A New Approach to Developing and Implementing Eager Database Replication Protocols

Bettina Kemme
and
Gustavo Alonso
Swiss Federal Institute of Technology (ETH)

Database replication is traditionally seen as a way to increase the availability and performance of distributed databases. Although a large number of protocols providing data consistency and fault-tolerance have been proposed, few of these ideas have ever been used in commercial products due to their complexity and performance implications. Instead, current products allow inconsistencies and often resort to centralized approaches which eliminates some of the advantages of replication. As an alternative, we propose a suite of replication protocols that addresses the main problems related to database replication. On the one hand, our protocols maintain data consistency and the same transactional semantics found in centralized systems. On the other hand, they provide flexibility and reasonable performance. To do so, our protocols take advantage of the rich semantics of group communication primitives and the relaxed isolation guarantees provided by most databases. This allows us 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.

Categories and Subject Descriptors: H.2.4 [**Information Systems**]: Database Management—*concurrency, distributed databases, transaction processing*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.4 [**Performance of Systems**]

General Terms: Algorithms, Management, Performance, Reliability

Additional Key Words and Phrases: Database replication, fault-tolerance, group communication, isolation levels, one-copy-serializability, replica control, total order multicast

---

This work was partially supported by ETH Zürich within the DRAGON Research Project (Reg-Nr. 41-2642.5)
An earlier version of this paper appeared in the *Proceedings of the IEEE International Conference on Distributed Computing Systems (May 26-29, 1998) pp. 156-163.*
Affiliation: Department of Computer Science, Federal Institute of Technology (ETH)
Address: Institute of Information Systems, ETH Zentrum, CH-8092 Zürich, Switzerland, E-Mail: {kemme,alonso}@inf.ethz.ch

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. In spite of the large number of existing protocols which provide data consistency and fault-tolerance [Bernstein et al. 1987], few of these ideas have ever been used in commercial products. There is a strong belief among database designers that most existing solutions are not feasible due to their complexity, poor performance and lack of scalability. As a result, current products adopt a very pragmatic approach: copies are not kept consistent, updates are often centralized, and solving inconsistencies is left to the user [Stacey 1994]. To bridge the gap between replication theory and practice, it is necessary to find a compromise between the correctness and fault-tolerance of research solutions and the efficiency of commercial systems. What is needed is a practical solution that guarantees consistency, provides clear correctness criteria and reasonable fault-tolerance, avoids bottlenecks, and has good performance. In this paper, we propose such a solution based on a combination of ideas from group communication and concurrency control.

Communication is a key issue in distributed computing, since efficiency can only be achieved when the communication overhead is small. Therefore, unlike previous replication protocols, our solution is tightly integrated with the underlying communication system. Following initial work in this area [Agrawal et al. 1997; Alonso 1997; Pedone et al. 1997; Kemme and Alonso 1998; Stanoi et al. 1998; Holliday et al. 1999], we exploit the semantics of group communication [Hadzilacos and Toueg 1993] in order to minimize the overhead. Group communication systems, such as Isis [Birman et al. 1991], Transis [Dolev and Malki 1996], Totem [Moser et al. 1996] or Horus [van Renesse et al. 1996], provide group maintenance, reliable message exchange, and message ordering primitives between a group of nodes. Their fault-tolerance and consistency services will be used to perform some of the tasks of the database, thereby avoiding some of the performance limitations of current replication protocols.

A second important point is correctness. It is well known that serializability provides the highest correctness level but is too restrictive in practice. To address this problem, our protocols implement different levels of isolation as implemented by commercial systems [Berenson et al. 1995]. The protocols also provide different levels of fault-tolerance to allow different failure behaviors at different costs. Our correctness criteria provide a high degree of flexibility, guaranteeing correctness when needed and allowing the relaxation of correctness when performance is the main issue. However, our approach always guarantees data consistency.

The basic mechanism behind our protocols is to first perform a transaction locally, deferring and batching writes to remote copies until transaction commit time. At commit time all updates (the write set) are sent to all copies using a total order multicast which guarantees that all nodes receive all write sets in exactly the same order. Upon reception, each site (including the local site) performs a conflict test checking for read/write conflicts. Only transactions passing this test can commit. Conflicting write operations are executed in arrival order, thereby serializing conflicting transactions. As a result, no explicit 2-phase-commit protocol is needed

Table 1. Classification of replication mechanisms

where \ when	Eager	Lazy
Primary Copy	Early Solutions in Ingres	Sybase/IBM/Oracle Placement Strat.
	Serialization-Graph based	
Update Everywhere	ROWA/ROWAA Quorum based Oracle Synchr.Repl.	Oracle Advanced Repl. Weak Consistency Strat.

and no deadlock can occur.

The work presented in this paper makes several contributions. First, it proposes a family of replication protocols combining several ideas: the use of group communication, the use of different levels of isolation to enhance performance, and the incorporation of communication semantics to optimize fault-tolerance. Second, the problem of implementing replication efficiently is explored in detail by means of a simulation study. We believe that our approach solves many of the problems related to database replication [Gray et al. 1996]. On the one hand, our protocols maintain data consistency, replication transparency and fault-tolerance. On the other hand, they can be easily integrated in current systems, provide flexibility, reasonable performance and the same transactional semantics found in centralized systems.

The paper is organized as follows: Section 2 provides an overview of existing replication solutions. Section 3 presents the basic ideas of our solution. Section 4 describes the system model. Section 5 presents a family of protocols providing different levels of isolation and Section 6 redefines the algorithms to provide different levels of fault-tolerance. The properties of the algorithms are formally proven in Appendix A and B. Section 7 provides an overview of the simulation system. Section 8 describes the conducted experiments and their results. Section 9 concludes the paper.

2. DATABASE REPLICATION: AN OVERVIEW

In databases, *replica control* mechanisms ensure data consistency between the copies. Gray, Helland, O'Neil, and Shasha [1996] categorize these mechanisms according to when updates are propagated and which copies can be updated (Table 1).

