A Suite of Database Replication Protocols based on Group Communication Primitives [*†]

Bettina Kemme **Gustavo Alonso**
Information and Communication Systems Research Group
Institute of Information Systems, Swiss Federal Institute of Technology (ETH)
ETH Zentrum, CH-8092 Zürich
E-mail: {kemme,alonso}@inf.ethz.ch

Abstract

This paper proposes a family of replication protocols based on group communication in order to address some of the concerns expressed by database designers regarding existing replication solutions. Due to these concerns, current database systems allow inconsistencies and often resort to centralized approaches, thereby reducing some of the key advantages provided by replication. The protocols presented in this paper take advantage of the semantics of group communication and use relaxed isolation guarantees to eliminate the possibility of deadlocks, reduce the message overhead, and increase performance. A simulation study shows the feasibility of the approach and the flexibility with which different types of bottlenecks can be circumvented.

1 Introduction

Replication is often seen as a mechanism to increase availability and performance in distributed databases. Most of the work done in this area, which we will refer to as traditional replication protocols, is on *synchronous* and *update everywhere* protocols based on *1-copy-serializability* [4], whereby an object must appear as one logical copy and the execution of concurrent transactions is coordinated so that it is equivalent to a serial execution over the logical copy. Unfortunately, very few of the ideas proposed along these lines are currently used in commercial products. There is a strong belief among database designers that such solutions are not feasible due to performance and scalability problems: the probability of deadlocks is directly proportional to the third power of the number of nodes in the system, serializability is often considered too restrictive even in a centralized database, and the message and logging overhead is extremely high, leading to resource contention and long transaction response times [8]. These considerations have lead current products to use *asynchronous* and *primary copy* replication instead [7, 14]. Asynchronous protocols propagate updates only after the transaction has committed, which decreases the response time but introduces data inconsistencies. Primary copy approaches centralize updates in a single copy, which eliminates concurrent updates but introduces a single point of failure. While this approach may be criticized, some of the arguments against traditional research solutions are justified, especially from the point of view of commercial products where performance often takes precedence over any other consideration.

In view of the gap between theory and practice, the question that needs to be addressed is whether it is possible to design synchronous, update everywhere protocols that do not suffer from the drawbacks outlined above. We believe that the answer lies in a tighter integration between transaction management and the underlying communication system. Following some initial work in this direction [1, 3, 15, 17], the idea is to exploit *group communication* [6] to push down in the software hierarchy the more basic functions, thereby avoiding some of the performance limitations of current solutions. In addition, the proposed protocols also take into consideration the fact that databases usually provide a range of consistency levels, commonly much less restrictive than those considered in traditional approaches. The family of protocols presented can thus be easily integrated in current sys-

*Part of this work has been funded by ETH Zürich within the DRAGON Research Project (Reg-Nr. 41-2642.5)

tems, provide reasonable performance, and the same transactional semantics found in centralized systems.

The paper is organized as follows: Section 2 describes the system model. Section 3 presents replication protocols with different isolation levels. Section 4 addresses fault-tolerance. Section 5 presents a performance evaluation. Section 6 summarizes the results.

2 Model

A distributed database consists of a number of nodes, N, that communicate via message passing, and a number of objects, X. Objects are accessed by executing transactions. We assume a fully replicated system, i.e., all nodes have copies of exactly the same objects. The components of interest in our context are the communication system, the transaction manager and the concurrency and replica control.

2.1 Communication Model

Group communication primitives [10, 6] are used according to the following notation. A node N *broadcasts/sends* a message to all nodes of a group. After the *reception* of a message at N, the message is *delivered* when the order is determined and the delivery guarantees are fulfilled. Of all the possible semantics of group communication, we are interested in the ordering and atomicity of message delivery and group maintenance. Regarding ordering of messages, we distinguish a basic service (no ordering guarantees) and a total order service (same total order at all sites). In regard to atomicity we consider *atomic delivery* (a message is delivered only when there is a guarantee it will be delivered at all sites) and *non-atomic delivery* where we only assume that a message sent between two correct nodes will eventually be delivered. Note that our notion of atomicity is related to the "uniformity" problem in that it encompasses all processes, not only the correct ones [16]. Failure detection is provided by the group maintenance services, which will exclude a failed node from the group. Before a group change becomes visible to the application (here the database system), the members of the new group coordinate the delivery of pending messages from a failed node providing the required atomicity level. In particular, if a node might have delivered an atomic message before its failure, the message must be delivered at all nodes before the group change becomes active.

