Database Replication based on Group Communication

Technical Report No. 289
ETH Zürich, Departement of Computer Science

Bettina Kemme   Gustavo Alonso
ETH Zürich
Institut für Informationssysteme
ETH Zentrum, CH-8092 Zürich
E-mail: {kemme,alonso}@inf.ethz.ch

February 5, 1998

Abstract

Database replication is traditionally seen as a mechanism to increase the availability and performance of distributed databases. However, very few of the numerous protocols that have been proposed in the literature are used in current systems. There is a strong belief among database developers that existing solutions introduce too much overhead to be feasible in practice. Instead, current products allow inconsistencies and often resort to centralized approaches, which eliminates some of the advantages of replication. To resolve this issue, this paper proposes a suite of replication protocols based on group communication primitives that tries to address the concerns of database designers. These protocols take advantage of the rich semantics of group communication primitives and the relaxed isolation guarantees provided by most databases to eliminate the possibility of deadlocks, reduce the message overhead, and increase performance. A detailed simulation study shows the feasibility of the approach and the flexibility with which different types of bottlenecks can be circumvented.

1 Introduction

Database replication has been traditionally used as a basic mechanism to increase the availability (by allowing *fail-over* configurations) and the performance (by eliminating the need to access remote sites) of distributed databases [BHG87]. A large number of protocols exist providing data consistency and fault-tolerance [BHG87]. Paradoxically, few of these ideas have ever been used in commercial products. There is a strong belief among database designers that most of the proposed solutions are not feasible due to performance and scalability problems. Instead, current products adopt a very pragmatic approach: copies are not kept consistent, updates are often centralized, and solving update inconsistencies is left to the user [Sta94]. While this approach may be criticized, some of the arguments against existing research solutions are justified [GHOS96], especially from the point of view of commercial products where performance often takes precedence over any other consideration.

This paper attempts to bridge the gap between replication theory and practice by addressing some of the concerns of database designers. Unlike previous replication protocols, the solution proposed is tightly integrated with the underlying communication system. Following some initial work done in this direction [SR96, AAES97, Alo97, PGS97], the idea is to exploit the semantics of *group communication primitives* [BSS91, CT91, MMSA$^+$96, DM96, vRBM96] in order to minimize the overhead of replication. Group communication systems manage the message exchange between a group of nodes by providing group maintenance, reliable message exchange, and message ordering primitives. The fault-tolerance and consistency services they provide will be used to perform some of the tasks of the database. The objective is to push down in the software hierarchy the more basic functions, thereby avoiding some of the performance limitations of current solutions. Moreover, in order to provide a realistic solution, the proposed protocols take into consideration the fact that databases usually provide a range of consistency levels, commonly much less restrictive than what is considered in theory. These consistency levels are included in the replication protocols to avoid overhead when the application does not demand it. Thus, the contribution of the paper is twofold. First, it presents a family of replication protocols based on group communication which provides different levels of isolation and fault-tolerance. We believe these protocols can be easily integrated in current systems, providing reasonable performance, the same transactional semantics found in centralized systems, and maintaining data consistency and replication transparency. Second, the problems of implementing replication efficiently is explored in significant detail by means of a detailed simulation study showing the feasibility of the proposed solutions and how these protocols can be implemented within a database system.

The paper is organized as follows: Section 2 motivates the approach by analyzing the problems related to replication. Section 3 describes the system model and its components. Section 4 presents a family of protocols which provides different levels of isolation and Section 5 redefines the algorithms to provide different levels of fault-tolerance. Section 6 provides an overview of the simulation system and the parameters. Section 7 describes the experiments we have conducted and their results. Section 8 summarizes the results.

2 Motivation

Most of the work done in database replication is based on *1-copy-serializability* [BHG87]: 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. These protocols are often *synchronous* (a transaction must update all or a quorum of copies before it commits) and *update everywhere* (a transaction can update any copy in the system regardless of where it is located). Recently, it has been pointed out that this approach is somewhat "dangerous" [GHOS96]:

- The probability of single-object deadlocks is $O(n^3)$, where n is the number of replicas. This severely limits scalability, since as more replicas are added, the probability of transactions creating a deadlock will become too high for the system to be able to function. This problem is further complicated by the overhead of having to perform distributed deadlock detection [BHG87, Kna87].
- Serializability as correctness criteria for transactional consistency is often too restrictive. Current database systems acknowledge this by providing several levels of isolation, however, these solutions have not yet been integrated in replicated systems.
- The message and logging overhead associated to replication protocols is too high to be used in practice, often leading to resource contention and high transaction response

times [BHG87, ET89].

These considerations have lead to traditional solutions not being used in practice. Current products simply ignore both synchronous replication and update everywhere semantics and instead provide *asynchronous* and *primary copy* replication. Asynchronous protocols propagate the updates of a transaction only after the transaction has committed, which decreases the response time but introduces data inconsistencies. Primary copy approaches centralize updates in a single copy to avoid concurrent updates to different copies, creating a single point of failure. Existing systems implement these ideas in different ways [Sta94, Gol94]. For instance, *Sybase Replication Server* propagates updates on the primary copy as soon as the transaction commits (*push* strategy) in an effort to minimize the time that the copies are inconsistent (an implicit acknowledgment of the importance of keeping copies consistent). *IBM Data Propagator*, on the other hand, adopts a *pull* strategy in which updates are propagated only at the client request (a client will not see its own update unless it requests it from the central copy). *Oracle Symmetric Replication* [Ora97] supports both push and pull strategies, as well as synchronous and asynchronous replication. In the case of asynchronous replication a best effort approach is taken in order to try to minimize the time that copies are inconsistent. Oracle also allows asynchronous, update everywhere replication. This however creates the non-trivial problem of reconciliation: conflict resolution can be done automatically using some of the mechanisms Oracle provides (highest timestamp, for instance) but it is left to the user to decide which one is the correct copy.

In view of this gap between theory and practice, the question that needs to be addressed is whether it is possible to design feasible, synchronous, update everywhere protocols that do not suffer from the drawbacks outlined above. We believe that the answer to this question lies in a tighter integration between the transaction management and the underlying communications in order to exploit the advantages of both systems and eliminate the redundancies that result in critical limitations. In the following, we show how this can be achieved.

3 Model

A distributed database consists of a number of nodes, $N_i, (0 < i \leq n)$, each of which is a collection of objects which can be accessed by executing transactions. Nodes communicate with each other by exchanging messages. We assume a fully replicated system, i.e., all nodes contain copies of the same objects and there is a copy of each object in every node. Figure 1 shows the main components of a node [BHG87, GR93]: communication system, transaction manager, and concurrency control, which are also responsible for the replica control.

3.1 Communication Model

Communications are based on group communication primitives [BSS91, CT91, MMSA$^+$96, DM96, vRBM96] according to the following notation. A node N *broadcasts/sends* a message to all nodes of a group. A message is *received* at a node N when it is physically stored at N. After the reception, the message is *delivered* to the application when the order is determined and the delivery guarantees are fulfilled.

