A Middleware for Consistent Data Replication: Is it feasible in WANs?

Yi Lin[1], Bettina Kemme[1], Marta Patiño-Martínez[2], and Ricardo Jiménez-Peris[2]

[1] McGill University, School of Computer Science, 3480 University Street, H3A 2A7, Montreal, Quebec, Canada
`(ylin30,kemme)@cs.mcgill.ca`

[2] Facultad de Informatica. Universidad Politecnica de Madrid , Spain
`(mpatino,rjimenez)@fi.upm.es`

Paper ID: 278

Abstract. Recent proposals have shown that middleware based database replication is able to provide 1-copy-serializability in LAN environments with excellent performance. This can be achieved by using powerful and fast multicast primitives that deliver messages to all sites in the same order in an all-or-nothing fashion. The question is whether a similar approach is feasible in WAN environments considering the increased message latency. Some of the approaches used in LANs might also work in WANs but others will be prohibitive. We have performed extensive tests of different protocols in a WAN testbed, in order to identify the most crucial bottlenecks, and the most promising optimizations. The performance results show that performance remains acceptable even for medium sized systems consisting of up to eight sites. As such, we believe that data replication guaranteeing 1-copy-serializability is a serious alternative to weaker approaches even in WAN environments.

Keywords: 1-copy-serializability, WAN, Group Communication, Data Replication, Atomicity

1 Introduction and Motivation

The tremendous advances in web technology, open-source application server and database technology, and cheap hardware have made e-transactions feasible in many domains. We buy books, sports and movie tickets, and other goods online, and we do our banking online. With the wide-spread implementation of such transaction processing systems comes the need for simple and cheap solutions for fault-tolerance, scalability, and fast response times. While large corporations go with expensive, proprietary solutions, small businesses look at open-source solutions with little hardware requirements, and rely on the internet for communication. Hence, replication, as the most common solution to fault-tolerance and scalability, has received a lot of attention in the last years.

This paper focuses on database replication. A big challenge for the replication protocol is to guarantee 1-copy-serializability and atomicity of the replicated transactions [7].

For a long time, 1-copy-serializable protocols have been considered prohibitively expensive in any kind of environment [13], and there existed only a few, proprietary solutions like Oracle's synchronous replication [23]. However, in recent years, a new class of 1-copy-serializable replication protocols has emerged [3, 6, 17, 11, 15, 9, 16, 18, 25, 10, 5, 4, 28]. Many of them use a middleware based approach to database replication. There are many reasons for that. Database systems are huge software systems, access to the source code is limited, and any optimized implementation within the database kernel will lead to a tight integration, disallowing replication across different database systems.

Many recent proposals ([17, 16, 10, 5, 9, 28, 4, 20]) have shown that middleware based replication provides scalability, fault-tolerance and fast execution in LANs. The idea is that when a transaction is submitted to the system its serialization order is determined and the middleware then makes sure that concurrent transactions are serialized according to this predefined order if they conflict. This solution is considerably faster than traditional distributed locking. Atomicity is achieved by returning a response to the client only when it is known that all replicas have received the necessary information to commit a transaction. Again, this is much faster than the traditional 2-phase-commit where the client has to wait until all sites have actually executed and committed the transaction.

While these approaches work well in LANs, little research has been done whether they can also be applied to WANs. [2] analyzes one particular protocol. [28] use multicast protocols that have been developed for WANs, however, an analysis of the replication protocols themselves has not been presented. Database replication in a WAN is at least as important as in a LAN although for different reasons. In an ideal world, it provides fast local access (no expensive access to a remote database server). Additionally, such a system is able to survive disaster cases where not only one machine crashes but all machines of a physical location.

While communication overhead plays little role in LANs it has a tremendous effect in WAN environments. As a result, basically all commercial WAN solutions are based on lazy replication. Transactions are executed only at one site, and changes are propagated to other sites only some time after the transaction commits. As such, transactions execute fast but the last transactions committed before a crash are lost losing atomicity. Only some restrictive lazy strategies guarantee consistency at the price of including communication in the client response time, and many allow inconsistencies between replicas [12, 23, 8]. It is difficult to detect these inconsistencies, and their reconciliation usually requires committed transactions to be rolled back. Hence, lazy approaches are not appropriate for transaction processing systems with high update rates.

As a result, it makes sense to revisit the existing successful replication solutions developed for LANs, and see whether they provide, although not excellent, but at least acceptable performance in WANs. Important goal is that throughput can be maintained or increased with increasing number of sites, while response times remain relatively low. While response times of less than 100 ms are probably not possible, a system with response times of a couple of seconds might still be acceptable for some applications. In fact, the client response time for existing web-based information systems is currently usually in the order of seconds.

In this paper, we analyze in Sections 2 and 3 existing middleware based solutions for database replication for LAN environments. For each solution alternative, we analyze its behavior and feasibility in a WAN. We have implemented a subset of the alternatives (Section 4), and provide an extensive performance analysis (Section 5). The key points are to reduce communication overhead as much as possible within the response time of an operation, to overlap execution with message delay, to separate queries from update transactions, to relax atomicity guarantees, and to keep the time transactions are stored at the middleware short. In such case, our performance results show that 1-copy-serializabilty and atomicity for critical transactions can be obtained with acceptable performance. In many of our experiments, response times remain below 1 second, and throughput increases over a centralized system. We are not aware of any other research that compares different replication strategies in such detail.

2 Middleware for Database Replication

There are basically two sets of recent approaches to provide 1-copy-serializability. The first approach has a controller or scheduler (centralized middleware). All database requests (e.g., SQL requests) are submitted to this controller ([5, 4, 9]. The controller then forwards queries to one replica, updates have to be executed at all replicas. The controller performs some concurrency control. In C-JDBC, e.g., the scheduler performs strict 2-phase-locking (or lower levels of isolation if requested) on a table basis. For that, the incoming SQL statements are parsed in order to determine the database tables to be accessed. The scheduler guarantees that all update operations are submitted to all replicas exactly in the same order. In order to facilitate concurrency control, [5, 4] require a transaction to indicate which tables it is going to access when it starts. [6] maintains a centralized serialization graph for concurrency control. In all cases, clients submit the operations of a transaction step by step requiring communication between controller and database replica for each operation. Having a centralized controller is unattractive in a WAN since a single scheduler forces remote messages between clients/data replica and the scheduler. Furthermore, exchanging one or more messages *per operation* of a transaction is prohibitive in a WAN.

