, 2001.

[FGS99] P. Felber, R. Guerraoui, and A. Schiper. Replication of CORBA Objects. In *Advances in Distributed Systems*. Springer, 1999.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proc. of the SIGMOD*, pages 173–182, Montreal, 1996.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[HAA99] J. Holliday, D. Agrawal, and A. El Abbadi. The Performance of Database Replication with Group Communication. In *29th Int. Symp. on Fault-tolerant Computing, Wisconsin*, June 1999.

[HAA00] J. Holliday, D. Agrawal, and A. El Abbadi. Using Multicast Communication to Reduce Deadlock in Replicated Databases. In *Proc. of the Int. Symp. on Reliable Distributed Systems (SRDS)*, pages 196–205, October 2000.

[Hay98] M. Hayden. The Ensemble System. Technical Report TR-98-1662, Department of Computer Science. Cornell University, January 1998.

[HT93] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*, pages 97–145. Addison Wesley, 1993.

[JPAA01] R. Jiménez Peris, M. Patiño Martínez, G. Alonso, and S. Arevalo. A Low-Latency Non-Blocking Atomic Commitment. In *Proc. of the 15th Int. Conf. on Distributed Computing, DISC'01. LNCS-2180*, pages 93–107, Lisbon, Portugal, October 2001. Springer.

[JPAK01] R. Jiménez Peris, M. Patiño Martínez, G. Alonso, and B. Kemme. How to Select a Replication Protocol According to Scalability, Availability, and Communication Overhead. In *Proc. of the Int. Symp. on Reliable Distributed Systems (SRDS)*, New Orleans, Louisiana, October 2001. IEEE Computer Society Press.

[KA00a] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, A new way to implement Database Replication. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, Cairo, Egypt, September 2000.

[KA00b] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM TODS*, September 2000.

[Kem00] B. Kemme. *Database Replication for Clusters of Workstations*. PhD thesis, Dept. of Computer Science, Swiss Federal Institute of Technology Zurich, 2000.

[KPAS99] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *Proc. of 19th IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 424–431, 1999.

[Maf95] S. Maffeis. Adding Group Communication and Fault-Tolerance to CORBA. In *Proc. of 1995 USENIX Conf. on Object-Oriented Technologies*, June 1995.

[MSEL99] G. Morgan, S.K. Shrivastava, P.D. Ezhilchelvan, and M.C. Little. Design and Implementation of a CORBA Fault-tolerant Object Group Service. In *Proc. of the Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems, DAIS'99*, June 1999.

[NMMS99] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Enforcing Determinism for the Consistent Replication of Multithreaded CORBA Applications. In *Proc. of the Int. Symp. on Reliable Distributed Systems (SRDS)*, pages 263–273, Lausanne, Switzerland, 1999. IEEE Computer Society Press.

[Ora97] Oracle. *Oracle 8 (tm) Server Replication*. 1997.

[PA00] G. Pardon and G. Alonso. CheeTah: A Lightweight Transaction Serve for Plug and Play Internet Data Management. In *Proc. of 26 th Int. Conf. on Very Large Databases*, Cairo, Egypt, September 2000.

[PF00] F. Pedone and S. Frolund. Pronto: A Fast Failover Protocol for Off-the-shelf Commercial Databases. In *Proc. of 19th Symposium on Reliable Distributed Systems*, pages 176–185, Nuremberg, Germany, October 2000. IEEE Computer Society Press.

[PGS97] F. Pedone, R. Guerraoui, and A. Schiper. Transaction Reordering in Replicated Databases. In *Proc. of 16th Symposium on Reliable Distributed Systems (SRDS)*, pages 175–182, Durham, NC, October 1997. IEEE Computer Society Press.

[PJKA00] M. Patiño Martínez, R. Jiménez Peris, B. Kemme, and G. Alonso. Scalable Replication in Database Clusters. In *Proc. of Distributed Computing Conf., DISC'00. Toledo, Spain*, volume LNCS 1914, pages 315–329, October 2000.

[Pos98] PostgreSQL. *v6.4.2*. http://www.postgresql.com, January 1998.

[PS98] F. Pedone and A. Schiper. Optimistic Atomic Broadcast. In S. Kutten, editor, *Proc. of 12th Distributed Computing Conference*, volume LNCS 1499, pages 318–332. Springer, September 1998.

[SW00] R. Schenkel and G. Weikum. Integrating Snapshot Isolation into Transactional Federations. In *5th Int. Conf. on Cooperative Information Systems*, September 2000.

[Wei91] G. Weikum. Principles and Realization Strategies of Multi-level Transaction Management. *ACM Transactions on Database Systems*, 16(1):132–180, March 1991.