Of the many characteristics of group communication systems we are particularly interested in the message ordering and delivery guarantees. In regard to message *ordering*, several different types of ordering are commonly provided. Of these, we are interested in two:

- The *basic* service which does not guarantee any ordering of the stream of received messages.

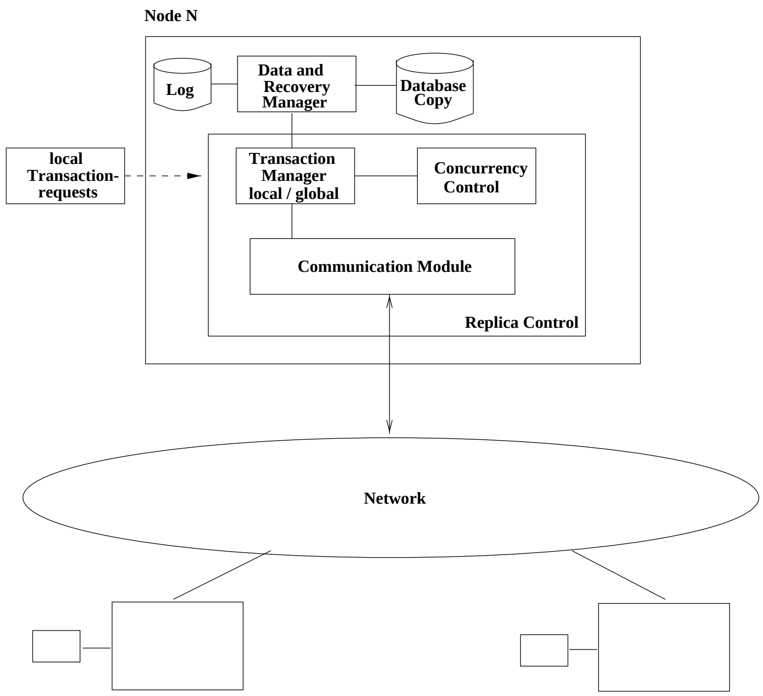


Figure 1: Architecture of a node in a replicated database system

- The *total* order, which delivers messages in the same order at all nodes.

Regarding delivery guarantees, most existing protocols handle message loss guaranteeing that a message broadcast by an available node will eventually be received by all available nodes. Considering fail-stop behavior of the nodes [SS83], two different services can be distinguished:

- The *non-atomic* (*non-reliable*) service, which delivers a message as soon as all preceding messages have been delivered.
- The *atomic* (*reliable*) service, which delivers a message m when it is guaranteed that all available nodes will deliver m.

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.

3.2 Transaction Model

Users interact with the database by invoking transactions. A transaction T_i is a partial order, $<_i$, of read $r_i(X)$ and write $w_i(X)$ operations on objects(X). Transactions are executed atomically, i.e., a transaction either commits, c_i, or aborts, a_i, the results of all its operations [BHG87]. Two operations conflict if they are from different transactions, access the same object and at least one of them is a write. A history H is used to describe the order between operations of different transactions. A history H is a partial order, $<_H$, of the operations of a set of transactions such that $\forall T_i \in H : <_i \subseteq <_H$. Furthermore, all conflicting operations contained in H must be ordered. 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 [GR93]. 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. The highest isolation level *serializability*, requires a history to be equivalent to a serial history. Lower levels of isolation are less restrictive but allow inconsistencies. Following [BBG+95], these inconsistencies can be defined in terms of several phenomena. The ones of interest for the purposes of this paper are:

- *Dirty read*, $P1 : w_1(X) \ldots r_2(X) \ldots (c_1 \text{ or } a1)$. T_2 reads an uncommitted version of X.
- *Lost Update*, $P2 : r_1(X) \ldots w_2(X) \ldots w_1(X) \ldots c_1$. T_1 might write X upon the result of its read operation not considering the new version of X created by T_2. T_2's update is lost.
- *Non-Repeatable Read*, $P3 : r_1(X) \ldots w_2(X) \ldots c_2 \ldots r_1(X) \ldots c_1$. T_1 reads two different versions of X.
- *Read Skew*, $P4 : r_1(X) \ldots w_2(X) \ldots w_2(Y) \ldots c_2 \ldots r_1(Y)$. If there exists a constraint between X and Y, T_1 might read versions of X and Y that do not fulfill this constraint.
- *Write Skew*, $P5 : r_1(X) \ldots r_2(Y) \ldots w_1(Y) \ldots w_2(X)$. If there exists a constraint between X and Y it might be violated by the two writes.

3.3 Concurrency Control Model

In most systems, locking protocols are used to implement the different isolation levels. Since write locks can not be released until commit time for a variety of reasons (mainly recovery and lost updates) 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. The different protocols used in this paper are:

Serializability is implemented using strict 2-phase-locking [BHG87]. Before accessing an object, a transaction acquires a read or a write lock on the object. Locks are held until the end of transaction (called long locks). There may not be two locks for conflicting operations active on an object at the same time. Strict 2-Phase-Locking provides serializability because it avoids all phenomena described before. The long write-locks avoid $P1$, the long read-locks avoid $P2 - P5$.

Cursor Stability avoids long delays of writers due to conflicting read operations [GR93]. Transactions often scan the database looking for particular records. Most of their read operations have no bearing on the results of the transaction and, thus, holding long read locks is certainly too restrictive. To address this issue, commercial databases provide cursor stability. Write-locks are still long, avoiding $P1$. Read locks, on the other hand, are short in most cases. They are released after the item has been read. However, read locks are long on objects the transaction wants to update. Hence, $P2$ is avoided. Cursor stability is not serializable because phenomena $P3 - P5$ may occur.

Snapshot Isolation is a way to eliminate read locks completely, thereby providing more concurrency and decreasing transaction response times [BBG+95, Ora95]. Each transaction T reads data from a snapshot of the database as of the time the transaction started and therefore always sees a consistent view of the database. Its own updates will be integrated in the snapshot. When T wants to write an object that has been updated since T started, T will be aborted. This feature is called *first committer wins*. Snapshot isolation avoids $P1$ because a transaction only reads committed data. $P2$ is avoided because T_1 will be aborted when it tries to update X (T_2 modified X after the start of T_1). Also $P3$ and $P4$ are avoided because T_1 will not read the versions that were created by T_2 but the versions of its snapshot.

However, $P5$ might arise because T_1 and T_2 write different data items. Note that read-only transactions see a consistent view of the database and are serializable. Snapshot isolation is provided by Oracle since version 7.3 [Ora95].

Hybrid Protocol combines 2-phase-locking and snapshot isolation by requiring that update transactions acquire long read and write locks while read-only transactions are provided with a snapshot of the database. This *hybrid* protocols provides full serializability but requires to distinguish between update transactions and queries.

3.4 Replica Control Model

In a replicated database the existence of multiple copies of an object should appear as a single logical copy. This is termed as *one-copy equivalence* [BHG87]. To achieve one-copy equivalence a version of the *all available copies* approach [BHG87] is used. A transaction T_i can be invoked at any node N in the system. T_i is called *local* at N. For all 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 all write operations, WO_i, is bundled into a single message and propagated to all available nodes (including the local node).