In order to reduce the number of remote messages, it seems more attractive, that the middleware is distributed and an instance is installed in front of each database replica (on the same node or within a LAN). Clients contact the closest middleware instance to submit transactions. Hence, client/middleware and middleware/database communication is local.

Most of the proposals following this approach ([3, 2, 11, 15, 16, 25, 10, 28, 5, 4]) take advantage of powerful multicast primitives for the communication between the middleware instances [27]. The middleware instances build a process group. Each group member can multicast a message to the group, and each message is received by all members (including the sender). The semantics of the typical multicast primitives can be categorized by two parameters [14]. The *ordering* semantics that are interesting in the context of database replication are *unordered*, *FIFO* (messages of one sender are received in the order they were sent), and *total* (for each two members receiving m and m', both receive them in the same order). The *reliability* semantics are *unreliable* (no guarantee that a message will be received at all members), *reliable* (whenever a member receives a message and does not fail for sufficiently long time, then all other group members will receive the message unless they fail), and *uniform reliable* (whenever a member p receives a message, all other members will receive the message unless they fail –even if p fails immediately after the reception). Note that uniform reliable delivery provides all-or-nothing even in failure cases, while reliable delivery allows failed members to have received messages that are not received by the rest of the group. In some systems (e.g., Spread [29]), a combination of reliable and total order is called agreed delivery, and a combination of uniform reliable and total order is called safe delivery. We adopt this notation for its simplicity. The more powerful the ordering and/or reliability semantics, the more complex is their implementation and the higher the message delay. The replication tools now use the total order semantic to determine the serialization order of conflicting transactions, and the uniform reliable delivery guarantee to guarantee atomicity.

Most middleware based replication tools using above multicast primitives assume that the middleware receives one client request per transaction. This seems to be an acceptable restriction considering current information system architecture based on web- and/or application servers. For instance, the J2EE architecture disallows transactions to span web request due to the communication overhead. In an online bookstore, e.g., this means that browsing the books and then buying the books are different transactions. Since we want to use the approaches in a WAN setting, such a restriction is even more important. In the following, we assume that the program implementing the transaction either resides within the middleware, or can be called from the middleware.

We will now present a simple replication protocol that will serve as the baseline protocol for the optimizations in the next section. It distinguishes between read-only transactions consisting only of read operations, and update transactions containing at least one update statement. This can be determined by parsing the SQL statements of the program implementing the transaction. The replication middleware performs the following actions:

I. *Upon receiving a request for the execution of an update transaction from the client*: multicast the request to all sites with safe delivery.
II. *Upon receiving an update request in safe delivery or a read-only request from the client*: enqueue the request in a FIFO queue.
III. *Once a transaction is the first in the queue*: submit the transaction for execution.
IV. *Once a transaction finishes execution*: remove it from the queue, and return the response to the client.

This simple protocol does not allow any concurrency but executes all transactions serially according to the total order delivery of transaction messages, and guarantees atomicity by relying on uniform reliable delivery. The protocol requires one message within the response time of update transactions. Read-only transactions are executed locally. [2], for instance, follows this approach. In order to allow transactions to execute concurrently, many approaches assume that the objects to be accessed are known in advance (similar to the controller based approaches presented in [5, 4]). Again, we can achieve this by parsing the SQL statements and extracting the tables to be accessed. The middleware can then implement a lock manager. Upon receiving an update request with safe delivery or read-only requests from the client, the transaction manager atomically requests locks for all tables the transaction is going to access. When all locks are granted, the transaction can start executing (optimizations are possible). In this case, if transaction T1 is received before transaction T2 but they access different objects, both will get their locks granted, and hence, can execute concurrently. Only if T2 conflicts with T1 on at least one lock, it is delayed by T1 since it requests its locks after T1. [15, 16, 18, 25, 10] follow this or similar approaches.

3 Replication Strategies and their Usefulness for WANs

There exist many optimizations over the simple protocol presented above. They have been either analyzed extensively for LAN environments but not for WANs, or they have not been studied at all. In this section, we analyze several of these optimizations and their potential effect in a WAN environment.

3.1 Queries

A middleware system should not blindly apply its replication protocol to all types of transactions. In particular, read-only transactions, i.e., queries, need special treatment. First of all, and as mentioned above, queries can always be executed at only one site. In a LAN environment, this might not necessarily be the site the query is submitted to but any site with low load. In a WAN, execution at the local site always seems to be the best choice in order to avoid message overhead – especially since query answers can be large.

If the middleware acquires locks for queries, they might be delayed by writers that might take longer in WANs. A solution is to run queries with lower level of isolation as often done in centralized systems. In this case, we can simply submit queries to the database in read committed or uncommitted read isolation level without requesting locks at the middleware. With this, they will not delay the execution of writers, and will not be artificially delayed by the middleware. We also do not need to know all tables they are going to access. Even better, in systems that provide readers a committed snapshot of the data, e.g., Oracle and PostgreSQL, read operations never acquire locks. In such case, serializability for queries can be guaranteed without any extra effort by the middleware.

3.2 Write sets

Adding new replicas will increase the overall throughput if the workload has a lot of queries. However, if update rates are high, scalability cannot be achieved because update transactions are executed at all sites. In order to achieve scalability even with update intensive workloads, the idea is to also execute update transactions only at one site. At the end of execution, this site then propagates the changes in form of a write set to the other sites. The write set contains, e.g., the primary key values of the updated tuples along with the new physical values of those attributes that have been modified. [18]

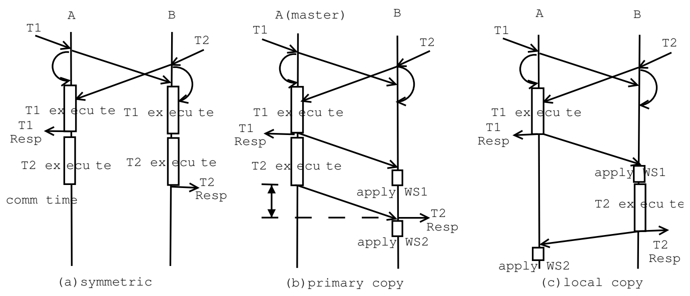


Fig. 1. With or without write set

tested the performance of applying write sets, and showed that it is faster than executing SQL statements, hence using less resources. In [16], this optimization was implemented at the middleware level, achieving scalability for a wide range of applications.