From a performance point of view, the different semantics differ in the message overhead and delivery delays, with basic message order suffering from less delays than total order and non-atomic delivery requiring less messages than atomic delivery.

2.2 Transaction and Concurrency Models

A transaction T_i is a partial order of read $r_i(X)$ and write $w_i(X)$ operations. Transactions are executed atomically, i.e., a transaction either commits, c_i, or aborts, a_i, the results of all its operations [4]. Two operations conflict if they are from different transactions, access the same object and at least one of them is a write. In order to guarantee correct executions, transactions with conflicting operations must be isolated from each other. For this purpose, different *levels of isolation* are used [9]. The different levels are a trade-off between correctness and performance in an attempt to maximize the degree of concurrency by reducing the conflict profile of transactions.

In most systems, locking protocols are used to implement isolation levels. Before accessing an object, a transaction acquires a read or a write lock on the object. There may not be two locks for conflicting operations active on an object at the same time. Since write locks can not be released until commit time for a variety of reasons (mainly for recovery purposes), the only possibility to reduce the conflict profile of a transaction is to release read locks as early as possible or to not get read locks at all. Serializability is commonly implemented using *strict 2-phase-locking (2PL)*: both read and write locks are kept until the end of the transaction (EOT). *Cursor stability* places a lock on an item as long as an SQL cursor is positioned on that item, but the lock is released as soon as the cursor moves on. This may lead to non-repeatable reads. To avoid lost updates, read locks are released only if the transaction does not write to that item. *Snapshot isolation* eliminates read locks by forcing transactions to "read from the log", instead of from the data items themselves. Hence, conflicts can only be detected between write operations and some read/write anomalies might occur. Finally, we also use a *hybrid protocol* in which 2PL is used for update transactions while read-only transactions use snapshot isolation. This protocol guarantees serializability but update transactions and queries must be identified in advance.

2.3 Replica Control Model

For simplicity, we use a version of the *all available copies* approach [4]. A transaction T_i invoked at a node N is said to be *local* to N. For other nodes, T_i is a *remote* transaction. All read operations of T_i are performed on the local copies of N. The write operations are deferred until all read operations have been executed and a description of the set of write operations, WS_i, is bundled into a single message and broadcast to the group (including the local node).

We exploit the total order provided by the com-

The lock manager of each node N coordinates the operation requests of the transactions as follows:

1. *A local transaction T_i makes a read request $r_i(X)$:* if there is no write lock on X, then grant the lock, else wait until the lock can be granted.
2. *A local transaction T_i makes a write request $w_i(X)$:* defer the write operation until T_i has submitted all operations.
3. *A local transaction T_i has submitted all operations:* form the write set WS_i and send it using the total order service.
4. *Upon delivery of WS_i, process it in an atomic step:*
   a. Request for each object X where $\exists w_i(X) \in WS_i$ a write lock:
      i. If there is a granted read lock $r_j(X)$ and the write set WS_j of T_j has not yet been delivered, abort T_j and grant $w_i(X)$. If WS_j has already been sent, then broadcast a_j using the basic service. Later N itself will ignore both the WS_j and a_j messages.
      ii. If there is another write lock on X or all read locks on X are from transactions whose write sets have already been delivered, then wait until all other locks are released and the new lock can be granted.
      iii. If there is no other lock on X, then grant the lock.
   b. If T_i is a local transaction, broadcast c_i using the basic service.
5. *Upon delivery of c_i:* wait until all operations of T_i have been executed, then commit T_i and release all locks.
6. *Upon delivery of a_i:* undo all operations already executed and release all locks (granted and waiting ones).