Following previous work [AAES97] we exploit the total order provided by the communication system in order to decide on the order of conflicting transactions. This concept of ordering is at the core of the replication mechanism. Upon receiving the write set WO_i of a transaction T_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 WO_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. We will see in section 4 that the processing of write sets is only a small fraction of the entire transaction execution. Note that with this approach deadlocks because of write-write conflicts are avoided entirely. Since the write sets are received at all nodes in the same order, single object deadlocks cannot occur. Furthermore, deadlocks involving two or more objects are avoided because the write requests of a transaction are processed in a single step. Thus, one of the main concerns regarding replication protocols is avoided by relying in the communication system to provide some ordering on the locking requests.

4 Database Replication Protocols

In what follows we present three replication protocols designed to avoid two of the drawbacks of synchronous replication. First, they avoid not only single-object deadlocks as described in the previous section but deadlocks in general. Second, the protocols provide different levels of isolation to be able to capture the varying requirements of different applications.

4.1 Replication with Serializability (SER)

Figure 2 describes the replicated version of the strict 2-Phase Locking protocol. It is based on one of the protocols proposed by Agrawal et al. in [AAES97]. The basic idea is to postpone writes and broadcast all updates at the end of the transaction. Upon delivery of such a message (totally ordered by the underlying communication mechanism) the transaction manager acquires all the necessary locks (aborting any conflicting readers, see below), thereby

The lock manager of each node N controls and coordinates the operation requests of the transactions in the following manner:

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 and submit the execution of the operation, 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:
   bundle all write requests to a write set WO_i and broadcast it with the total order semantics of the communication system.
4. On delivering WO_i, process it in an atomic step:
   a. Perform for each object X where $\exists w_i(X) \in WO_i$:
      i. If there is a granted read lock $r_j(X)$ and the write set WO_j of T_j has not yet been received, abort T_j. If WO_j has already been sent, then broadcast a_j. There are no ordering requirements on abort messages. Later N itself will ignore both the WO_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 received, 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 and submit the execution of the operation.

   b. if T_i is a local transaction, broadcast c_i. There are no ordering requirements on commit messages.
5. On delivering c_i:
   wait until all operations of T_i have been executed, then commit T_i and release all locks.
6. On delivering a_i:
   undo all operations already executed and release all locks (granted and waiting ones).
7. On releasing a lock:
   look whether another transaction is waiting for the lock and grant it, handle write before read locks.

Figure 2: Protocol description: serializability

guaranteeing that conflicting transactions are executed in the same order at all sites. The proof of the correctness of the algorithm can be found in [AAES97].

In this protocol, the execution of a transaction T_i requires two messages. One for the write set and another with the decision to abort or commit, since only the owner of T_i knows about the read operations of T_i. To avoid deadlocks in the case of read-write conflicts the algorithm aborts any conflicting read operation. Since read operations are only known locally but write operations are executed globally, it is necessary to give write operations priority over read operations, hence aborting readers when conflicting write operations arrive. Note also that write requests are given preference before read requests (step 7). If a read lock is granted when a write request is waiting, the reader would be aborted immediately.

4.2 Replication with 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 if the transaction does not want to write the object afterwards. How far the abort rate can be reduced depends strongly on the transaction type. The algorithm described in the previous section can be extended in a straightforward way to include short read locks. Figure 3 shows the steps that need to be changed.

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 and submit the execution of the operation, else wait until the lock can be granted. If the request is a short lock, release the lock once the operation has been performed.

4.a.iii If there is another write lock on X or all read locks on X are either short or from transactions whose write sets have already been received, then wait until all other locks are released and the new lock can be granted.

7. On releasing a lock:
   look whether another transaction is waiting for the lock and grant it, handle write before long read locks.

Figure 3: Protocol changes: cursor stability

4.3 Replication with Snapshot Isolation (SI)

Snapshot isolation uses the notion of timestamps to identify the begin (BOT) and end (EOT) of a transactions. To determine these timestamps we use the following approach:

- The sequence numbers of WO messages are used to timestamp transactions. Since WO messages are totally ordered, these sequence numbers can be easily generated.
- The BOT timestamp $TS_i(BOT)$ is set to the highest sequence number of a message WO_j so that transaction T_j and all transactions whose WO have lower sequence numbers have committed. (Note that the BOT timestamp of different transactions does not need to be unique since it is used solely to determine the snapshot they will access).
- The EOT timestamp $TS_i(EOT)$ is the sequence number of the WO_i message (and therefore is unique).

Furthermore, snapshot isolation works with object versions. An object X is labeled with a transaction T_i if T_i was the last transaction to update X. Figure 4 describes our algorithm for replication with snapshot isolation. The algorithm is very similar to the implementation of snapshot isolation in Oracle [Ora95]. The only significant difference lies in the way EOT timestamps are created. Oracle uses a counter of committed transactions. We use the order of the delivery of write sets to guarantee that all copies of an object are labeled in the same way. Therefore, on delivery of WO_i, all nodes will make the same decision whether to commit or abort T_i.

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 X is labeled with a transaction T_j where $TS_j(EOT) > TS_i(BOT)$ then *reconstruct* a copy of the highest version X_k that was created by a transaction T_k where $TS_k(EOT) \leq TS_i(BOT)$. This can be done by applying undo successively until the requested version is generated.
2. On the delivery of WO_i, process WO_i in an atomic step:
   a. Perform for each object X where $\exists w_i(X) \in WO_i$ 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 stop checking locks and abort T_i. Otherwise grant the lock.
      ii. If there is another write lock on X or some other 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 stop checking locks and abort T_i. Otherwise wait for the lock.
3. On having executed a write operation of T_i, the lock can be released.
4. On having executed all operations of T_i, commit the transaction.
5. On releasing a lock, look whether another transaction is waiting for the lock and grant it.

Figure 4: Protocol description: snapshot isolation

With this algorithm, each node can decide locally whether a transaction will commit or abort. No extra commit/abort message is necessary. T_i might do a preliminary check as described in step 2 before sending WO_i to see whether there already exists a write conflict with another transaction. In this case it can abort locally and avoid sending any messages. However, the check must be repeated upon reception of WO_i. Locks need not to be held until EOT because write operations are only performed if the transaction will commit. Note that while serializability aborts readers when a concurrent write arrives, snapshot isolation aborts one of two concurrent writers. We can therefore assume that, regarding the abort rate, the advantages of one or the other algorithm will depend on the ratio between read and write operations. Furthermore, with snapshot isolation, read operations do not need to wait for write operations to finish and vice versa. Furthermore, snapshot isolation needs one message less per transaction than serializability and cursor stability.

5 Atomic Broadcast 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, the protocols are confronted with the consensus problem [FLP85]. In practice, existing databases use 2-Phase-Commit (2PC) to reach consensus among the nodes although it may force transactions to block in the case of node failures [BHG87]. In replicated scenarios, databases allow a transaction to commit locally even if its changes have not been propagated to other sites. The available copies will remain inconsistent and conflicting updates will later be solved using reconciliation techniques [Sta94, Gol94, Ora97]. As with the different isolation levels provided, this is a pragmatic solution. In systems with a large transaction volume, it is considered acceptable to introduce inconsistencies in a few transactions in case of failures in exchange for being able to process the bulk of them as fast as possible.