In fact, sending write sets might be the only feasible solution in some situations. If transactions have non-deterministic operations (e.g., set an attribute to the current local time), then, if the replicas execute the transaction independently, they will have different values for this attribute. In case of such non-determinism, we have to execute the transaction only at one site which then forwards the physical changes.

However, if write sets are sent within transaction boundaries response times might become too high. This depends on where update transactions are executed. In principle, any site can execute any update transaction. Upon receiving an update request with safe delivery, each site requests the locks for the transaction according to the total order. But only one site executes the transaction. The execution starts once all locks are granted locally. At the end of the execution, the write set is sent to all other sites. No ordering or uniform delivery is needed (since if the site crashes, another site can simply reexecute). The executing site can commit immediately, and release the locks. The other sites apply the write set once the locks are granted locally. Conflicting transactions might be executed at different sites, but all sites will either execute or apply the write sets of conflicting transactions according to the total order since locks are requested in this order.

Let's illustrate two possibilities with examples and compare them to the approach discussed in Section 2. Figure 1(a) depicts the *symmetric approach* described in Section 2 where each site executes the entire transaction (for updates). In the figure, update transactions T1 and T2 are submitted concurrently to sites A and B, multicast, and then executed according to the total order delivery at all sites (only in case they conflict).

Figure 1(b) and (c) depict two write set based variations. In the *primary copy approach* (Figure 1(b)), proposed in [22], each object (e.g., table), has a primary copy, and a transaction updating this table must execute on the site holding the primary copy. This is also called primary or master. If a transaction wants to access objects that have different masters, a simple assignment strategy can assign this transaction to the master of one of these objects. In the figure, assume that A is master for objects accessed by T1 and T2. If T1 and T2 conflict, A executes T1 before T2, otherwise concurrently. At the end of the execution of a transaction, A multicasts the write set to the other sites. For T1 it also returns a confirmation to the client since it is a local transaction. B, upon receiving T1 and T2 with safe delivery, requests the locks in this order but does not execute the transactions. Instead it waits for the write sets from A and then applies them (T1 before T2 if they conflict). It can return a confirmation to the client for T2 once it has received T2's write set (it does not need to wait until T2's write set is locally applied). Since for T1 the primary copy is the local site, the response time only includes the message overhead of the transaction message, and hence, is the same as in the symmetric approach. For T2, the response time also includes the delay of the write set since the local site is not the primary.

In a LAN setting, this approach has the advantage that, since each site is primary for a subset of objects, most of the execution a site performs is only on a subset of the data. Hence, it is more likely that these objects reside in main memory when requested which speeds up the execution. This is, of course, most effective, if also queries are executed on the primary copy. In a WAN, however, the scalability and locality gain might be lost by the increased message overhead.

In the *local copy approach* depicted in Figure 1(c), each update transaction is executed at the site it is submitted to. In the figure, T1 will be executed at A, and T2 at B. Since T1 is received before T2, A will first execute T1, multicasts the write set, commit and return the confirmation to the user. If T1 and T2 conflict, B waits to receive and apply T1's write set, then executes T2, multicasts its write set and returns the confirmation to the user. The message overhead is generally as follows. If two conflicting transactions are submitted concurrently at different sites as depicted in the figure, the first one to be received (T1) has no additional message delay, the second (T2), however, has to wait for the write set of the first to arrive before it can execute locally. This is similar to the primary copy approach. However, if there are no concurrent conflicting transactions (e.g., T1 and T2 do not conflict), the only message overhead is the transaction message itself as in the symmetric approach. We have developed the local copy approach in the context of this paper because we hope it has better response times than the primary copy approach in WAN environments.

Yet another alternative executes a transaction locally, and then the write set is sent with safe delivery. In this case, an optimistic concurrency control mechanism at the middleware must check for conflicts with concurrent, previously submitted transactions. [18, 28] follow this approach. The analysis of this approach in an WAN will be focus of future work.

3.3 Reliable v.s. Uniform Reliable

In order to guarantee atomicity of transactions although no 2-phase-commit is run, most solutions rely on uniform reliable message delivery. Whenever a group member receives a message, all other members will receive the message unless they fail. As such, when a site receives a transaction, executes and commits it, and then fails before sending the write set, we are sure that at least the transaction will be received at other sites. They can then take over and still reexecute the transaction so that the transaction is finally committed at all sites in the same order as at the failed site. The problem is that uniform reliable delivery is expensive in terms of message delay since it requires basically acknowledgements from all sites before a message can be delivered to the application. It is important to understand in which cases a non uniform reliable delivery violates transaction atomicity. Only in the case a site receives a user request, multicasts the request, receives it, executes and commits it, returns the response to the client and then fails while all other sites either do not receive the request or also fail, the surviving sites will not execute a transaction for which a client received a response. In a LAN, such case will occur seldomly. But since uniform delivery is fast, it is still not costly to prevent it. In a WAN, however, the unreliable Internet might provoke more such errors, especially since alive sites might be easily suspected to have failed. Hence, the choice between reliable and uniform reliable delivery depends completely on the application.

3.4 Optimisitic Delivery

Optimistic execution has received considerable attention recently. The principle idea is to deliver a message twice, once optimistically, and once when the desired degree of reliability and ordering has been established. First introduced in [19], only ordering was assumed. Here, a message is first delivered when it is physically received, and then again when the total order is established. The transaction can start executing upon the first delivery. If the order of optimistic delivery is not the same as the final total order, the transaction might have to be aborted, but only if the wrongly ordered transactions conflict. The advantage of optimistic delivery is that the time to determine the total order overlaps with transaction execution, and hence, potentially reduces the overall response time. In contrast, without this optimism, message delivery and transaction execution are sequential.