Figure 1: Replication protocol guaranteeing serializability

munication system in order to decide on the order of conflicting transactions. This concept of ordering is at the core of the replication mechanism. Upon delivery of the write set WS_i, T_i will only start the execution of an operation on an object X *after* all conflicting operations of previous transactions have been executed. This is done by handling the lock requests within WS_i as a single step before processing the next write set. This, however, does not imply that transactions are processed sequentially. Non conflicting operations of different transactions can be executed in parallel. Note that, with this approach, deadlocks due to write/write conflicts are avoided entirely. Since the write sets are delivered at all nodes in the same order, single object deadlocks cannot occur. Furthermore, deadlocks involving two or more objects are also avoided because the write locks of a transaction are requested within a single step. Thus, one of the main concerns regarding replication is avoided by bundling write operations and relying on the communication to provide some ordering of the locking requests.

3 Database Replication Protocols

In what follows we present three replication protocols designed to avoid two of the drawbacks of synchronous replication. First, they avoid any type of deadlock as described before. Second, the protocols provide different levels of isolation to be able to capture the varying requirements of different applications.

3.1 Serializability (SER)

Figure 1 describes a replicated version of the strict 2PL protocol based on one of the protocols proposed in [1]. Upon delivery of a write set message WS (totally ordered by the underlying communication mechanism), the transaction manager acquires all the necessary locks (aborting any conflicting readers, see below), thus guaranteeing that conflicting transactions are executed in the same order at all sites.

To avoid deadlocks in the case of read/write conflicts the algorithm aborts read operations when conflicting write operations arrive. It is necessary to give write operations priority over read operations because read operations are only known locally but write operations are executed globally. In this protocol, the execution of a transaction T_i requires two messages. One for the write set and a second with the decision to abort or commit, since only the owner of T_i knows about the read operations of T_i and, therefore, about a possible abort of T_i. The "causal and atomic bcast based protocol" of [17] has similar behavior although with a more complex message exchange.

3.2 Cursor Stability (CS)

Cursor stability can be used to avoid having to abort readers when writers arrive by using short read locks instead of long ones. The algorithm described in the previous section can be extended in a straightforward way to include short read locks. A detailed description can be found in [11].

3.3 Snapshot Isolation (SI)

Snapshot isolation, as provided in Oracle [13], uses the notion of object versions to provide individual snapshots. Older versions of an object X can be *reconstructed* by applying undo successively to X until the requested version is generated. Each object version is labeled with the transaction T that created the version. A transaction T reads the version of an object X labeled with the latest transaction which updated X and committed before T started. When T wants to write an object updated after T started, T will be

The lock manager of each node N coordinates the operation requests of the transactions as follows:

1. *A local transaction T_i makes a read request $r_i(X)$:* reconstruct the version of X labeled with T_j where T_j is the transaction with the highest $TS_j(EOT)$ so that $TS_j(EOT) \leq TS_i(BOT)$.
2. *Upon delivery of WS_i:* perform in an atomic step for each object X where $\exists w_i(X) \in WS_i$ a *version check*:
   i. If there is no other write lock on X and X is labeled with T_j: if $TS_j(EOT) > TS_i(BOT)$, then abort T_i. Otherwise grant the lock.
   ii. If there is a write lock on X or a write lock is waiting, and T_j will be the last transaction to modify X before T_i: if $TS_j(EOT) > TS_i(BOT)$, then abort T_i. Otherwise wait for the lock.
3. *Upon having executed all operations of T_i:* commit the transaction and release all locks.

Figure 2: Replication protocol guaranteeing snapshot isolation

aborted (*first writer wins* strategy). Timestamps are used to identify the begin (BOT) and end (EOT) of a transaction. To synchronize timestamps in a distributed environment we use the sequence numbers of WS messages as a global virtual time. The BOT timestamp $TS_i(BOT)$ of transaction T_i is set to the highest sequence number of a message WS_j so that transaction T_j and all transactions whose WS have lower sequence numbers than WS_j have committed. When a message WS_i is delivered $TS_i(EOT)$ is set to WS_i's sequence number (and therefore is unique).