In the previous section, we took advantage of the levels of isolation provided by databases to relax the consistency guarantees of the replication protocols. The same approach can be applied in the case of the atomicity requirements of the broadcast primitives, which can be relaxed in order to minimize message and logging overhead. As before, this relaxation is based on what is accepted practice in commercial database products. The result of this relaxation is three versions of the previous protocols (non-blocking, blocking, and reconciliation based) depending on the different atomicity guarantees provided by group communication. Due to space limitations we only present the solutions for serializability and cursor stability. For snapshot isolation the approach is similar.

5.1 Node Failures and Recovery in Databases

We will focus on *node failures*. We assume that nodes are *fail-stop* [SS83]. We will assume in here that a node has recovered only after its state is identical to that of the other working nodes. There are a number of recovery procedures that can be followed [BHG87, GR93], but in the case of replicated databases, the simplest one is to install a copy of the database taken from one of the other working nodes.

Unlike in the general case of the consensus problem, in replicated databases the problem is simplified by the existence of the database log, which allows to undo and replay the transactional history. There are two cases to consider. First, that in which a node commits a transaction, fails, and the other nodes decide to abort that same transaction. In this case, upon recovery, reconciliation [Ora97] or compensation [KLS90] techniques have to be used to undo the changes of the committed transaction. Although these techniques are not trivial to use, commercial systems, for instance Oracle Symmetric Replication [Ora97], provide several procedures for reconciliating inconsistent databases on a transaction per transaction basis. Due to the complexities involved in reconciliation and compensation, this is the most undesirable case, and therefore, corresponds to the lowest level of fault tolerance. The second case is that in which the failed node has aborted a transaction while the rest of the system decides to commit the transaction. This is, in general, not a problem since upon recovery the failed node will incorporate the committed transaction. To understand why this works, one has to consider that this case can only arise if the node fails while the transaction is still active (all active transactions are first aborted in the recovery procedure, even if later on they are re-executed when the node is brought up to date). Since the transaction was still active, it cannot have had any external effects. Since this second case can be easily dealt with, protocols allowing this case will still be consider as being fault-tolerant.

Moreover, and helping to simplify the problem even further, commercial database systems provide mechanisms to detect the failure of a node and propagate this information immediately to other nodes. These mechanisms involve automatic and manual procedures since recovering a database is not necessarily trivial but, for the purposes of this paper, it suffices to assume that node failures are promptly detected and notified to all other nodes in the system. In our replicated scenario, when a node fails, all other nodes must find out which transactions from the failed node are still active and then agree on what to do with them. This is done by a *coordination* protocol.

5.2 Non-Blocking Protocols

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

1. The delivery of a write set WO_i is atomic.
2. The delivery of a commit message c_i is atomic.

Note that there is no need to send an abort message atomically because if no node receives commit, they will decide to abort.

When node N fails, each surviving node N' has to decide on the active transactions of N. A transaction T_i of node N is active for N' if N' has received a message related to T_i but neither a_i nor c_i have been delivered. When the failure of N is detected a coordinator node C initiates the coordination protocol and decides on each of the active transactions of N. The decision depends on the knowledge the coordinator has about T_i:

1. C does not know about the outcome of T_i (it has received or even delivered WO_i, but it has not yet received c_i or a_i): C decides to abort T_i. This is possible due to requirement 2, which guarantees that if C has not received c_i then nobody has committed T_i. All available nodes can abort T_i and N will abort T_i upon recovery.
2. C knows that T_i is able to commit (it has received and delivered WO_i and received c_i): C decides to commit T_i. Since C has received c_i, everybody has received WO_i and nobody has received a_i. Therefore, all available nodes can commit T_i. It is unknown whether N has committed T_i. If it could not commit the transaction because of the failure, the transaction will be aborted upon recovery but it will then be re-executed when the changes from other databases are installed.

The coordinator sends all the decisions in a single *decision message* atomically to the other nodes. This means that the coordinator will not perform its own decisions before the other nodes have received the message. If the coordinator fails before the message can be delivered, the surviving nodes can select another coordinator and repeat the procedure.

Upon delivery of the decision message, a node N' decides on a transaction T_i as follows:

1. T_i was included in the decision message. N' follows the decision of the coordinator.
2. T_i was not included in the decision message, i. e., it was not active at the coordinator. Two cases must be distinguished:
   a.) N' has already received c_i. In this case C had already received WO_i due to requirement 1. If C does not mention T_i, it must already terminated it, therefore, it must already have delivered c_i before the crash. Therefore N' can also commit T_i.
   b.) N' has not yet received c_i. Here, again two cases must be distinguished:
      i. The coordinator did not know about T_i because it had not received the write set WO_i: due to requirement 1, N' had not delivered WO_i and can abort it. N had also not yet delivered T_i and will abort it upon recovery.
      ii. The coordinator had already aborted T_i because it had received (and therefore delivered) a_i before N failed. This means that N' was only missing the a_i message and can now safely abort T_i.

   Note, that C and no other node can have committed T_i due to requirement 2.

This protocol is non-blocking and correct. Correct means that all processes that reach a decision reach the same one, no process reverses its decision after it has reached one, and at any point in this execution, if all existing node-failures are repaired and no new failures occur for sufficiently long, then all processes will eventually reach a decision. It is trivial to see that the latter two points are fulfilled. Also, the protocol description above already includes the reasoning why all available nodes and the failed node N will reach the same decision. The argument is similar for any other node that fails during the coordination process. If the coordinator fails before the decision message can be delivered a new coordinator can independently decide because no node will deliver the decisions of the previous coordinator.

5.3 Blocking Protocols

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 requirements for this case are:

1. The delivery of a write set WO_i is atomic.
2. The delivery of a commit message c_i is non-atomic.

The same scenarios explained above apply, except for the first decision of the coordinator.

1. C does not know about the outcome of T_i (it has received or even delivered WO_i, but it has not yet received c_i or a_i): N might or might not have committed T_i. C sends a request to all available nodes and they answer with what they know. If one node has already received c_i or a_i, C decides to commit/abort, otherwise C must block the transaction and all wait until N recovers.

Several points must be noted here. First, blocking does not imply that the nodes will be completely unable to process transactions. Transactions that do not conflict with the blocked transactions can be executed as usual. Only transactions that conflict with the blocked transactions cannot be executed. In practice, it is probably reasonable to abort these transactions to avoid locking too many resources in addition to those unavailable because of the failure. Second, this same protocol can be used in a slightly different manner. Instead of blocking the transaction, the coordinator can decide to abort the transaction and later let the failed node use compensation to undo the changes of the transaction in case it had committed it. This version is effectively the same as the one below but it incurs in a higher message overhead.

5.4 Reconciliation based Protocols

The overhead of atomic delivery can be avoided entirely by not broadcasting any message (WO_i or c_i) atomically. The result is that the failed node N may have committed T_i but the rest of the nodes have not even received WO_i. In this case, the available nodes will ignore 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 that this case happens is small enough to make this protocol a viable alternative when performance needs to be improved.