Variations of optimistic delivery exist. For instance, [1] suggests to delay optimistic delivery until one can be relatively sure that the optimistic order is not too far from the final total order. [21] consider uniform reliable delivery. In summary, one can combine optimistic delivery with final delivery taking different combinations of ordering and reliability into account. In regard to ordering, optimistic delivery can be (a) before the total order is established or (b) only afterwards. The final delivery is always after the total order is established. In regard to reliability, optimistic delivery is basically always at most reliable. The final delivery can be either (i) only reliable, or (ii) uniform reliable. Useful combinations are (a) with (i) or (ii), and (b) with (ii). If we combine (b) with (ii), then there is the guarantee of no aborts since already the optimistic delivery is in total order.

[16] has implemented a middleware replication tool taking advantage of optimistic delivery with combination (a)/(i). Upon optimistic delivery it requests the locks and starts execution once the locks are granted. However, the transaction is only committed after agreed delivery. If message order is not the same as in the optimistic order, transactions are undone if conflicting transactions should have been executed first. However, since the system is LAN based, the advantage of optimistic delivery is relatively small due to the fast network.

If we want to analyze the potential performance gain in a WAN, we have to consider two impacts of the WAN environment. The time between optimistic and final delivery is potentially much higher than in a LAN, hence, there is more potential to overlap

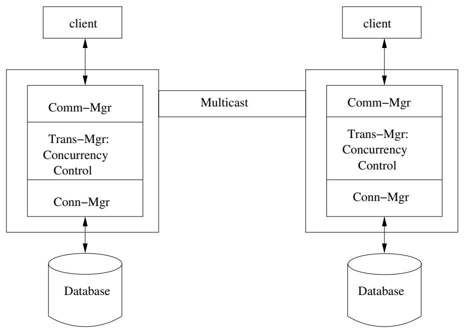


Fig. 2. Middleware Architecture

message delay and transaction execution. However, the probability that optimistic order and final order are different is also higher, possibly leading to higher abort rates.

4 System Description

Based on discussion of the previous section, there are 30(=2*3*(2+3)) protocols which are combinations of four aspects. (1) The middleware performs concurrency control for queries or not. (2) Transaction execution is symmetric, follows a primary copy approach, or is local at every site. (3) Message delivery is reliable, uniform-reliable, or (4) follows three alternatives for optimistic delivery.

We have implemented several of these combinations. Our implementation follows the architecture of Figure 2. We used PostgreSQL 7.2 as database backend. We extended PostgreSQL to provide two functions to the application. One to get the changes performed by a transaction in form of a write set, and a second that takes this write set as input and applies these changes without reexecuting the entire SQL statement. Other database systems already export such functionality, e.g., Microsoft SQLServer. This functionality is required for write set based replication. Furthermore, we used the open-source group communication system Spread [29], and enhanced it with an optimistic message delivery component.

The middleware is divided into three modules. The communication manager receives requests from clients (through any communication channel like sockets, RMI, HTTP), communicates with other middleware instances through multicast, and forwards transaction requests to the transaction manager. It also accepts write sets from the transaction manager and multicasts them to all sites. The transaction manager receives requests from the communication manager. It performs locking, and decides whether to handle queries in a special way or not. It also decides where each transaction is executed. When a transaction is executed locally, it forwards it to the connection manager, otherwise it waits until the write set once it is received from the communication manager. The connection manager manages a pool of open connections to the database system for efficiency reasons. Upon receiving transaction request, it is executed. In our prototype, transactions consist of a simple program executing a sequence of SQL statements. At the end of execution the connection manager retrieves the write set from the database if necessary, and forwards it to the transaction manager. The connection manager waits for confirmation from the transaction manager before committing a transaction. Upon receiving a write set, it applies the write set. The middleware is completely Java based. Due to the modularity of the framework it was very easy to implement the different protocols.

Spread and Optimistic Message Delivery The implementation of the multicast primitives has a great impact on the performance of the system. Hence, it is important to understand Spread and our extensions. The open-source version of Spread implements nearly all different multicast alternatives presented in Section 2 using a token based approach. A token circulates among all group members. If the application of a site wants to send a message it has to wait until the underlying Spread daemon has the token since only the token holder may send messages (independently of the message type). The token contains a counter, each new message to be sent in total order increases the counter and then gets the new value as timestamp. At each site, the Spread daemon delivers total order messages in timestamp order. That is, if Spread physically receives a message m but is missing a message with smaller timestamp, m is delayed until the preceding message arrives. However, for messages with no order requirement, Spread delivers the message immediately upon physical reception to the application. For messages with uniform reliable delivery, the token piggybacks acknowledgments, and only if a Spread daemon has received acknowledgments from all other group members is the message delivered. Note that [2], who also evaluate database replication in a WAN, also use Spread, however, they do not use the open-source version but a proprietary implementation with more efficient agreed and safe implementations.

We integrated the following optimistic delivery into Spread. A message sent with the new optimistic total order protocol, is first delivered to the application when it is physically received, and then again when the total order and uniform reliable delivery is established. That is, we follow combination (a)/(ii) of above.

At this point we would like to note that the token is a nice flow control feature for LANs, however, it has considerable impact on performance in WANs. The message delay for all reliable (non-uniform) message types is very similar and determined by how long the sender waits for the token. In a WAN, this can take considerable time (n/2 sequential token forwards if n is the number of group members). On the other hand, our optimistic delivery is in most cases already in the correct final total order since only at most one site is sending. Disordering can only happen if messages are lost or routed differently. We conducted experiments with three sites in Canada, Spain, and Switzerland at 90 messages per second (100 bytes per message), and there were only 12 out of order messages among over 20,000 messages sent. As a result, although we implemented combination (a)/(ii), it has nearly the semantics (and overhead) of a (b)/(ii) implementation.

5 Experimental Results

5.1 Experiment Setup

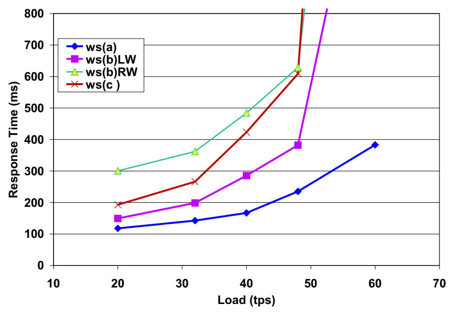
The experiments were conducted in Planetlab [26], an open, globally distributed computing infrastructure. We simply choose several educational sites in North America (attempting to choose sites that are not heavily loaded). All machines have similar strength (mostly Intel Pentium 4 or Xeon, 2.4GHz with 1GB memory) running Red Hat Linux 3.2.2-5.