Figure 2 describes the algorithm providing snapshot isolation. Each node can decide locally whether a transaction will commit or abort. No extra commit/abort message is necessary. Furthermore, read operations do not need to wait for write operations to finish and vice versa. Note that while serializability aborts readers when a conflicting write arrives, snapshot isolation aborts one of two concurrent writers. We can therefore surmise that, regarding the abort rate, the advantages of one or the other algorithm will depend on the ratio between read and write operations.

4 Atomic Delivery and Consistency

To guarantee consistency in a replicated environment, once a transaction is committed at one node, it must be committed at all nodes. In a failure free environment, this consistency is guaranteed by the total order. However, if failures may occur, practical solutions to this problem range from 2-Phase-Commit to reconciliation techniques [4, 7, 14], always under the assumption that, in systems with a large transaction volume, it is considered acceptable to introduce inconsistencies in a few transactions in exchange for being able to process the bulk of them as fast as possible.

We follow the same line and relax the atomicity of the broadcast primitives in order to minimize message and logging overhead leading to three versions of the previous protocols: non-blocking, blocking, and reconciliation based. Due to space limitations we only present the solutions for serializability and cursor stability. For snapshot isolation the approach is similar.

4.1 Node Failures and Recovery in Databases

We assume that in the case of node or communication failures only a primary group is able to continue. Once a node recovers, it is allowed to join the group only after its state is identical to that of the available nodes (achieved by, for instance, installing a copy of the database taken from one of the working nodes [4]).

If a node fails there are two cases to consider. First, that in which a node commits a transaction and then fails while the other nodes decide to abort. In this case, upon recovery, reconciliation or compensation techniques can be used to undo the changes of the committed transaction as it is done in, for instance, Oracle Symmetric Replication [14]. Due to the complexities involved, this corresponds to the lowest level of fault-tolerance. In the second case, the failed node has not yet committed a transaction while the rest of the system commits it. This, in general, is not a problem since upon recovery the failed node can incorporate the committed transaction. Hence, protocols allowing this case will still be considered as being fault-tolerant.

In our replicated scenario, when node N fails and is excluded from the group, each surviving node N' has to decide on the *in-doubt* transactions of N. A transaction T_i invoked at node N is in-doubt for N' if N' has delivered WS_i but neither a_i nor c_i have been delivered.

4.2 Non-Blocking Coordination

The non-blocking versions of the protocols are based on the following atomic delivery properties:

The delivery of both the write set WS_i and the commit message c_i is atomic.

If both message types are sent atomically, each node can decide independently to abort in-doubt transactions of a failed node N. Due to the atomicity properties, the group change that excludes N is not announced to the application of N' before all messages N might have delivered are also delivered at N'. Since

c_i was not delivered before the group change, no other node (including N) has delivered or will deliver c_i. Furthermore, the atomic delivery of WS_i excludes the scenario where all nodes of the remaining group have received c_i but none of them has received WS_i. Note that abort messages, on the other hand, need not to be sent atomically. This approach guarantees that if surviving nodes decide to abort a transaction, no failed node has committed it.

As long as the underlying group communication system is non-blocking, the database will also not block when a failure occurs.

4.3 Blocking Coordination

Some of the overhead of the previous approach can be avoided by risking not being able to reach a decision about the transactions of a failed node. The atomic delivery requirement for this case is:

The delivery of a write set WS_i is atomic.

The atomic delivery of WS_i is necessary to exclude the case where a node delivers WS_i and c_i, commits T_i, and fails, while the remaining nodes have not even received WS_i and will therefore ignore T_i.

Since the delivery of both c_i and a_i is non-atomic, now a node cannot decide independently on an in-doubt transaction T_i. It might be that other nodes (including the failed one) have either delivered c_i or a_i and terminated T_i, or are also in-doubt. Therefore, a coordination protocol among the members of the new group is needed. If a transaction T_i is in-doubt at all nodes, T_i must be blocked until the recovery of N, because N might have aborted or committed T_i and reported the result to the user. However, if a transaction T_i is in-doubt at node N' but not at N'', N'' should inform N'.