5.5 Atomicity for Snapshot Isolation

In the snapshot isolation protocol, only one message is exchanged. Then the only case to consider is whether the write set is delivered atomically or not. The discussion is very similar to the one for serializability and cursor stability and omitted due to space limitations.

6 Simulation Model

The performance of the replication protocols described above has been studied in detail through a simulation model similar to those used in other evaluations of concurrency control protocols [ACL87, CL89, GHR97]. The simulation parameters are summarized in Table 1.

The database is modeled as a collection of *DBSize* objects and each of the *NumSites* sites stores copies of all objects. CPUs and disks are modeled as resources that can be consumed. The access time parameters capture the time a resource is used to perform the operation.

General	*NumSize* *DBSize*	Number of sites in the system Number of objects of the database
Database System	*NumCPU* *NumDisks* *ObjectAcessTime* *DiskAccessTime* *BufferHitRatio*	Number of processors per site Number of disks per site CPU time needed to process a single operation disk access time % of operations that do not need disk access
Communication	*CommDelayBasic*	Basic delay for broadcast
Transaction Type	*TransSize* *WriteAccessPerc* *ReadWritePerc* *TransTypePerc*	Number of op. of a transaction % of write op. of the transaction type % of read op. on objects that will be written later % of the workload that belongs to this type
Concurrency	*InterArrival*	Average time between the arrival of two transactions at one node

Table 1: Simulation model parameters

In contrast to other simulation studies of distributed systems [CL89, CFLS91, GHR97], message transfers do not only consist of packing, transferring and unpacking, but also of ordering the messages and flow control. Although a couple of performance studies on group communication protocols exist [BC94, FvR95, MMSA+96], the results vary considerably and cannot be directly applied to our configuration. Since these studies do not provide a detailed analysis of the CPU overhead of the communication module, we assume that each site has an extra processor for communication purposes. Communication costs is determined as a function of the communication semantics. Every broadcast has a basic communication delay of *CommDelayBasic*. If a message is sent either in total order or atomically, this time is multiplied by 2. If a message 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.

We allow different transaction types. Each type is determined by a number of parameters. The objects accessed by a transaction are chosen randomly from the database. Transaction execution and concurrency control is modeled according to the algorithms described in the previous sections. When a transaction is initiated, it is assigned to a node and all read operations are performed sequentially at that node. At the end of the transaction, the write set is sent to all nodes. Each operation includes a request to obtain a lock, in some cases a disk I/O to read the object from the disk, followed by a period of CPU usage for processing the object access. The writing of pages takes place asynchronously after the transaction has committed. A transaction that is aborted is restarted immediately. We use an open queuing model. At each node every *InterArrival* time units a new transaction is started. Therefore, the number of local transactions active at a node (multiprogramming level MPL) varies depending on how long transactions need to execute.

7 Experiments and Results

We have conducted an extensive set of simulation experiments but due to space limitations only some of the results will be discussed. The baseline settings are shown in Table 2. All tests are repeated until a 90% confidence interval is reached.

The main performance metric is the response time, i.e., the time from the start of a transaction until its completion. The response time of a transaction consists of the execution time

NumSize	10
DBSize	10000
NumCPU	1
NumDisks	10
ObjectAcessTime	0.3 ms
DiskAccessTime	20 ms
BufferHitRatio	80%

Table 2: Baseline parameter settings

(CPU + I/O) and the communication delay. In addition, abort rates in each protocol are also evaluated to provide a more complete picture of the results. In the following figures the following abbreviations are used: SER for serializability, CS for cursor stability, and SI for snapshot isolation. Only the non-blocking and the reconciliation based versions of the algorithms are compared. The latter version is labeled with RB in the figures.

7.1 Experiment 1: The Impact of the Communication Delay

Since existing performance studies on group communication systems present considerable different results [BC94, FvR95, MMSA+96], our first test suite investigates the impact of the communication delay on the replication service. Table 3 shows the settings for this test suite. Since read-only transactions do not require any communication, these experiments are only conducted with update transactions. We have performed tests with two different transaction types (*short* and *long*) to analyze workloads with different data contention. The parameter for the basic communication delay has been varied from 2 to 100 ms.

	Short	Long
TransSize	10	30
WriteAccessPerc	40%	40%
ReadWritePerc	30%	30%

Table 3: Experiment 1: transaction types

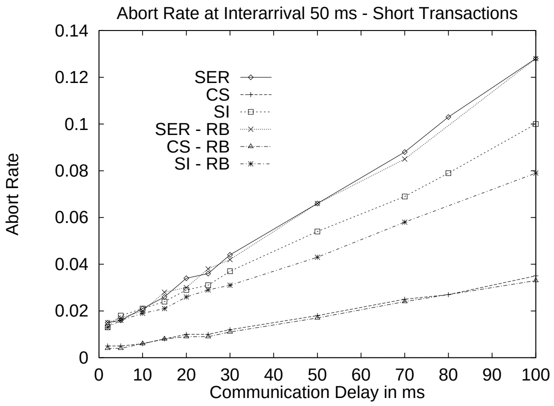


Figure 5: Experiment 1: abort rate of short transactions at an inter arrival time of 50 ms

Figure 5 illustrates the abort rates of the protocols as a function of the communication delay. The results are for the special case of short transactions at an inter arrival time of 50 ms but some general observations that hold for all transaction types and inter arrival times can be derived. The fact that abort rates increase as the communication delay increases is explained by the number of transactions in the system. Slow communication delays transactions which causes transactions to spend more time in the system. As more transactions coexist, the probability of conflicts increases and with it the abort rate. In the case of SER and CS, the non-blocking and the reconciliation based protocols behave similarly. The reason is that in these protocols a transaction can only be aborted when it is in its reading phase or when it waits for its write set to be delivered. These phases are the same in both versions of the protocol. With SI, on the other hand, the reconciliation based protocol has a lower abort rate than the non-blocking version because it shortens the time from BOT to the delivery of the write set, thereby reducing the conflict profile of the transaction.

Comparatively, CS has by far the lowest abort rate. SER and SI have similar abort rates with low communication delays, but as the communication delay increases the behavior of SER degrades. This is due to readers aborted upon arrival of a write transaction which is later also aborted. Aborting the readers was unnecessary, but, as the communication delay increases, the likelihood of such cases increases. SI does not have this problem because the decision on abort or commit can be done independently at each node and a transaction T_i only acquires locks when it is able to commit.