Update propagation can be done within or outside the transaction boundaries. In the first case, replication is *eager*, otherwise it is *lazy*. Eager replication allows the detection of conflicts before the transaction commits. This approach provides data consistency in a straightforward way, but the resulting communication overhead increases response times significantly. To keep response times short, lazy replication delays the propagation of changes until after the end of the transaction, implement-

ing update propagation as a background process. However, since copies are allowed to diverge, inconsistencies might occur.

In terms of which copy to update, there are two possibilities: centralizing updates (*primary copy*) or a distributed approach (*update everywhere*). Using a primary copy approach, all updates are first performed at a primary copy and then propagated to the secondary copies. This avoids concurrent updates to different copies and simplifies concurrency control, but it also introduces a potential bottleneck and a single point of failure. Update everywhere allows any copy to be updated requiring the updates to the different copies to be coordinated.

2.1 Eager Replication

Early research in replication addressed eager replication since it provides strong consistency and a high degree of fault-tolerance. The conventional correctness criterion for eager replication is *1-copy-serializability* (1CSR) [Bernstein et al. 1987]: despite the existence of multiple copies, an object appears as one logical copy (called *1-copy-equivalence*) and the execution of concurrent transactions is coordinated so that it is equivalent to a serial execution over the logical copy (*serializability*).

Table 1 classifies some of the better known eager protocols [Bernstein et al. 1987; Ceri et al. 1991]. Early solutions used eager primary copy approaches [Alsberg and Day 1976; Stonebraker 1979]. Later algorithms followed an update everywhere approach based on *quorums*, e.g. *read-one/write-all* (ROWA) [Bernstein et al. 1987] or *read-one/write-all-available* (ROWAA). Significant efforts have been devoted to optimizing quorum sizes [Gifford 1979; Maekawa 1985; El Abbadi and Toueg 1989; Cheung et al. 1990; Agrawal and El Abbadi 1990]. More recently, epidemic protocols have been proposed [Agrawal et al. 1997] in which communication mechanisms providing causality are augmented to ensure serializability.

2.2 Lazy Replication

Despite the elegance of eager replication, only the most straightforward solutions have been included in commercial systems. Users are often warned to use eager replication only when full synchronization is required [Oracle 1997]. In practice, commercial databases favor lazy propagation models. Many different approaches exist (Table 1), which are mostly based on primary copy and often highly specialized to either "On Line Transaction Processing" (OLTP) or "On Line Analytical Processing" (OLAP) [Stacey 1994; Goldring 1994]. For example, *Sybase Replication Server* has a primary copy approach where updates are propagated to the other copies immediately after the commit of the transaction. This *push* strategy is an effort to minimize the time that the copies are inconsistent. *IBM Data Propagator* is geared towards OLAP architectures. It adopts a *pull* strategy in which updates are propagated only at the client request. This implies that a client will not see its own updates unless it requests them from the central copy. *Oracle Symmetric Replication* [Oracle 1997] supports both push and pull strategies, as well as eager and lazy replication. Eager replication is implemented through stored procedures activated by triggers. Oracle's lazy replication schemes allow both for primary copy and update everywhere replication. In the latter case however, when transactions perform conflicting operations, the non-trivial problem of reconciliation arises.

In addition to commercial products, considerable work has been done to develop

lazy replication strategies. A wide range of approaches has been proposed using weak consistency models [Pu and Leff 1991; Krishnakumar and Bernstein 1991; Chen and Pu 1992], economic paradigms [Sidell et al. 1996] or epidemic strategies [Agrawal et al. 1997]. There are also many solutions outside the database domain, e.g., distributed file systems, replication on the web [Rabinovich et al. 1996], or document replication [Alonso et al. 1997].

More recently, efforts have been made to develop lazy solutions guaranteeing serializability [Chundi et al. 1996; Pacitti et al. 1999; Breitbart et al. 1999]. All of them are based on primary copy approaches. The basic idea is to restrict the placement of primary and secondary copies and to control the order in which updates are applied to the secondary copies. These approaches are able to provide fast response time (no message exchange during the execution of a transaction) and guarantee serializability. However, both the set of possible configurations and transaction executions are severely restricted.

Another recent approach combines eager and lazy replication techniques [Breitbart and Korth 1997; Anderson et al. 1998]. The system is eager in the sense that communication takes place within the boundaries of the transaction to determine the serialization order. This is either done by using a distributed locking scheme (before each access a lock for the primary copy must be acquired) or by building a global serialization graph to detect and avoid non-serializable executions. The system is also lazy since the execution of a transaction takes place at one site and propagation of updates to the remote copies is only done after the commit. A 2-phase-commit is not necessary since the mechanisms guarantee that the updates of committed transactions will be applied at all sites. This solution is primary copy based and does not allow transactions to update primary copies residing on different sites.

3. MINIMIZING THE OVERHEAD OF REPLICATION PROTOCOLS

In view of these ideas, the question to ask is whether it is possible to combine the correctness properties of eager update everywhere with the performance advantages of lazy replication. This is the goal of the protocols we propose and for this purpose we use a number of techniques:

Reducing the Message Overhead. To separate the problem of global serializability and replica control, most conventional protocols treat each operation in a transaction individually. This leads to a significant number of messages per transaction – an approach that cannot scale beyond a few nodes. To minimize the message overhead, we bundle update operations into a single message as it is done in optimistic schemes and in lazy replication. We do this by postponing all writes to replicated data until the end of the transaction. Furthermore, assuming that most applications have significantly more read than write operations, we use a ROWAA solution. With this, read operations are performed locally and no information about read operations needs to be exchanged.

Eliminating Deadlocks. Gray, Helland, O'Neil, and Shasha [1996] have shown that in some configurations the probability of deadlock is directly proportional to n^3, n being the number of replicas. A way to avoid deadlocks is to pre-order transactions; we do this by using group communication [Hadzilacos and Toueg

1993]. With group communication it is possible to ensure that all messages are received in the same total order at all sites. If all operations of a transaction are sent in a single message and transactions arrive at all sites in the same order, it suffices to grant the locks in the order the transactions arrive to be able to guarantee that all sites perform the same updates and in exactly the same order and to avoid any kind of deadlock. Note that the total order on the delivery of transactions does not imply a serial execution. Non-conflicting operations can be executed in parallel. Even if serial execution were to be necessary, it has been recently suggested that if enough CPU is available, it pays off to execute transactions serially, skipping concurrency control entirely [Shasha 1997; Whitney et al. 1997].