Our database setup has been simple but flexible in order to be able to test with variable types of workloads. For most experiments the database consists of 10 tables, each with 10000 tuples. In our middleware, we use a table based locking scheme. Hence, the number of tuples in a table does not really have an impact on concurrency. There are two types of transactions, an update transaction performing 10 update operations, and a read-only transaction. The workload always consisted of 50% update transactions and 50% read-only transactions. In order to validate that this quite simplistic setup is still valuable in evaluating and comparing the different protocols, we also performed a suite of tests using the OSDL Database Test benchmark OSDL-DBT-1 [24] for PostgreSQL which implements the TPC-W benchmark specification of the Transactional Processing Council(TPC) [30]. The benchmark is also over 10 tables. Three typical workloads are browsing(95% read), shopping(80% read) and ordering(50% read). Hence, the experiments shown in this paper are a simplified version of the ordering workload of TPC-W (our update statements do not have such a complex table access pattern). Response times and maximum achievable throughput in the OSDL benchmark ordering workload are very similar to our system above in a centralized PostgreSQL database.

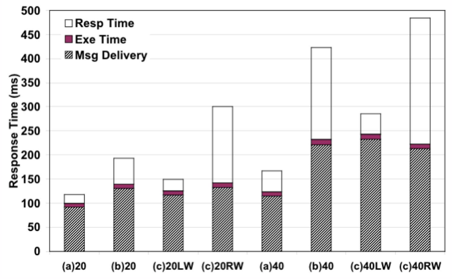
Test runs were conducted as follows. A client emulator on each node submits transactions to the middleware in such a rate as to achieve the desired system-wide workload. Submission is asynchronous, i.e., the emulator does not wait for the response. Instead, a receiver thread within the client will receive the responses and calculate the response time. This simulates a set of traditional synchronous clients submitting transactions in parallel. At startup each middleware instance creates a pool of connections to the local database. If more local transactions or write sets have to be executed concurrently than connections are available, they remain queued at the middleware even if they do not conflict.

For each test run, each client submits 800 transactions to the system. The first and last 10% of the results were discarded. Furthermore, we ran every experiment several times. For all our experiments the real results are with a 90% confidence within +/- 5% of the shown results. Using Planetlab, we did not have exclusive access to the machines, and many machines are heavily used. As a result, CPU utilization on the chosen machines changed from minute to minute, destroying not only the individual test run but basically the entire test suite since the test run had to be repeated on different machines. For this reason, most results are shown with only 4 sites.

In nearly all our experiments, we show the response or execution times with increasing workload submitted to the system. We do not show results when the system is overloaded, i.e., in all cases the throughput of the system is equal to the submitted load. In most cases, we show results as long as the response time is smaller than 800



(a) Response time



(b) Execution time at 20 and 40 tps

Fig. 3. Performance of different Write Set Options

milliseconds. We have set this as an upper limit of what might be acceptable for the response time of the database backend. In many cases, the load could be increased further without saturation of the system, however response times would increase above 1 second.

5.2 Centralized Database and Replication in a LAN

We first tested our benchmark on a single site. Each client connects directly to the database server. We only measure the performance after the connection has been established, and keep a client connected throughout the experiment. We conducted experiments with clients residing on the same node as the database, on a node within the LAN, and remote (WAN). For local clients response time never exceeded 25 ms up to the saturation point of close to 40 transactions per second (tps). For remote clients, response time was over 1500 milliseconds, and the system could only achieve a throughput of 5 tps. This is probably due to inappropriate handling of remote connections, and high message overhead. When we tested our middleware in a LAN setting (4 sites) a throughput of far over 100 tps was achieved, with response times starting at 25 ms and increasing only slowly with increasing load. The machines, however, were more powerful than the Planetlab machines used for WAN testing (P4, 3GHZ, 1GB memory).

5.3 Write Set Options

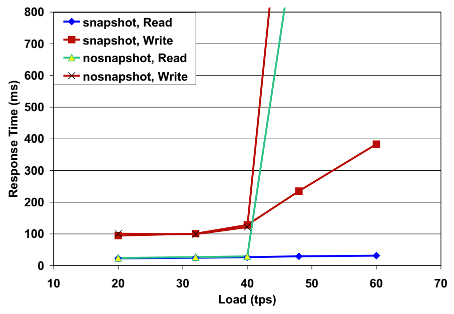
Our first tests in a WAN look at the write set alternatives since they have the largest impact on the performance of the system. In this experiment, we used 4 nodes, used agreed delivery for update transactions, and submitted queries directly to the DBS without acquiring locks at the middleware. Figure 3(a) depicts the response time of update transactions with increasing load for the three write set options discussed in Section 3.2. ws(a) depicts the symmetric approach, ws(b) the primary copy approach, and ws(c) the local copy approach. The primary copy approach has two different values. One is for update transactions that are submitted to the primary copy, called local writes (LW).

The other is for transactions that are submitted to a node that is not primary for this transaction, called remote writes (RW).

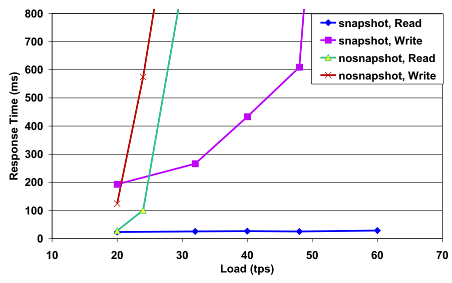
Response times vary between 100 ms to 800 ms up to a load of around 50 tps. Higher loads up to 60 tps can be achieved for the symmetric approach, or if response times over 1 second are acceptable (1.5 -3 seconds). Note that the achieved throughput is higher than in a centralized system that only achieves 40 tps. However, it does not scale as well as the LAN approach with far over 100 tps. The symmetric approach has the best performance throughout the experiment. Local writes in the primary copy approach have lower response time than the transactions in the local copy approach. Finally, remote writes in the primary copy approach experience the worst response time. This is immediately linked to the number of messages sent. Figure 3(b) shows a more detailed analysis.