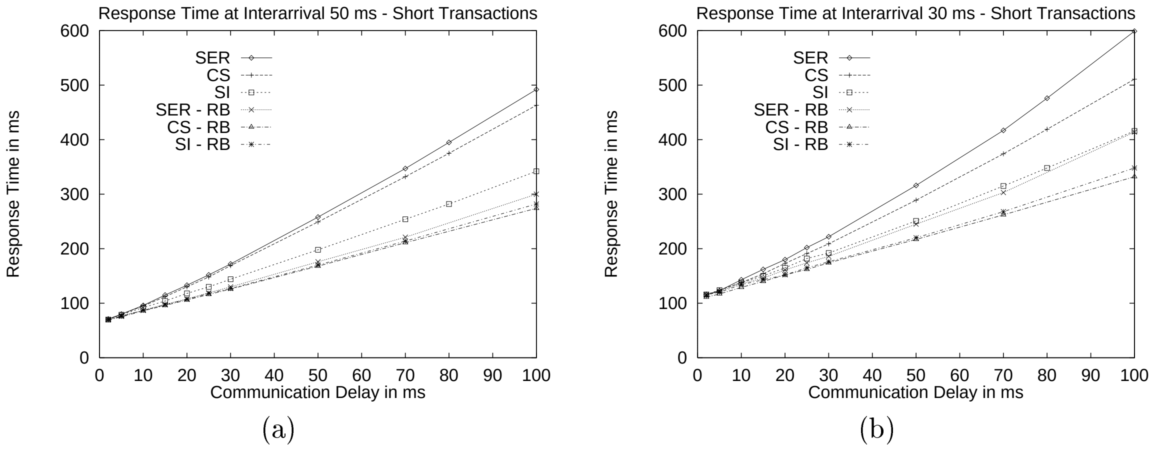
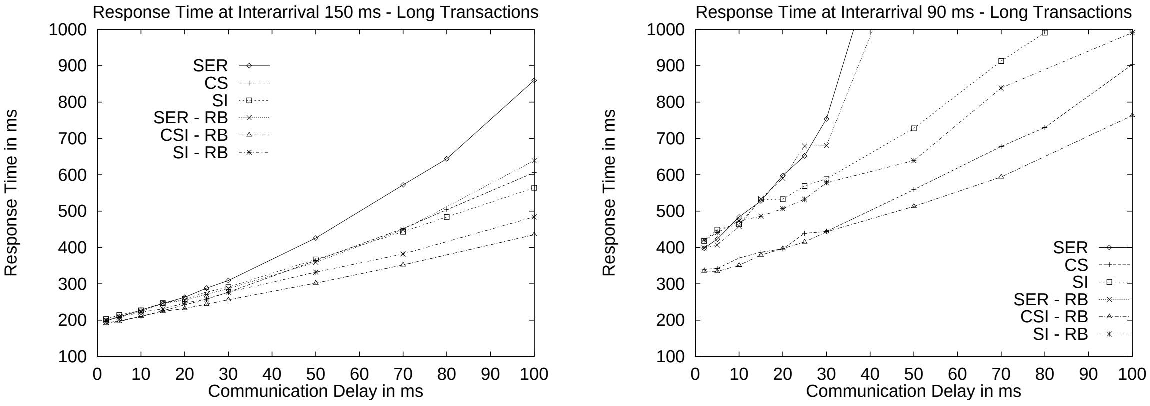


Figure 6: Experiment 1: response time of short transactions at an inter arrival time of (a) 50 ms and (b) 30 ms

Figure 6 shows the response time for short transactions for an inter arrival time of 50 ms and of 30 ms, respectively. At 50 ms resource and data contention is small for all communication delays. Hence, the message delay has a great impact on the response time. With a low communication delay the response time is due to the execution time which is roughly similar for all protocols. However, with increasing message delay, the response time of the different protocols increases depending on the number and complexity of the message exchanges. The non-blocking protocols tend to show worse performance. At an inter arrival time of 30 ms data and resource contention are generally higher (data contention was observed to be three times higher). Therefore, the response time is higher than with longer inter arrival times. The qualitative behavior of the protocols is, nonetheless, similar for both inter arrival rates.

Figure 7 shows the response time for long transactions. Since long transactions have three times as many operations than the short ones we chose as inter arrival times 150 ms and 90 ms



(a) (b)

Figure 7: Experiment 1: response time of long transactions at a inter arrival time of (a) 150 ms and (b) 90 ms

to have similar resource contention as in the previous experiment. In this case, the response time is much more influenced by the data contention and we can always observe performance differences between the concurrency control methods for both inter arrival times.

At an inter arrival time of 150 ms the abort rates for CS have always been under 10% while they reached over 50% for SER and SI. Therefore, CS provides better performance than the other methods when the message overhead is low. However, with increasing message delay the message overhead becomes as dominant as in the previous experiments. At an inter arrival time of 90 ms the conflict rates are so high that the concurrency control method becomes the dominant factor in the response time. Since the resource utilization is high, the abort and restart of a long transaction degrades performance more than long message delays. CS (both the non-blocking and the reconciliation based protocols) shows clearly the best performance for all message delays because of its low conflict rate. SER, on the other hand, has such a high data contention that it starts thrashing very quickly as communication slows down.

We can conclude that the choice of the best protocol depends on the workload and the configuration of the system. If the workload is low, or has low conflict rates and communication delay serializability and atomic message delivery could be used to provide full consistency and correctness. However, if the communication system is slow, performance can only be maintained by choosing 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.

7.2 Experiment 2: Resource and Data Contention

The distribution of read and write operations is relevant in two ways. First, it influences the resource contention, since read operations are only performed at one node and write operations at all nodes. Second, depending on the replica control method, conflict resolution is either done by aborting readers or writers. Thus, in this set of experiments, we consider the same two transaction types as in experiment 1 but the percentage of write operations is varied from

10% to 100%. The read-dependency is set to 30% or less (with only one write operation, it is not possible that 30% of read operations are on objects that will be written later). 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.

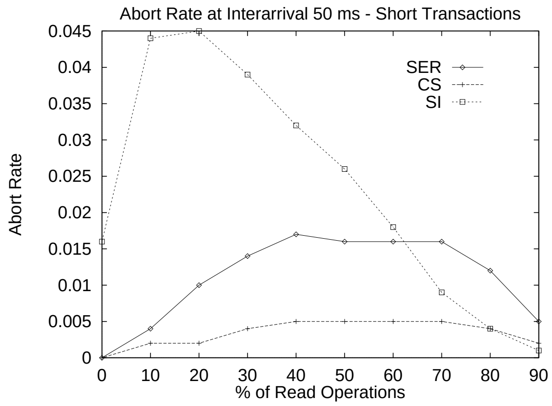


Figure 8: Experiment 2: abort rate of short transactions at an inter arrival time of 50 ms

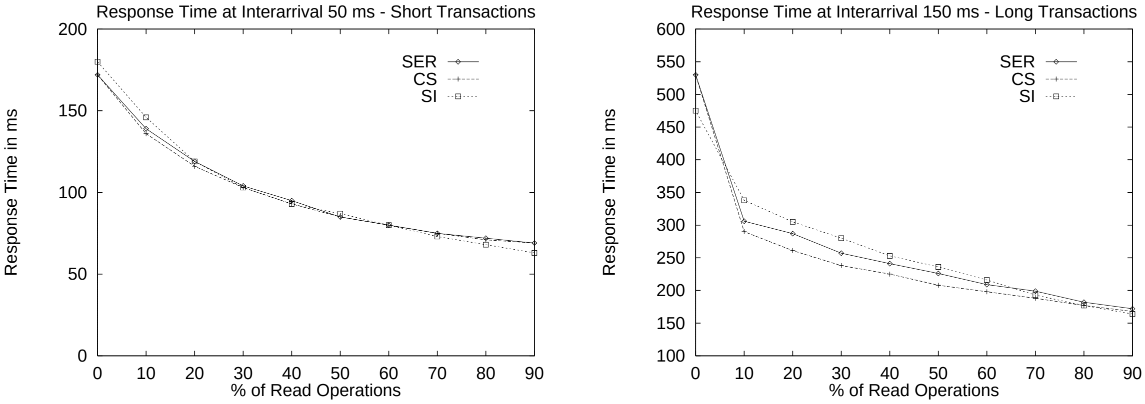
As in the previous example we first want to present some general observations that hold for all tests of this experiment. First, the resource utilization decreases substantially with increasing read rates because updates must be performed at all nodes while reads are only done locally at one node. The resource utilization also depends slightly on the abort rate.