Optimizations Using Different Levels of Isolation. Serializability [Eswaran et al. 1976] is often too restrictive and commercial databases use alternative correctness criteria that allow lower levels of isolation [ANSI X3.135-1992 1992; Gray and Reuter 1993; Berenson et al. 1995]. 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. They significantly improve performance but allow inconsistencies. They are somehow comparable to lazy solutions but they have the advantage of being more understandable and well accepted among database practitioners. These relaxed correctness requirements are even more relevant in a distributed environment where the longer transaction execution times lead to higher conflict rates.

Optimizations Using Different Levels of Fault-Tolerance. Most traditional protocols introduce a considerable amount of additional overhead during normal processing in order to provide fault-tolerance. As it is done with different levels of isolation, full correctness can be weakened to provide faster solutions. In the context of this paper, the reliability of message delivery will determine the overall correctness. While complex message exchange mechanisms guarantee the atomicity of transactions beyond failures, faster communication protocols only provide a best effort approach. This trade-off is very similar to that found in the 1-safe and 2-safe configurations typically found in back-up systems [Gray and Reuter 1993; Alonso et al. 1996].

4. MODEL

In this section we present a formal model of a distributed database. A distributed database consists of a number of nodes, $N_i, (0 < i \leq n)$, also called sites. Each node maintains a set of objects which can be accessed by executing transactions. We assume a fully replicated system. Nodes communicate with each other by exchanging messages. Figure 1.a shows the main components of a node [Bernstein et al. 1987; Gray and Reuter 1993].

4.1 Communication Model

Communication takes place using the multicast primitives provided by a group communication system [Birman et al. 1991; Chandra and Toueg 1991; Moser et al. 1996; Dolev and Malki 1996; van Renesse et al. 1996]. The primitives manage the message exchange between groups of nodes providing message ordering and fault-tolerance. For the moment, we assume that a group is static and consists of a

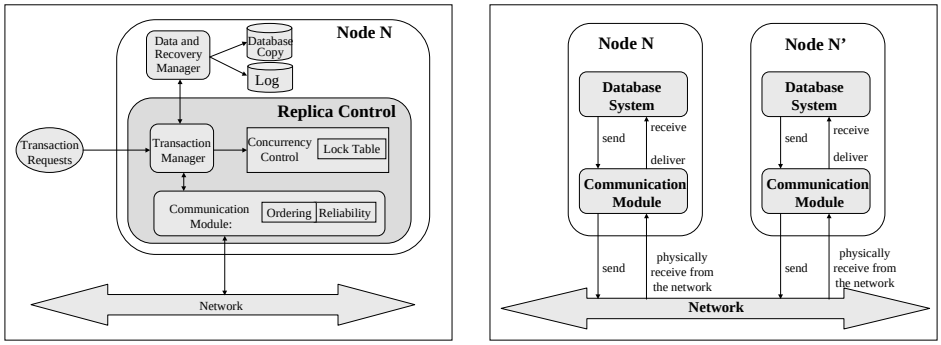

Fig. 1. (a) Node architecture and (b) communication model

fixed number of nodes that never fail. In Section 6 we enhance the model and the protocol semantics to account for failures and recovery of sites.

Figure 1.b depicts the layered configuration we use. At each site, the database system sends messages by passing them to the communication module which, in turn, forwards them to all sites including the sending site. We say that a node N *multicasts* or *sends* a message to all group members. The communication module on each node N physically *receives* the message from the network and *delivers* it to the local database system. Upon delivery, the local database *receives* the message. In the following sections, we use the terms "a message is delivered at/to node N" and "node N receives a message" interchangeably to refer to the reception of a message by the database. In general, the delivery of the same message at the different nodes does not take place at the same time: at one site a message might already be delivered while at another node the communication module has not yet received the message from the network.

Group communication systems usually provide a *delivery guarantee*, i.e., a message is eventually delivered at all sites, even when some failure occurs (e.g., message loss). Another important feature is *message ordering*. The concept of ordering is motivated by the fact that different messages might depend on each other. For instance, if a node multicasts two messages, the content of the second message might depend on the content of the first, and hence, all nodes should receive the messages in the order they were sent (FIFO). Causality [Lamport 1978] extends the FIFO ordering. Formally, we define the *causal precedence* of messages as the reflexive and transitive closure of: m causally precedes m' if a node N receives m before it sends m' or if a node N sends m before sending m'. FIFO and causal orders do not impose any delivery order on messages that are not causally related. Thus, different nodes might receive unrelated messages in different orders. A total order prevents this by ensuring that all messages at all nodes are delivered in the same serial order. To summarize, group communication systems provide different delivery orders:

—*Basic* service: A message is delivered whenever it is physically received from the network. Each node receives the messages in arbitrary order.

—*FIFO* service: Messages sent by a given node are delivered at all nodes in the order they were sent.

—*Causal* order service: If a message m causally precedes a message m', then m is delivered before m' at all nodes.

—*Total* order service: Messages are delivered in the same total order at all sites, i.e., if any two nodes N and N' receive some messages m and m', then either both receive m before m' or both receive m' before m.

There exist many different algorithms to implement these delivery orders. For instance, Totem [Moser et al. 1996] uses a token to implement total order. Another well known approach has been implemented in ISIS [Birman et al. 1991], where a master site determines the sequence numbers and multicasts them regularly to the other sites. The exact number of physical messages per multicast and the delay between sending and receiving a multicast message strongly depend on the ordering algorithm and whether broadcast or point-to-point communication is used.