It first depicts for a load of 20 tps how execution time is spent for transactions with the symmetric approach (a)20, for local writes of the primary copy approach (b)20LW, for remote writes of the primary copy approach (b)20RW, and for transaction with the local copy approach (c)20. Then it does the same analysis for 40 tps. In the symmetric approach, every transaction has only one message within the response time. This message makes most of the response time. The execution itself is very fast (as in all approaches). The rest of the response time is spent at the middleware, mostly waiting in queues, or with client communication. Comparing a load of 20 tps with 40 tps, we see that all times increase (although in different proportion). Although the primary copy approach also has only one message delay for local writes, the response time is slightly higher than in the symmetric approach. This is due to the higher loaded group communication system that has to handle double as many messages than in the symmetric approach. We can see that the message delay for the transaction message is longer (more extreme for 40 tps than for 20 tps) while execution and remaining time is similar than in the symmetric approach. Remote writes of the primary copy approach have to wait for the write set which is reflected within the white part of the response time. Note that in a system with 4 sites, on average 25% of transactions are local, while 75% are remote (unless there is locality in client requests and clients are more likely to submit requests directly to the primary). For the local copy approach some of the transactions have no write set delay within the response time. This is more likely at low loads when there are less transactions in the system, and hence, less transactions conflict. As a result, at 20 tps the response time for the local approach is closer to that of local writes of the primary copy approach, at 40 tps it is closer to that of remote writes of the primary copy approach. If there is locality, i.e., clients of different nodes are unlikely to conflict, then the response time will always be close to the response time of local writes of the primary copy approach.

As a result, while an asymmetric approach has proven to boost performance in a LAN it seems to be less useful in a WAN due to the additional overhead of sending the write set which might be included in the response time. It also puts higher load on the group communication system. If a write set based approach is needed due to non-determinism, the local approach seems favorable over the primary copy approach since on average response times are smaller.



(a) Symmetric approach



(b) Local copy approach

Fig. 4. Snapshots vs. middleware based locking: response times for queries and updates

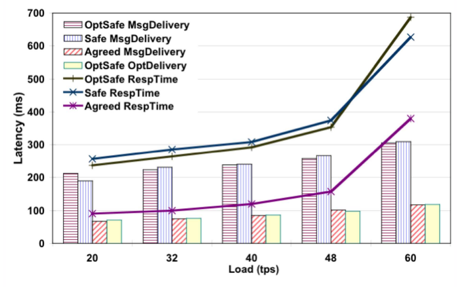
5.4 Special Treatment of Queries

The next experiment suite shows the importance of treating queries in a special way. Figure 4 shows the response time for update transactions and queries in a system with 4 sites for increasing load. Message delivery for updates was with agreed delivery. We show results for the symmetric (Figure 4(a)) and local copy approach (Figure 4(b)). Queries are either immediately submitted to the database relying on PostgreSQL's snapshot concurrency control (denoted as *snapshot* in the figure), or included into the lock queues of the middleware system (denoted as *nosnapshot* in the figure).

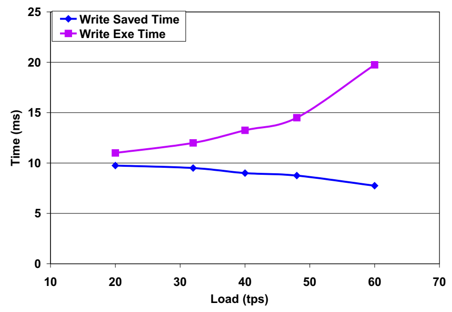
For the symmetric approach, the difference is only very small for small to medium loads, while snapshots for queries are significantly better than locking with the local copy approach (the same holds for primary copy approach). The reason for that different behavior is the time difference in regard to how long update transactions are queued at the middleware. In the symmetric approach, a transaction is queued when it is received and dequeued when it is locally committed. This includes local execution time and time waiting for conflicting transactions to finish. This time is generally very short until the response time for updates increases (at around 50 tps). Hence, queries and update transactions do not hinder each other significantly for low loads. With the local copy approach, some update transactions wait for write sets of conflicting transactions to be delivered even at low throughputs. They block succeeding transactions until these write set messages arrive. Also, queries hinder update transactions that are ordered behind the queries. As a result, the system deteriorates fast. Hence, using special solutions for queries seems to be necessary whenever transactions are delayed at the middleware layer. This is particularly the case when a transaction waits for a message to arrive before terminating. In the following, all presented experiments use snapshots for queries.

5.5 The cost of atomicity

In this section, we analyze the performance of update transactions using the symmetric approach depending on whether the transaction message is sent using agreed message



(a) Agreed vs. Opt-safe vs. Safe (symmetric)



(b) Analyzing Opt-safe (symmetric)

Fig. 5. (a) Delivery guarantees: delivery and response times and (b) Analysis of opt-safe delivery

delivery (reliable), safe message delivery (uniform reliable), or our opt-safe delivery where the optimistic delivery is immediately after the physical reception while the final delivery is at the same time as a safe delivery. Safe and opt-safe have the same final message delivery guaranteeing transaction atomicity, but opt-safe can overlap transaction execution with the message delay (to some degree in our implementation) at the cost of potentially having to abort transactions. The results for primary copy and local copy approach show the same relative performance behavior as the results presented here with the symmetric approach. However, response times are generally higher.

Figure 5(a) shows as curves the response time of update transactions for the three different delivery mechanisms with increasing load. The bars indicate the time of optimistic delivery, agreed delivery, safe delivery, and opt-safe delivery. As can be predicted, the optimistic delivery takes as long as the agreed delivery because of the token based approach of the group communication system. Opt-safe and safe delivery are also nearly the same (providing the same guarantee). All delivery delays increase with increasing load, and the transaction response time increases accordingly. The reason is the higher load leading to more CPU consumption plus more lock contention at the middleware. Throughout the experiment, agreed performs better than opt-safe, and opt-safe performs slightly better than safe. Agreed is best because it can commit a transaction once its total order is determined and the transaction is executed. Opt-safe will start to execute upon optimistic delivery but has to wait for the final delivery. Hence, the response time is worse than for agreed. opt-safe is faster than safe since it overlaps transaction execution while guaranteeing uniformity. However, the difference is very small since transaction execution is very small compared to the safe message delay. However, all results are well below 1 second even for safe delivery. That is, 100% atomicity can be achieved if response times of 1 second are acceptable. A more efficient implementation of agreed and/or safe delivery might further decrease response times.