Figure 8 presents the abort rate as a function of the read rate for short transactions and an inter arrival time of 50 ms. CS always has the smallest abort rate. For SER and CS the abort rate is zero with only write operations, since writers are never aborted. For SI the abort rate is rather small because no operations are performed until the write set is sent. For SER and CS the abort rate then increases slightly because reads might be aborted by writers but then decreases because there are less writers 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 readers. Note that snapshot isolation has higher conflict rates than SER for high update rates although it does not provide serializability due to the optimistic approach of SI which is optimized for small update rates.

Figure 9 shows the response time of short transactions at an inter arrival time of 50 ms and of long transactions at an inter arrival time of 150 ms. The behavior of all protocols is very similar and mainly determined by the resource utilization. The response time decreases with an increase of the read rate. For small transactions data contention is so small that it has no effect on the performance. At low read rates the system is near its thrashing point because of the high resource utilization.

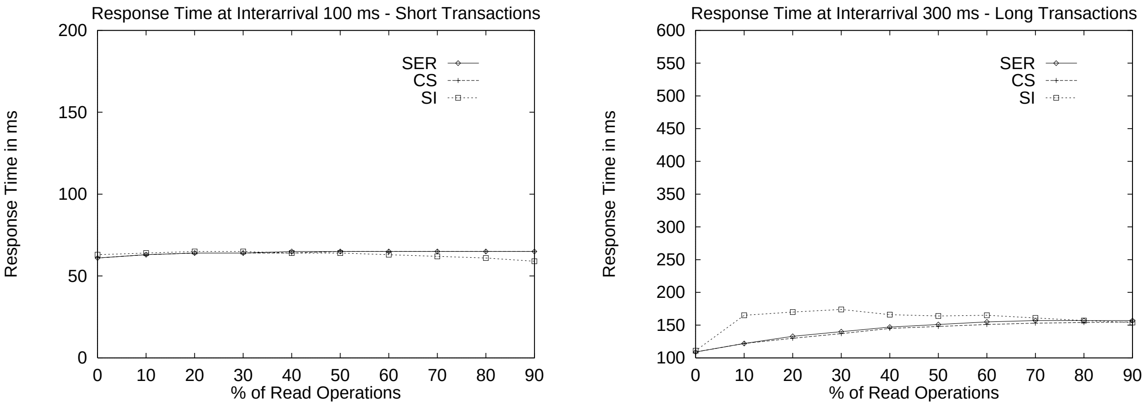
For long transactions the abort rates are generally higher having more impact on the response time and we can see performance differences reflecting the abort rates depicted in Figure 8. With no read operations, the abort rates are very high and the restarts eat up the resources leading to thrashing behavior.

We also conducted tests with very low workloads. Figure 10 shows the response time of short transactions at an inter arrival time of 100 ms and of long transactions at an inter arrival



(a) (b)

Figure 9: Experiment 2: response time of (a) short transactions at an inter arrival time of 50 ms and of (b) long transactions at an inter arrival time of 150 ms



(a) (b)

Figure 10: Experiment 2: response time of (a) short transactions at an inter arrival time of 100 ms and of (b) long transactions at an inter arrival time of 300 ms

time of 300 ms. At these loads, the transactions are nearly performed sequentially and any resource or data contention is avoided. With such a configuration we can observe lower response times at high update rates. This is due to the fact that the updates are delayed and then performed in parallel. The higher the update rate the more operations of a transactions are executed concurrently, decreasing the transaction's response time. Furthermore, we can still observe the disadvantage of SI compared to the other protocols at high update rates.

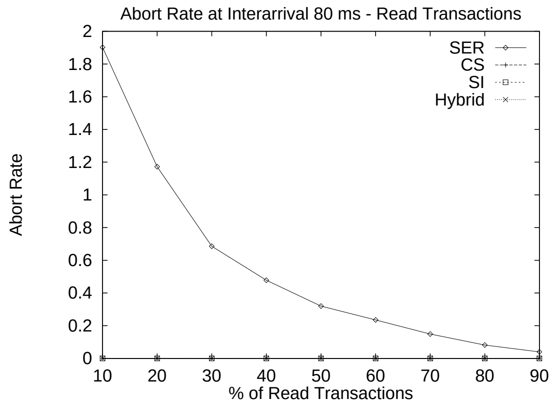
In the configuration studied, resource contention is a more relevant factor than data contention if the update rates are high. This problem is inherent to replication schemes where all updates are performed at all copies. However, update rates in practical applications do not seem to be extremely high: even in OLTP workloads like the TPC-C benchmark [Com95], transactions normally have at least 50% read operations.

7.3 Experiment 3: Read-Only Transactions

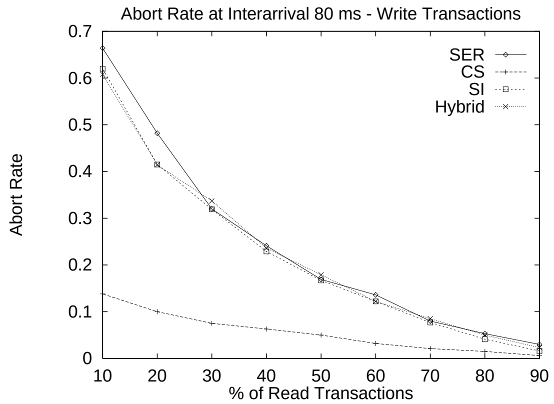
The main gain of replication lies in the ability to perform read operations locally. Replication pays off when a great part of the transactions are queries which can be executed locally without any communication costs. As a last experiment we have performed a test suite analyzing the behavior of the concurrency control protocols using a mixed workload consisting of updating and read-only transactions. Table 4 shows the parameter setting for the transaction types. The percentage *TransTypePerc* of both transactions types are varied between 10% to 90%. Again, we chose a fast communication system and only look at the non-blocking protocols. This time, we also analyze the hybrid protocol using 2-Phase Locking for updating transactions and a snapshot for read-only transactions.