4.2 Transaction Model

Users interact with the database by submitting transactions to one node [Bernstein et al. 1987]. Transactions execute atomically, i.e., a transaction T_i either commits, (c_i), or aborts, (a_i), its results at all participating sites. A transaction T_i is a partial order, $<_i$, of read $r_i(X)$ and write $w_i(X)$ operations on logical objects X. When a transaction reads or writes a logical object more than once during its execution, the operations are indexed accordingly (e.g. if a transaction reads an object twice, the two operations are labeled $r_{i1}(X)$ and $r_{i2}(X)$ respectively.). Since we use ROWAA, each logical read operation $r_i(X)$ is translated to a physical read $r_i(X_j)$ on the copy of the local node N_j. A write operation $w_i(X)$ is translated to physical writes $w_i(X_1), \ldots, w_i(X_n)$ on all (available) copies. Operations conflict if they are from different transactions, access the same copy and at least one of them is a write. A history H is a partial order, $<_H$, of the physical operations of a set of transactions $\mathcal{T}$ such that $\forall T_i \in \mathcal{T} : \textit{if } o_i(X) <_i o_i(Y), o \in \{r, w\}, \textit{ then } \forall o_i(X_j), o_i(Y_{j'}) \in H : o_i(X_j) <_H o_i(Y_{j'})$. 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 [Gray et al. 1976; Gray and Reuter 1993]. The highest isolation level, *conflict serializability*, requires a history to be conflict-equivalent to a serial history. Lower levels of isolation are less restrictive but allow inconsistencies. The inconsistencies that might occur are defined in terms of several phenomena. The ANSI SQL standard specifies four degrees of isolation [ANSI X3.135-1992 1992]. However, recent work has shown that many protocols implemented in commercial systems and proposed in the literature do not match the ANSI isolation levels [Berenson et al. 1995; Adya 1999; Adya et al. 2000]. In the following, we adopt the notation of Berenson, Bernstein, Gray, Melton, O'Neil, and O'Neil [1995] and describe the phenomena of interest for this paper as:

P1 *Dirty read:* $w_1(X_i) \ldots r_2(X_i) \ldots (c_1 \text{ or } a_1)$. T_2 reads an uncommitted version of X. The most severe problem of dirty reads is cascading aborts. If T_1 aborts the write operation $w_1(X_i)$ will be undone. Hence, T_2 must also be aborted since it has read an invalid version.

P2 *Lost Update:* $r_1(X_i) \ldots w_2(X_i) \ldots w_1(X_i) \ldots c_1$. T_1 might write X upon the result of its read operation but not considering the new version of X created by T_2. T_2's update is lost.

P3 *Non-Repeatable Read:* $r_{11}(X_i) \ldots w_2(X_i) \ldots c_2 \ldots r_{12}(X_i)$. T_1 reads two different values of X.

P4 *Read Skew:* $r_1(X_i) \ldots w_2(X_i) \ldots w_2(Y_j) \ldots c_2 \ldots r_1(Y_j)$. If there exists a constraint between X and Y, T_1 might read versions of X and Y that do not fulfill this constraint.

P5 *Write Skew:* $r_1(X_i) \ldots r_2(Y_j) \ldots w_1(Y_j) \ldots w_2(X_i)$. If there exists a constraint between X and Y it might be violated by the two writes.

4.3 Concurrency Control Model

In most systems, locking protocols [Bernstein et al. 1987; Gray and Reuter 1993] are used to implement the different isolation levels. Since write locks are usually not released until commit time for recovery purposes (to avoid P1), 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. Accordingly, the different protocols proposed in this paper are based on the following criteria:

Serializability implemented via strict 2-phase-locking [Bernstein et al. 1987] avoids all phenomena described above. All locks are held until the end of transaction (called long locks).

Cursor Stability avoids long delays of writers due to conflicts with read operations [Gray and Reuter 1993; Berenson et al. 1995; Adya et al. 2000] and is widely implemented in commercial systems. If transactions scan the database and perform complex read operations, long read locks can delay write operations considerably. In order to avoid such behavior, commercial systems often release read locks immediately at the end of the read operation (short read locks). This, however, allows inconsistencies to occur. To avoid the most severe of these inconsistencies, lost update ($P2$), the systems keep long read locks on objects that the transaction wants to update later on. For instance, SQL *cursors* keep a lock on the object that is currently referred to by the cursor. The lock is usually held until the cursor moves on or it is closed. If the object is updated, however, the lock is transformed into a write lock that is kept until EOT (end of the transaction) [Berenson et al. 1995]. As another example, the reading SQL SELECT-statement can be called with a "FOR UPDATE" clause, and read locks will not be released after the operation but kept until EOT. Hence, if the transaction later submits an update operation on the same objects, no other transaction can write the object in between. We call these mechanisms cursor stability in analogy to the discussion in Berenson, Bernstein, Gray, Melton, O'Neil, and O'Neil [1995]. Note that write locks are still long, avoiding $P1$. Lost updates ($P2$) can be avoided by using cursors or the "FOR UPDATE" clause. Cases $P3$ to $P5$ might occur.

Snapshot Isolation is a way to eliminate read locks completely [Berenson et al. 1995]. Transactions read consistent snapshots from the log instead of from the current database. The updates of a transaction are integrated into the snapshot. Snapshot isolation uses object versions to provide individual snapshots. Each object version is labeled with the transaction 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. This version is reconstructed applying *undo* to the actual version of X until the requested version is generated. Before a transaction writes an object X, it performs a version check. If X was updated after T started, T will be aborted. This feature is called *first committer wins*. Snapshot isolation avoids $P1 - P4$ but allows $P5$. Snapshot isolation has been provided by Oracle since version 7.3 [Oracle 1995]. Note that snapshot isolation avoids all inconsistencies described in the ANSI SQL standard although formally it does not provide serializable histories.

Hybrid Protocol combines 2-phase-locking and snapshot isolation by requiring that update transactions acquire long read and write locks. Read-only transactions use a snapshot of the database. This protocol, unlike cursor stability and snapshot isolation, provides full serializability but requires update transactions to be distinguished from queries.

4.4 Replica Control Model

In what follows, we will use a variation of the *read-one/write-all-available* (ROWAA) approach. All the updates of a transaction will be grouped in a single message, and the total order of the communication system will be used to order the transactions.