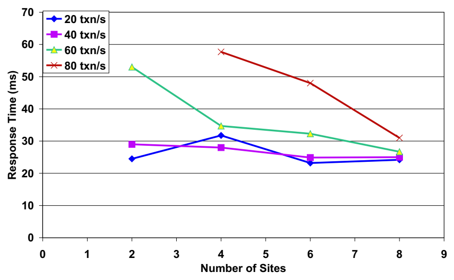
5.6 A closer look at opt-safe

Let's have a closer look at how much time can really be saved by using optimistic delivery. The best that can be achieved is if execution time (from the time the transaction is the first in all queues and the first operation is submitted to the database until completing the last operation) completely overlaps with determining the total order or safe delivery. The worst case occurs when the transaction starts executing after the final delivery; then we have not saved anything. We can save some time if the transaction starts executing but does not finish execution before final delivery. Figure 5(b) shows an analysis of saved time of the experiment of the last section. The figure shows the real execution time, and the time this execution overlapped with the safe delivery. We see that at low load, there is a complete overlap but since execution time is small, the overall gain is not big. With increasing load, execution times increase due to resource consumption at the database. However, less and less of this time overlaps with the delivery. The reason is that at higher loads, more transactions are in the system, and hence, more transactions are queued on the lock table at the middleware. Although a transaction is already opt-delivered and enqueued it will not execute because it is not the first in the queue. It is actually quite likely that it will be already finally delivered before it actually can start executing. If the middleware would use finer granularity locking, more time could be saved (whilst the database is not saturated).

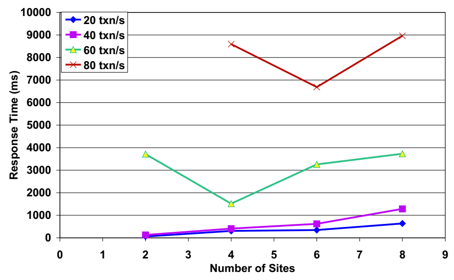
At this timepoint, we would like to discuss how far Spread, and our implementation has influenced the performance of opt-safe. As mentioned in Section 4, the token based approach leads to nearly identical message delay for unreliable, reliable FIFO, and agreed delivery. As a result, our optimistic delivery is nearly as expensive as agreed. We can see the performance penalty in our results. On the other hand it gives us a nearly total order at the time of optimistic delivery. As a result, we have barely any aborts due to wrong orderings. If one considers a different total order implementation, where sites send independently, disordering is much higher since the sending site receives a message immediately while there is delay to other sites. This might lead to many aborts if the middleware uses table based locking, and the system has a high throughput. Some of these aborts might be unnecessary, since the transaction might not actually conflict, but a coarse granularity of the middleware cannot detect this.

5.7 Scalability in WAN

In this experiment we want to see how the performance changes with an increasing number of sites. In our first test suite we use opt-safe delivery and the primary copy approach as an example of a system where we can expect high response times. Figure 6(a) shows the response time of queries when the number of sites increases from two to eight. The different curves show the response time for different system loads. We can see that for all loads the response time decreases. The reason for this is that by increasing the number of sites but keep the system load steady, we decrease the load of queries that each site has to perform, hence, CPU is less utilized leading to faster responses. Figure 6(b) shows the response time for local update transactions. At a low load (20/40 tps), we can see that the response time increases slowly but steadily and is at just above 1000 ms for a throughput of 40 tps. The reason for the increase is not a

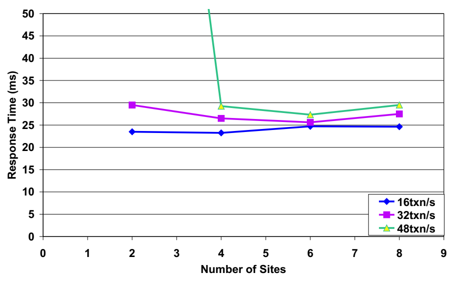


(a) Queries

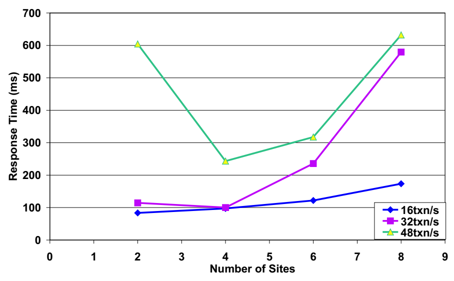


(b) Local update transactions

Fig. 6. Scalability: Response times for different system sizes with local copy approach



(a) Queries



(b) Local update transactions

Fig. 7. Scalability: Response times for different system sizes with symmetric approach

heavier load (the load per node is, as mentioned above, actually less) but the increase in message delay with an increasing number of sites.

When looking at 60 transactions per second, however, the behavior is different. Response times are generally much higher due to high CPU utilization and lie in the seconds. When moving from 2 to 4 sites, response time decreases because the load is distributed among more sites (remember that read-only transactions are only executed at one site, and for updates, remote sites only apply the write set). This alleviates the load on each site, and hence, response times are faster even for updates. That is, we gain more by adding CPU power than we loose by increasing communication delay. However, if we move now to 6 and 8 sites, communication delay again becomes the predominant factor and response time increases again. Although we don't think that the response times for a load of 80 transactions per second are acceptable (more than 7 seconds), we show them for two reasons. Firstly, as a proof that such a load can still be handled by the system, although at a price of high response times. Secondly, as an indicator that even at these high loads adding new replicas can help to improve performance. Two replicas were not able to handle 80 tps, but more than two are. Moving from four to six replicas even helped decreasing the response time.

The second test suite looks at the performance with agreed delivery, using a symmetric approach. Figures 7(a-b) show the response times of queries and update transactions. For queries, the response again decreases with increasing number of sites, but much less than in the primary copy approach. The reason is that in the symmetric approach all update transactions are executed at all sites, while the primary copy approach only executes it once and applies write sets otherwise. Hence, the primary copy approach can take better advantage of the increased computing power of more sites. For update transactions, the results are similar than for the primary copy approach, but response times are generally better due to the agreed delivery.