After a group change a coordinator node, C, sends a decision request message req with all its in-doubt transactions to the new group. Upon delivery of req, a participant P sends the following response message res back to C:

- For each T_i cited in req: if P has already received c_i/a_i, it includes it in res. (So that C can terminate its in-doubt transaction T_i.)
- If P has an in-doubt transaction T_i not cited in req, T_i is included in res. (So that C, that must have received c_i/a_i, informs P.)

After the delivery of all res, C knows about the state of all group members. A transaction, T_i, cited in req but whose c_i/a_i was not included in any res, is in-doubt at all nodes and must be blocked. For all other transactions, C can now make a decision. It atomically broadcasts this information to the new group. After the delivery of this broadcast each node can continue normal processing. In the case of a failure of C before the delivery of the last broadcast the coordination can be simply restarted by a new coordinator.

Note, that only transactions conflicting with the blocked transactions cannot be executed. This same protocol can be used in a slightly different manner. The coordinator decides to abort the in-doubt transactions and later let the failed node use compensation to undo the changes in case it had committed them. However, if these inconsistencies are allowed the following approach could be used, with less overhead.

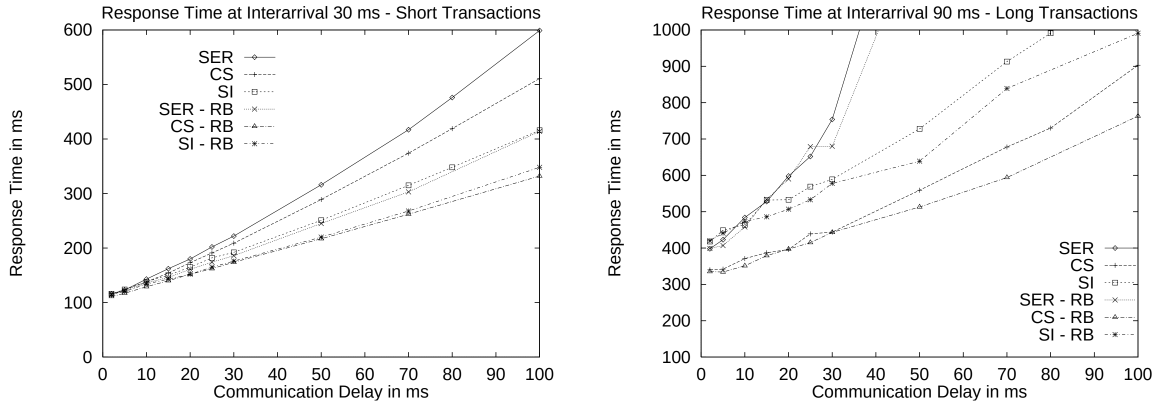
4.4 Reconciliation based Protocols

Reconciliation based coordination does not broadcast any message atomically. The result is, as described before, that the failed node N may have committed T_i but the rest of the nodes have not even received WS_i and will ignore T_i, or will decide to abort T_i. Upon recovery, N needs to reconciliate its database with that of the working nodes and compensate the changes introduced by T_i. Following standard practice in database systems, we consider the probability of such an event to be small enough to make this protocol a viable alternative when performance needs to be improved.