The execution of a transaction can be summarized as follows. A transaction T_i can be submitted to any node N in the system. T_i is *local* at N and N is the owner of T_i. For all other nodes, T_i is a *remote* transaction. All read operations of T_i are performed on the local replicas of N. The write operations are deferred until all read operations have been executed and bundled into a single *write set message* WS_i. If the updates are necessary for consecutive read operations, they can be performed on private copies. When the transaction wants to commit, the write set is sent to all available nodes (including the local node) using the total order. This total order is used to determine the serialization order whenever transactions conflict. Upon receiving the write sets, each transaction manager performs a test checking for read/write conflicts and takes appropiate actions. Furthermore, it orders conflicting write operations according to the total order determined by the communication system. This is done by requesting write locks for all write operations in a write set WS_i in an atomic step before the next write set is processed. The processing of lock requests in the order of message delivery guarantees that conflicting write operations are ordered in the same way at all sites. A transaction will only start the execution of an operation on an object after all previous conflicting operations have been executed. Note that only the process of requesting locks is serialized but not the execution of the transactions. As long as operations of successive transactions do not conflict, their execution may interleave.

All the proposed protocols serialize write/write conflicts according to the total order. The total order we use includes causality, in order to account for situations that arise when failures occur. The differences between the protocols arise from how they check for read/write conflicts and how they handle them. An important characteristic of all these new protocols is that the total order together with a careful treatment of read operations make a 2-phase-commit protocol unnecessary.

4.5 Comparison with Related Work

The model we use differs significantly from quorum-based protocols where individual messages are sent for each single update and transaction ordering is provided by distributed 2-phase-locking. It is also quite different to the epidemic approach proposed in Agrawal, El Abbadi, and Steinke [1997]. Whereas we simply use the total order in a modular way, Agrawal, El Abbadi, and Steinke [1997] enhances the causal order of the communication system to ensure serializability. As a result, their approach needs a 2-phase-commit and requires reads to be made visible at all sites.

The combination of eager and lazy replication proposed in Breitbart and Korth [1997] and Anderson, Breitbart, Korth, and Wool [1998] has certain similarities with our solution. In both approaches, transactions are first executed locally at a single site but communication takes place before the transaction commits in order to determine the serialization order. In both cases, the local site can commit a transaction before the other sites have applied the updates. In our approach, the local site can commit the transaction once the position in the total order is determined and relies on the fact that the remote sites will use the same total order as serialization order; in Breitbart and Korth and Anderson et al. updates are sent only after the commit. Neither approach needs a two-phase-commit. However, there are also some significant differences. Breitbart and Korth and Anderson et al. send "serialization messages" and use distributed locking or a global serialization graph to determine the serialization order. Updates are sent in a separate message. We use the delivery order of the write set messages to determine the serialization order. Unlike our approach, two of the mechanisms proposed by Breitbart and Korth and Anderson et al. exchange serialization messages for each operation. The third algorithm is more similar to ours in that it first executes the transaction locally and only checks at commit time whether the global execution is serializable.

5. DATABASE REPLICATION PROTOCOLS

With the previous ideas, the different protocols we propose are as follows (failures are discussed in Section 6).

5.1 Replication with Serializability (SER)

We construct a 1-copy-serializable protocol by using a replicated version of strict 2-phase-locking [Agrawal et al. 1997; Kemme and Alonso 1998]. Whenever a write set is received, a conflict test checks for read/write conflicts between local transactions and the received write set. If the write set intersects with the read set of a concurrent local transaction, the reading transaction is aborted. The protocol is shown in Figure 2 and it can be best explained through the example of Figure 3. (Note that we refer to the copies of objects associated with N simply as X and omit the index.)

The vertical lines in Figure 3 show a run at the two nodes N_1 and N_2. Both nodes have stored copies of objects X and Y. T_1 reads X and then writes Y. T_2 reads Y and then writes X. Both transaction are submitted at around the same time and start the local reads setting the corresponding read locks (reading phase 1 in Figure 2). After the reading phase, both transactions multicast independently their write sets WS_1 and WS_2 (send phase 2). The communication system orders WS_1

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

(1) *Local Reading Phase:* Acquire local read lock for each read operation $r_i(X)$. Defer write requests $w_i(X)$ until end of reading phase.

(2) *Send Phase:* If T_i is read-only, then commit. Else bundle all writes in WS_i and multicast it (total order service).

(3) *Lock Phase:* Upon delivery of WS_i, request all locks for WS_i in an atomic step:
   (a) For each operation $w_i(X)$ in WS_i:
      i. If there is no lock on X, grant the lock.
      ii. If there is a write lock on X or all read locks on X are from transactions that have already processed their lock phase, then enqueue the lock request.
      iii. 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 multicast abort message a_j (basic service).
   (b) If T_i is a local transaction, multicast commit message c_i (basic service).

(4) *Write Phase:* Whenever a write lock is granted perform the corresponding operation.

(5) *Termination Phase:*
   (a) *Upon delivery of a commit message c_i:* Whenever all operations have been executed commit T_i and release all locks.
   (b) *Upon delivery of an abort message a_i:* Undo all operations already executed and release all locks.

Fig. 2. Replication protocol guaranteeing serializability (SER)