6 Conclusions

In this paper, we present a detailed performance analysis studying the feasibility of 1-copy-serializabily and atomicity guaranteeing database replication in WAN environments. We analyze in detail the effect of different, middleware based data replication strategies on the performance in a WAN environment. In general, we believe that consistent database replication is feasible. Throughput does not deteriorate and can even be increased with increasing number of sites. Our results show that the system can scale up to a medium size of 8 sites. Response time is clearly affected by the high message delay in a WAN environment. However, by keeping the number of messages exchanged within the response time of a transaction to a minimum, the overall response time remains acceptable.

Summarizing the different alternatives, a symmetric approach seems to be preferable over write set based approaches because the latter have more than one message exchange within the response time of some transactions. However, a symmetric approach might not be feasible for non-deterministic SQL statements. In that case, the local copy approach works better than a primary copy approach for the majority of transactions. Handling queries in an optimized way is even more crucial in a distributed environment than in a centralized database due to the increased response time of update transactions that might hold locks for much longer than in a centralized system. Choosing agreed delivery over safe delivery can help reduce response times, and it can be expected, that transaction atomicity will not be violated too often, although possible. But even if the application requires atomicity in all situations, safe delivery can still lead to acceptable response times. Optimistic delivery is able to overlap execution time with message delay. In our experiments, the gain was not very high. However, with other implementations of optimistic delivery, or if transactions generally take longer to execute (what would be the case with a database not fitting in memory), it might have a considerable impact.

In future work, we want to look at a couple of further issues. We want to analyze how the concurrency control at the middleware has an influence on the performance in a WAN, and whether a finer locking granularity will lead to better results. This could be achieved, for instance, with a parameter based approach of detecting conflicts between transactions or through a tighter cooperation between database system and the concurrency control component of the middleware. Furthermore, we are looking at other implementations of optimistic delivery that might lead to higher performance gains.

References

1. A. Sousa and J. Pereira and F. Moura and R. Oliveira. Optimistic Total Order in Wide Area Networks. In *Proc. of Int. Symp. on Reliable Distr. Systems*, 2002.
2. Y. Amir, C. Danilov, M. Miskin-Amir, J. Stanton, and C. Tutu. On the Performance of Consistent Wide-Area Database Replication. Technical Report CNDS-2003-3, CNDS, John Hopkins University, 2003.
3. Y. Amir and C. Tutu. From Total Order to Database Replication. In *Proc. of Int. Conf. on Distr. Comp. Systems (ICDCS)*, July 2002.
4. C. Amza, A. L. Cox, and W. Zwaenepoel. Conflict-Aware Scheduling for Dynamic Content Applications. In *USENIX Symp. on Internet Technologies and Systems*, 2003.
5. C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed Versioning: Consistent Replication for Scaling Back-End Databases of Dynamic Content Web Sites. In *Proc. of Middleware*, volume LNCS 2672, pages 282–304, 2003.
6. T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool. Replication, Consistency, and Practicality: Are These Mutually Exclusive? In *ACM SIGMOD Conf.*, 1998.
7. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, MA, 1987.
8. Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. In *ACM SIGMOD Conf.*, pages 97–108, Philadephia, Pennsylvania, June 1999.
9. E. Cecchet, J. Marguerite, and W. Zwaenepoel. RAIDb: Redundant Array of Inexpensive Databases. Technical Report Technical Report 4921, Inria, 2003.
10. E. Pacitti, T. Özsu, C. Coulon. Preventive Multi-master Replication in a Cluster of Autonomous Databases. In *Euro-Par Conf.*, 1997.
11. U. Fritzke and P. Ingels. Transactions on Partially Replicated Data based on Reliable and Atomic Multicasts. In *Proc. of the IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 284–291, 2001.
12. R. Goldring. A discussion of relational database replication technology. *InfoDB*, 8(1), 1994.
13. 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.
14. V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, pages 97–145. Addison-Wesley, 1993.
15. J. Holliday, D. Agrawal, and A. E. Abbadi. The Performance of Database Replication with Group Communication. In *29th Int. Symp. on Fault-tolerant Computing, Wisconsin*, June 1999.
16. R. Jiménez-Peris, M. P. no Martínez, B. Kemme, and G. Alonso. Scalable Database Replication Middleware. In *Proc. of 22nd IEEE Int. Conf. on Distributed Computing Systems, 2002*, Vienna, Austria, July 2002.
17. K. Böhm and T. Grabs and U. Röhm and H.J. Schek. Evaluating the Coordination Overhead of Replica Maintenance in a Cluster of Databases. In *Proc. of Intern. Euro-Par Conf.*, volume LNCS 1900. Springer, Sept. 2000.
18. 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)*, 2000.
19. 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.
20. A. I. Kistijantoro, G. Morgan, S. K. Shrivastava, and M. C. Little. Component Replication in Distributed Systems: A Case Study Using Enterprise Java Beans. In *Proc. Of SRDS*, pages 89–98, 2003.

21. L. Rodrigues and P. Vicente. An Indulgent Uniform Total Order Algorithm with Optimistic Delivery. In *Proc. of the Int. Symp. on Reliable Distributed Systems (SRDS)*, 2002.
22. M. Patiño-Martínez and R. Jiménez-Peris and 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, Oct. 2000.
23. Oracle. *Oracle 8 (tm) Server Replication*. 1997.
24. OSDL. OSDL Database Test 1 (OSDL-DBT-1). homepage: http://www.osdl.org/.
25. F. Pedone, R. Guerraoui, and A. Schiper. Exploiting Atomic Broadcast in Replicated Databases. In D. J. Pritchard and J. Reeve, editors, *Proc. of 4th International Euro-Par Conference*, volume LNCS 1470, pages 513–520. Springer, Sept. 1998.
26. Planetlab. homepage: http://www.planet-lab.org/.
27. D. Powell and other. Group communication (special issue). *Communications of the ACM*, 39(4):50–97, April 1996.
28. L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. Strong Replication in the GlobData Middleware. In *Workshop on Dependable Middleware-Based Systems*, pages 503–510. IEEE Computer Society Press, 2002.
29. Spread. homepage: http://www.spread.org/.
30. Transaction Processing Performance Counil. homepage: http://www.tpc.org/.