5 Experiments and Results

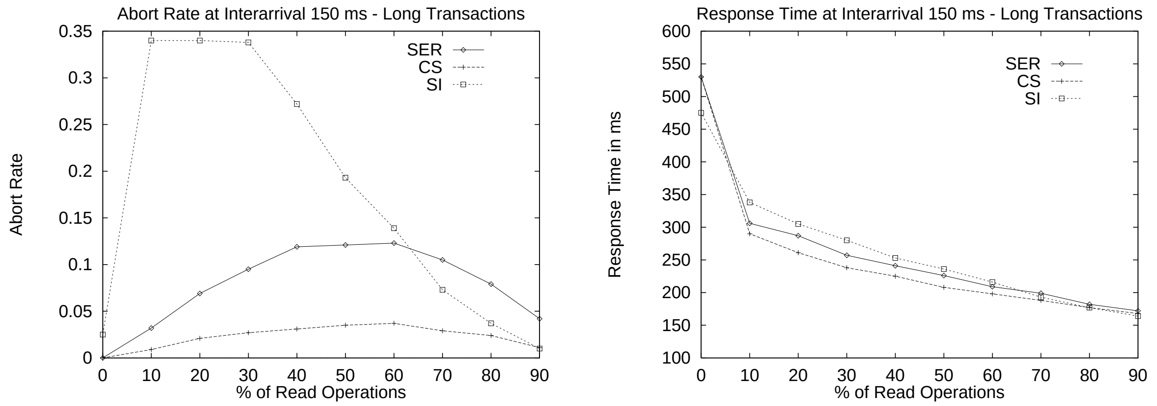
The performance of the replication protocols described above has been studied using a simulation study similar to [2]. A detailed description of the model and the parameters can be found in [11]. Our system carefully models the use of hardware resources like CPU, disk and the network. Communication costs are determined as a function of the communication semantics. Every broadcast has a *basic communication delay*. If a message is sent either in total order or atomically, this time is multiplied by 2. If it is sent both in total order and atomically this time is multiplied by 2.5. This approach models a rough estimation of the overhead of the different broadcast semantics. Transaction execution and concurrency control is modeled according to the algorithms described in the previous sections. We use an open queuing model. At each node every certain time unit (inter arrival time) a new transaction is started.

Due to space limitations we will only discuss some of the results of the many experiments we have conducted. The presented experiments base on a 10-node-system and a database with 10000 objects. All tests were repeated until a 90% confidence interval was reached. The main performance metric is the response time of a transaction, i.e., the time from BOT to EOT. It consists of the execution time (CPU + I/O) and the message delay. The following abbreviations are used:



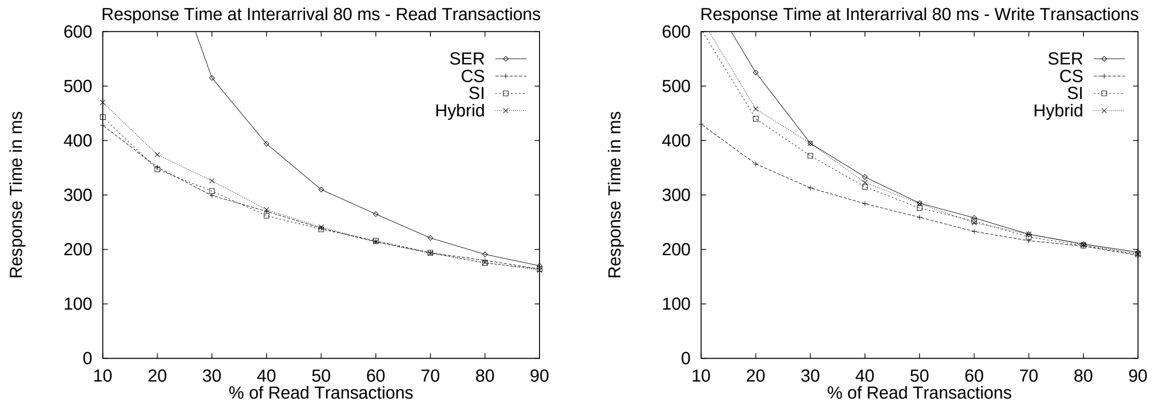
(a) (b)

Figure 3: Response time of (a) short transactions and (b) long transactions for different communication delays



(a) (b)

Figure 4: (a) Abort rate and (b) response time of long transactions for different read/update rates



(a) (b)

Figure 5: Response time of (a) read-only transactions and (b) update transactions for different rates of read-only transactions

SER for serializability, CS for cursor stability, and SI for snapshot isolation. Only the results for the reconciliation based (RB) and non-blocking versions of the algorithms are compared.