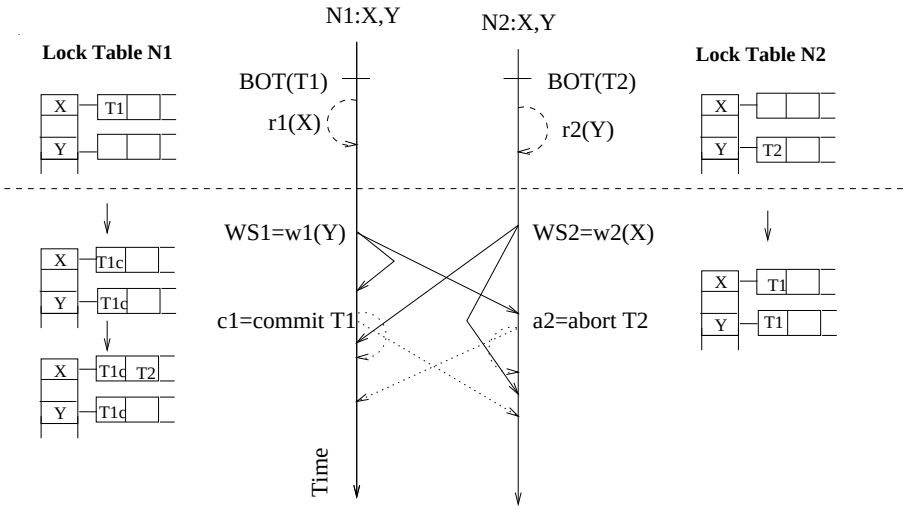

Fig. 3. Example execution with the SER protocol

before WS_2. We first look at node N_1. When WS_1 is delivered, the lock manager first requests all locks for WS_1 in an atomic step (lock phase 3). Since no conflicting locks are active, the write lock on Y is granted (3.a.i). From herein, the system will not abort the transaction. This is denoted with the c attached to the lock entries of T_1. The commit message c_1 is multicast without any ordering requirement (3.b) and then the operation is executed (write phase 4). T_1 can commit on N_1 once c_1 is delivered and all operations have been performed (termination phase 5.a). When WS_2 is delivered, the lock manager processes the lock phase of WS_2. The lock for X must wait (3.a.ii). However, it is assured that it only must wait a finite time since T_1 has already successfully received all necessary locks and can therefore commit and release the locks (note that the lock phase ends when the lock is included in the queue). We now have a look at node N_2. The lock manager also first requests all locks for WS_1 (lock phase 3). When requesting the lock for $w_1(Y)$ the lock manager encounters a read lock of T_2 (conflict test). Since T_2 is still in send phase (WS_2 has not yet been delivered) T_2 is aborted and the lock is granted to T_1 (3.a.iii). If T_2 were not aborted but instead $w_1(Y)$ waited for T_2 to finish, we would observe a write skew phenomenon (on N_1 there is $r_1(X) <_H w_2(X)$, on N_2 there is $r_2(Y) <_H w_1(Y)$). This results in a non-serializable execution that is not locally seen by any node. To avoid this problem, N_2 aborts its local transaction T_2. Since WS_2 was already multicast, N_2 sends an abort message a_2 (no ordering required). When the lock manager of N_2 now receives WS_2, it simply ignores it. Once a lock managers receives the decision (commit/abort) message for T_1/T_2 it can terminate the transaction accordingly (termination phase 5.a, 5.b).

The protocol is deadlock free, 1-copy-serializable, and guarantees transaction atomicity, i.e., if a node commits/aborts a local update transaction T then all nodes commit/abort T. The proof of correctness of this protocol can be found in Appendix A.

In this protocol, the execution of a transaction T_i requires two messages. One to multicast the write set (using the total order) and another with the decision to abort or commit (using the simple order). The second message is necessary since only the owner of T_i knows about the read operations of T_i and, therefore, about a possible abort of T_i. Once the owner of T_i has processed the lock phase for WS_i, T_i will commit as soon as the commit message arrives. Due to the actions initiated during the lock phase, it is guaranteed that remote nodes can obey the commit/abort decision of the owner of a transaction. When granting waiting locks, write locks should be given preference over read locks to avoid unnecessary aborts.

Several mechanisms could be used to avoid aborting the reading transaction. One possibility is to abort the writer instead of the reader. However, this would require a 2-phase-commit protocol since a writer could be aborted anywhere in the system. This is exactly what our approach tries to avoid by enabling each site to independently decide on the outcome of its local transactions. A second option would be to use traditional 2-phase-locking, i.e., the write operation has to wait until the reading transaction terminates. This approach creates deadlocks and is therefore not desirable. The third alternative is that each node informs the other nodes about local read operations so that each site can check individually whether read/write conflicts lead to non-serializable executions. The only way to do this efficiently would be to send information about the read set together with the write

(1) *Local Reading Phase:* Acquire local read lock for each read operation $r_i(X)$. In case of short locks (i.e., T_i will not update X later on), release the lock after the execution of the operation, otherwise keep it. Defer write requests $w_i(X)$ until end of reading phase.
(3.a.ii) If there is write lock on X or all read locks on X are either short or from transactions that have already processed their lock phase, then enqueue the lock request.

Fig. 4. Protocol changes: cursor stability (CS)

set. This information, however, is rather complex since it must not only contain the identifiers of all read objects but also which transactions had written the versions that were read. As a result, performance would be seriously affected.

To summarize, aborting readers is expensive but it provides 1-copy-serializability, avoids distributed deadlocks, keeps reads local, and avoids having to use 2-phase-commit. However, it might pay off not to abort read operations. The next two protocols explore this approach.

5.2 Replication with Cursor Stability (CS)

The weak point of the SER protocol is that it aborts read operations when they conflict with writes. The protocol may even lead to starvation of reading transactions if they are continuously aborted. A simple and widely used solution to this problem is cursor stability; this allows the early release of read locks. In this way, read operations will not be affected too much by writes, although the resulting execution may not be serializable.

The algorithm described in the previous section can be extended in a straightforward way to include short read locks. Figure 4 shows the steps that need to be changed. Step (1) now requires short read locks to be released immediately after the object is read, Hence, the modified step (3.a.ii) shows that the protocol does not need to abort upon read/write conflicts with short read locks since it is guaranteed that the write operation only waits a finite time for the lock. Upon read/write conflicts with long read locks aborts are still necessary. Note that when granting waiting locks, short read locks need not be delayed when write sets arrive but can be granted in the order they are requested since they do not lead to an abort. How far the protocol can really avoid aborts depends on the relation between short and long read locks; this is strongly application dependent. Like the SER protocol, the CS protocol requires two messages: the write set sent via a total order multicast and the decision message using a simple multicast.

The CS protocol fulfills the atomicity requirement of a transaction in the same way as the SER protocol does. It does not provide serializability but avoids $P1$ and $P2$. $P3$, $P4$ and $P5$ may occur. The proof can be found in appendix A.

5.3 Replication with Snapshot Isolation (SI)

Although cursor stability solves the abort problem of SER, it generates inconsistencies. The problem is that read/write conflicts are difficult to handle since they are only visible at one site and not in the entire system. Snapshot isolation effectively separates read and write operations thereby avoiding read/write conflicts entirely. This has the advantage of allowing queries (read-only transactions) to be performed without interfering with updates. In fact, since queries do not need to be aware

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

(1) *Local Reading Phase:* For each read operation $r_i(X)$, 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) *Send Phase:* If T_i is read-only, then commit. Else bundle all writes in WS_i and multicast it (total order service). The WS_i message also contains the $TS_i(BOT)$ timestamp.