	Update	Read Only
TransSize	30	30
WriteAccessPerc	40%	0%
ReadWritePerc	30%	0%

Table 4: Experiment 3: transaction types



(a)

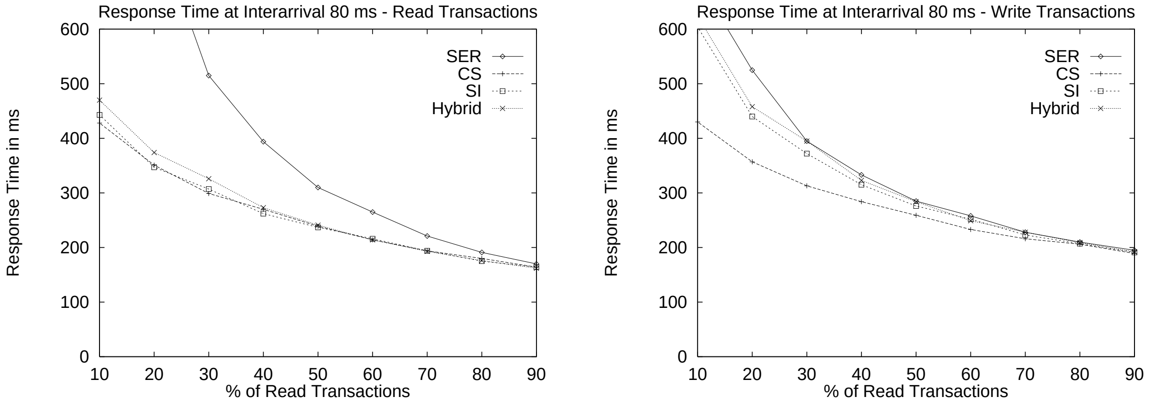


(b)

Figure 11: Experiment 3: abort rate of (a) read-only transactions and (b) update transactions at an inter arrival time of 80 ms

We conducted the experiments for an inter arrival time of 80 ms. The workload is higher than in the previous experiments because with decreasing number of write operations the resource utilization decreased significantly. Figure 11 shows the abort rate for the read-only transactions and update transactions as a function of the percentage of read-only transactions. Only serializability aborts readers. It has very high abort rates if many update transactions are in the system, however the abort rate decreases very fast with an increasing number of read-only transactions. The abort rate for update transactions is rather high if many updaters are in the system as shown in experiment 1 and cursor stability behaves significantly better than the others. Serializability behaves worse than the other algorithms for up to 20% of read-only transactions because the combined high abort rate of read-only and update transactions leads to higher resource contention and the system is close to thrashing. With increasing number of read-only transactions the abort rates decrease fast.

Figure 12 shows the response time for the read-only transactions and update transactions



(a) (b)

Figure 12: Experiment 3: response time of (a) read-only transactions and (b) update transactions at an inter arrival time of 80 ms

as a function of the percentage of read-only transactions in the workload for an inter arrival time of 80 ms. The response times decrease when the percentage of read-only transactions increases because of less data contention and less resource contention. The differences in the protocols directly reflect the different abort rates for readers and writers of the different protocols.

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 also has good performance results, it does not provide repeatable reads, which may be problematic in certain applications.

8 Conclusion

In this paper, we have analyzed several approaches to maintain a replicated and distributed database by using group communication to provide message exchange and fault-tolerance, to support the ordering of transactions and to avoid the problem of distributed deadlocks. Starting with a fault-tolerant, 2-phase locking protocol, presented by [AAES97] 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 system and workload configurations. Using a detailed simulation model of a replicated database system, we evaluated the response time of the approaches and gave an analysis of their behavior.

Our experiments demonstrate the following points:

1. 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. These protocols perform significantly better than full fault-tolerant protocols when the communication is slow. Also our technique to minimize the number of messages per transaction allows to limit the communication overhead. For snapshot isolation we were able to reduce the message overhead to one message per transaction.

2. With our protocols, the message overhead is constant and does not increase with the transaction size. Therefore, the direct impact of the message delay is smaller for longer transactions. 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 is more severe for long than for short transaction. It can be overcome by using concurrency control protocols with lower isolation levels, e.g., cursor stability.

3. Resource contention can become a severe problem in a replicated system if the update rates are high. To alleviate the problem, conflict rates can be reduced by using cursor stability or snapshot isolation for concurrency control.

4. The hybrid protocol proposed is an elegant solution to avoid having to abort queries when serializability is enforced. Transactions updating the database are executed using the normal protocol while queries are executed using snapshot isolation.

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 if the percentage of read-only transactions is reasonable high, then standard 2-phase locking 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.

References

[AAES97] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Euro-Par'97*, Passau, Germany, August 1997. An extended version is available as technical report, Department of Computer Science, University of California, Santa Barbara, 1996.

[ACL87] R. Agrawal, M.J. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609–654, 1987.

[Alo97] G. Alonso. Partial database replication and group communication primitives. In *2nd European Research Seminar on Advances in Distributed Systems (ERSADS'97)*, Zinal (Valais, Switzerland), March 1997.

[BBG+95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–10, June 1995.

[BC94] K. Birman and T. Clark. Performance of the isis distributed computing toolkit. Technical report, Department of Computer Science, Cornell University, TR-94-1432, June 1994.

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

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[CFLS91] M. J. Carey, M. J. Franklin, M. Livny, and E. J. Shekita. Data caching tradeoffs in client-server DBMS architectures. In *Proceedings of the 1991 ACM SIGMOD Int. Conference on Management of Data*, pages 357–366, Denver, May 1991.

[CL89] M. Carey and M. Livny. Parallelism and concurrency control performance in distributed database machines. In *Proc. of the 1989 ACM SIGMOD*, June 1989.

[Com95] D. E. Comer. *Internetworking with TCP/IP, Volume 1.* Prentice Hall, 1995.

[CT91] T. D. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325–340, August 1991.

[DM96] D. Dolev and D. Malki. The transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):63–70, April 1996.

[ET89] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems*, 14(2):264–290, June 1989.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[FvR95] R. Friedman and R. van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. Technical report, Department of Computer Science, Cornell University, TR-95-1527, July 1995.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, June 1996.

[GHR97] R. Gupta, Jayant Haritsa, and K. Ramamritham. Revisiting commit processing in distributed database systems. In *Proc. of the 1997 ACM SIGMOD*, pages 486–497, Tucson, Arizona, June 1997.

[Gol94] R. Goldring. A discussion of relational database replication technology. *InfoDB*, 8(1), 1994.

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

[KLS90] H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *Proc. of the 16th Int. Conference on Very Large Data Bases*, pages 95–106, Brisbane, Australia, September 1990.

[Kna87] E. Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys*, 19(4):303–328, 1987.

[MMSA+96] 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.

[Ora95] Oracle. *Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7*, July 1995. White Paper.

[Ora97] Oracle. *Oracle8(TM) Server Replication, Concepts Manual*, 1997.

[PGS97] F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. In *16th IEEE Symposium on Reliable Distributed Systems (SRDS'97)*, Durham (USA), October 1997.

[SR96] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, April 1996.

[SS83] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computing Systems*, 1(3):222–238, August 1983.

[Sta94] D. Stacey. Replication: Db2, oracle, or sybase. *Database Programming & Design*, 7(12), 1994.

[vRBM96] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.