5.1 Communication Delay

Existing studies on group communication systems present considerable different results [5, 12], thus our first test suite investigates the impact of the communication delay. We have performed tests with two different transaction types (*short* transactions with 10 operations and *long* transactions with 30 operations, both having 40% of writes) to analyze different data contention workloads. The parameter for the basic communication delay varies from 2 to 100 ms.

Figure 3 shows the response time for short and long transactions, respectively. The arrival times are chosen so that the resource contention is similar and rather high for both workloads. Short transactions show little data contention, and therefore, the message delay has a great impact on the response time. With a low communication delay the response time is purely due to the execution time which is roughly similar for all protocols. However, with increasing message delay, the response times increase depending on the number and complexity of the message exchanges. The non-blocking protocols show worse performance. In the case of long transactions, the response time is much more influenced by the data contention. Since the resource utilization is high, the abort and restart of a transaction degrades performance more than long message delays. Both CS protocols clearly show the best results for all message delays because of their low conflict rate. SER, on the other hand, has such a high data contention that it starts thrashing very quickly. This degradation is due to readers aborted upon arrival of a write transaction which is later also aborted. Aborting the readers was unnecessary, but, as the message delay increases, the likelihood of such cases increases. SI does not have this problem because the decision to abort or commit is done independently at each node and the write set of a transaction T is only executed when T is able to commit.

As a summary, the choice of the best protocol depends on the workload and the system configuration. Low conflict rates or fast communication permits to use serializability and atomic message delivery to provide full correctness and consistency. However, slow communication forces to choose protocols with low message overhead like snapshot isolation or reconciliation based protocols. If, on the other hand, data contention is high, lower levels of isolation are the only alternative to achieve acceptable response times.

5.2 Resource and Data Contention

The distribution of read and write operations is relevant in two ways. First, depending on the replica control method, conflict resolution is either done by aborting readers or writers. Second, the distribution influences the resource contention, since read operations are only performed at one node and write operations at all nodes. Thus, in this experiment, we consider transactions with 30 operations and the percentage of write operations varied from 10% to 100%. The basic communication delay is set to be very low (5 ms) because we want to investigate pure resource and data contention. Since the level of atomicity does not have an impact at so low communication costs we only show the results for the non-blocking protocols.

Figure 4 shows the abort rate and the response time as a function of the read rate. CS always has the smallest abort rate. For SER and CS the abort rates are zero with only write operations, since writes are never aborted. The abort rates increase slightly because reads might be aborted by writes but then decreases because there are less writes in the system. SI has a high abort rate when the update rate is high because it solves write/write conflicts by aborting one transaction, but then the abort rate decreases fast because SI does not abort reads. This shows that SI is especially indicated for high read rates. The abort rate, however, has only a small impact on the response time, which is mainly determined by the resource utilization, and hence, similar for all protocols. At low read rates the system is near its thrashing point because updates are performed at all nodes. However, with increasing read rates, the resource utilization and with it the response time decreases substantially because reads are only done locally.

In the configuration studied, resource contention is a more relevant factor than data contention if the update rates are high. This problem is inherent to replication schemes where all updates are performed at all copies. However, update rates in practical applications do not seem to be extremely high: even in pure OLTP workloads like the TPC-C benchmark, transactions normally have at least 50% read operations.

5.3 Read-Only Transactions

The main gain of replication lies in the ability to perform read operations locally. Replication pays off when a great part of the transactions are queries which can be executed without any communication costs. As a last experiment we look at a mixed workload consisting of update and read-only transactions, both having 30 operations. The percentage of both types are varied between 10% to 90%. Again, we chose a fast communication and only look at the non-blocking protocols. This time, we also analyze the hybrid protocol.

Figure 5 presents the response times for the read-only transactions and update transactions as a function of the percentage of read-only transactions. The response times of both transaction types decrease when the percentage of read-only transactions increases because of less resource and data contention. Although the performance is mainly determined by the resource utilization, differences in the protocols can be observed. They are due to different abort rates. Only SER aborts read-only transactions leading to high response times. If many updaters are in the system CS behaves better than the others for update transactions. However, the abort rates – and with them the performance differences – decrease very fast with an increasing number of read-only transactions.