(3) *Lock and Version Check Phase:* Upon delivery of WS_i, perform in an atomic step: For each operation $w_i(X)$ in WS_i:
   (a) If there is no write lock on X and the current version of X is labeled with T_j. Then, if $TS_j(EOT) > TS_i(BOT)$, stop checking locks and abort T_i. Otherwise grant the lock.
   (b) If there is a write lock on X or a write lock is waiting. Then let T_j 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 enqueue the lock request.

(4) *Write Phase:* Whenever a write lock is granted, perform the corresponding operation.

(5) *Commit Phase:* Whenever all operations have been executed, commit T_i and release all locks.

Fig. 5. Replication protocol based on snapshot isolation (SI)

of replication at all, the replication protocol based on snapshot isolation is only concerned with transactions performing updates.

In order to enforce the "first committer wins" rule, as well as to give appropriate snapshots to the readers, object versions must be labeled with transactions and transactions must be tagged with BOT (begin of transaction) and EOT (end of transaction) timestamps. The BOT timestamp determines which snapshot to access and does not need to be unique. The EOT timestamp indicates which transaction did what changes (created which object versions), and hence, must be unique. Oracle [Bridge et al. 1997] timestamps transactions using a counter of committed transactions. In a distributed environment, the difficulty is that the timestamps must be consistent at all sites. To achieve this, we use the sequence numbers of WS messages. Since write sets are delivered in the same order at all sites the sequence number of a write set is easy to determine, unique and identical across the system. Therefore, we set the EOT timestamp $TS_i(EOT)$ of transaction T_i to be the sequence number of its write set WS_i. The BOT timestamp $TS_i(BOT)$ 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 terminated. It is possible for transactions with higher sequence numbers to have committed but their changes will not be visible until all preceding transactions (with a lower message sequence number) have terminated.

Figure 5 describes the algorithm for replication with snapshot isolation. We assume the amount of work to be done for version reconstruction in the reading phase (1) is small. Either this version is still available (Oracle, for instance, maintains several versions of an object [Bridge et al. 1997]) or it can be reconstructed by using a copy of the current version and applying undo until the requested version is generated. Oracle provides special rollback segments in main-memory to provide efficient undo [Bridge et al. 1997] and we will assume a similar mechanism is available. The lock phase includes a version check (3). If $w_i(X) \in WS_i$, and X was

updated by another transaction since T_i started, T_i will be aborted. We assume the version check to be a fast operation (the check occurs only for those objects that are updated by the transaction). In addition, to reduce the overhead in case of frequent aborts, a node can do a preliminary check on each local transaction T_i before it sends WS_i. If there already exists a write conflict with another transaction, T_i can be aborted and restarted locally. However, the check must be repeated upon reception of WS_i on each node.

With this algorithm, each node can decide locally, without having to communicate with other nodes, whether a transaction will be committed or aborted at all nodes. No extra decision message is necessary since conflicts only exist between write operations. The write set is the only message to be multicast.

Note that while serializability aborts readers when a conflicting write arrives, snapshot isolation aborts all but one concurrent writers accessing the same item. 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. However, snapshot isolation has some other advantages compared to serializability or cursor stability. It only requires a single multicast message to be sent and has the property that read operations do not interfere with write operations.

The SI protocol guarantees the atomicity of transactions. However, it does not provide serializability. It avoids $P1 - P4$ but $P5$ might occur. The proof can be found in appendix A.

5.4 Hybrid Protocol

Snapshot isolation provides queries with a consistent view of the database. However, update transactions are not serializable. Moreover, if objects are updated frequently the aborts of concurrent writes might significantly affect performance. Long transactions also suffer under the first committer wins strategy. To avoid such cases, a hybrid approach could use the protocol for full serializability for update transactions and snapshot isolation for queries. This combination provides full serializability. However, to be able to decide which protocol must be applied, transactions must be declared read-only or update in advance. In addition, both approaches must be implemented simultaneously leading to more administrative overhead. Both update transactions and objects must receive timestamps to be able to reconstruct snapshots for the read-only transactions. Additionally, the lock manager must be able to handle the read locks of the update transactions. This overhead might be justified, since a replicated database makes sense only for read intensive applications.

6. FAULT-TOLERANCE

Group communication and databases follow different approaches to fault-tolerance. Their solutions must be modified to make them fit together [Guerraoui and Schiper 1995]. Following accepted practice in databases, we will not only look at solutions that provide full correctness and consistency, but also propose best effort approaches which may lead to inconsistencies in case of failures but provide better performance. As with the different isolation levels, this is a pragmatic solution based on a typical trade off between introducing a few inconsistencies in case of failures and being able to process most transactions as fast as possible.

In our context, we assume a crash failure model [Neiger and Toueg 1988]. A node runs correctly until it fails. From then on, it does not take any additional processing steps until it recovers. The system is asynchronous, i.e., different sites may run at different rates, and the delay of messages is unknown, possibly varying from one message to the next.

6.1 Failure Handling in Group Communication Systems

Group communication systems handle failures by providing *group maintenance services* and different degrees of *reliability*. Node failures (and communication failures) lead to the exclusion of the unreachable nodes and are mainly detected using time-out protocols [Chandra and Toueg 1991]. We will assume that only a primary partition may continue to work while the other partitions stop working and, hence, behave like failed nodes. The application is provided with a *virtual synchronous* view of failure events in the system [Birman et al. 1991].

To do so, the communication module of each site maintains a view V_i of the current members of the group and each message is sent with respect to this view. Whenever the group communication system observes the failure of one or more of the members, it runs a coordination protocol called *view change protocol*. This protocol guarantees that the communication system will deliver exactly the same messages at all non-failed members. Only then is the new view which excludes the failed nodes installed and the application is informed via a so called *view change message*. Hence, the application programs on the different sites perceive process failures at the same virtual time. Note that, since we do not allow partitions, all nodes of view V_i change to the same consecutive view V_{i+1} unless they fail.