As a conclusion, read-only transactions need a special treatment to avoid unnecessary aborts. The hybrid protocol seems to be a good alternative to provide good performance for read-only transactions and serializability for updating transactions. However, transactions must be declared read-only in advance to allow their special treatment. Although cursor stability has the best performance results it may be problematic in certain applications due to its low isolation level.

6 Conclusion

In this paper, we have analyzed several approaches to maintain a replicated and distributed database by using group communication. Starting with a fault-tolerant, 2PL protocol presented in [1], we suggested several optimizations to handle data contention and to achieve high performance even in the case of slow communication systems. To do so, we weakened the correctness and consistency criteria as it is typically done in existing database systems. We quantitatively investigated the performance implications of the different methods under various workload configurations, using a detailed simulation model.

Our experiments demonstrate several points: The efficiency of communication plays a major role in replicated transaction processing. However, severe performance problems can be avoided by using protocols that reduce the message overhead but weaken fault-tolerance. We suggest protocols that guarantee consistency, however allow the user to see incorrect data in the rare cases of node failures. With our protocols, the number of messages per transaction is constant. However, long message delays increase the probability of data contention because the response times of the transactions are longer, leading to more concurrent transactions in the system. This problem can be solved by using concurrency control protocols with lower isolation levels. These lower isolation levels also help to reduce the conflict rate among transactions.

We believe synchronous update everywhere replication is feasible for a wide spectrum of applications and configurations. With fast communication, a low system load, low conflict rates or a high percentage of read-only transactions, standard 2PL can be used. If the system configuration is not ideal, as it will happen in most cases, the optimizations described in the paper help to maintain reasonable performance while still guaranteeing consistency and replication transparency to a high degree.

Acknowledgments

We would like to thank A. Schiper, R. Guerraoui and F. Pedone for helpful discussions on the topic.

References

[1] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Euro-Par'97*, Passau (Germany), August 1997.

[2] R. Agrawal, M.J. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Trans. on Database Systems*, 12(4):609–654, 1987.

[3] G. Alonso. Partial database replication and group communication primitives. In *2nd Europ. Research Seminar on Advances in Distr. Systems (ERSADS'97)*, Zinal (Switzerland), March 1997.

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

[5] K. Birman and T. Clark. Performance of the Isis distributed computing toolkit. Technical report, Dep. of Computer Science, Cornell University, TR-94-1432, June 1994.

[6] David Powell et al. Group communication (special issue). *Communications of the ACM*, 39(4):50–97, April 1996.

[7] R. Goldring. A discussion of relational database replication technology. *InfoDB*, 8(1), 1994.

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

[9] J. Gray and A. Reuter. *Transaction Processing.* Morgan Kaufmann, 1993.

[10] V. Hadzilacos and S. Toueg. *Distributed Systems*, chapter 5, pages 97–145. Addison-Wesley, 1993. Edited by S. Mullender.

[11] B. Kemme and G. Alonso. Database replication based on group communication. Technical report, Department of Computer Science, ETH Zürich, No. 289, February 1998.

[12] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[13] Oracle. *Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7*, 1995. White Paper.

[14] Oracle. *Oracle8(TM) Server Replication, Concepts Manual*, 1997.

[15] F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. In *16th IEEE Symp. on Reliable Distributed Systems (SRDS'97)*, Durham, USA, October 1997.

[16] A. Schiper and A. Sandoz. Uniform reliable multicast in a virtually synchronous environment. In *13th IEEE Int. Conf. on Distributed Computing Systems*, Pittsburgh, USA, 1993.

[17] I. Stanoi, D. Agrawal, and A. El Abbadi. Using broadcast primitives in replicated databases. In *18th IEEE Int. Conf. on Distributed Computing Systems*, Amsterdam, The Netherlands, May 1998.