Using virtual synchronous communication, the database system at each node N generates a sequence of communication events. An event is either *sending a message* m, *receiving a message* m (this is also denoted as message m *is delivered* at N) or receiving a *view change* notification consisting of a new view V_i. Without loss of generality we assume the first and last event to be view changes. A run R_i of the database system at node N, $N \in V_i$, is the sequence of communication events at N starting with view V_i and ending either with consecutive view V_{i+1} (we then say N is V_i-available) or the failure of N (we then say N fails during V_i or that N is faulty in V_i). Virtual synchrony provides the following guarantees:

(1) *View Synchrony*: if a message m sent by node N in view V_i is delivered at node N', then N' receives m during R_i.

(2) *Liveness:* if a message m is sent by node N in view V_i and N is V_i-available, then m is delivered at all V_i-available nodes.

(3) *Reliability:* Two degrees of reliability are provided: Let m be a message sent by node N in view V_i.
   (a) *Reliable delivery*: if m is delivered at node N' and N' is V_i-available, then m is delivered at all V_i-available nodes.
   (b) *Uniform reliable delivery*: if m is delivered at any node N' (i.e., N' either is V_i-available or fails during V_i), then m is delivered at all V_i-available nodes.

Uniform reliability and (simple) reliability differ in the messages that might be delivered at failed nodes. With uniform reliable delivery, a node N receives the

same messages as all other nodes until it fails [Schiper and Sandoz 1993]. In this case the set of messages delivered at a failed node N is a subset of the messages delivered at the surviving nodes (in the case of total order, it is a prefix). With reliable delivery failed nodes might receive (and process) messages that no other node receives. In any case, the ordering of messages must be correct - within a single run R_i and across view changes.

The view change protocol must be able to exchange messages to guarantee their delivery at all non-faulty sites. To do so, the communication module of each node keeps messages until they are *stable*, i.e., until each other node in the group has acknowledged their reception from the network. In a simple view change protocol, each remaining node would send all *unstable* messages to all non-failed nodes (called *flush*).

The degree of reliability has a big impact on the message delay during normal processing. Uniform reliable delivery cannot deliver a message to the application until it is stable, hence, delaying the message for an additional message round. This is similar to a 2-phase-commit, although it does not involve a voting process. Using reliable delivery, a message can be delivered as soon as all preceding messages have been delivered. For example, a simple multicast message can be delivered immediately after its reception from the network since it is not ordered with respect to any other message.

Numerous view change protocols have been proposed which always work with specific implementations of the causal and total order services and provide different degrees of reliability [Birman et al. 1991; Schiper and Sandoz 1993; Moser et al. 1994; Friedman and van Renesse 1995b]. Since the view change might introduce a significant overhead, these systems only work when failures occur at a lower rate than messages exchange. Inter-failure intervals in the range of minutes or hours are, however, acceptable.

6.2 Failure Handling in Database Systems

Databases deal with failures based on the notion of atomicity and consistency.

Atomicity implies that a transaction must either commit or abort at all nodes. Therefore, after the detection of a failure, the remaining nodes have to agree on what to do with pending transactions. This usually requires to execute a *termination protocol* among the remaining nodes before processing can be resumed [Bernstein et al. 1987]. The termination protocol has to guarantee that, for each transaction the failed node might have committed/aborted, the remaining available nodes decide on the same outcome. This approach is very similar to the view change protocol in distributed systems. Whereas replicated database systems decide on the outcome of pending transactions, group communication systems decide on the delivery of pending messages. Hence, comparing typical protocols in both areas [Bernstein et al. 1987; Friedman and van Renesse 1995b], we believe the overhead to be similar. A significant difference is that database systems decide negatively – abort everything that does not need to be committed – while group communication systems behave positively – deliver as many messages as possible. In what follows, we will look at how the view change protocol can assume the tasks of the termination protocol.

The replicated database is *consistent* if all replicas of an object residing on non-faulty nodes converge to the same value. This is achieved in part by the atomicity

of transactions: all sites commit the updates of exactly the same transactions (unless they fail). In the next subsections we will show that all proposed algorithms guarantee transaction atomicity on all non-faulty nodes. Furthermore, using the total order, all these updates are applied in the same order at all sites. Hence, consistency among non-faulty nodes is provided.

In what follows we combine the protocols presented in section 5, with both uniform reliable and reliable message delivery. For each of the possible combinations we will discuss what needs to be done when failures occur and which atomicity guarantees can be provided. Appendix B contains the detailed theorems and proofs. In the following we look at the run R_i at each node N, i.e., from installing view V_i until installing V_{i+1} or the failure of N. The extension to an entire execution is straightforward.

6.3 Snapshot Isolation (SI)

For nodes that do not fail, there is no difference between reliable and uniform reliable delivery using SI. For both types of delivery, when a node N is in view V_i and receives view change message V_{i+1}, it simply continues processing transactions unless the group change is due to a network partition that would disallow the current group of N to continue processing transactions. Let $\mathcal{T}$ be the set of transactions whose write sets have been sent in view V_i.

THEOREM 1. *The SI protocol, either with reliable or uniform reliable delivery, guarantees atomicity of all transactions in $\mathcal{T}$ on all V_i-available nodes.*

PROOF. (Proof Sketch) Both reliable and uniform reliable delivery (guarantee 3) together with the total order of WS messages and the view synchrony (1) guarantee that the database modules of V_i-available nodes receive exactly the same prefix of WS messages before receiving view change V_{i+1}. Furthermore, liveness (2) guarantees that write sets of V_i-available nodes will always be delivered. Thus, within this group of available nodes (excluding failed nodes), we have the same delivery characteristics as we have in a system without failures. Hence we can rely on the atomicity guarantee of the SI protocol in the failure-free case. □

Uniform reliable and reliable delivery for SI differ in the set of transactions committed at failed nodes and, consequently, in what must be done when nodes recover.

With *uniform reliable message delivery* a node failing during V_i delivers a prefix of the sequence of messages delivered by a V_i-available node. Thus, the behavior of an available node and a failed node is the same until the failure occurs and the failed node simply stops. From here, it is easy to see that atomicity of all transactions is guaranteed on all nodes, including failed nodes (see theorem 9 in appendix B).

If *reliable message delivery* is used, a failed node might commit a transaction that the available nodes do not commit. This can happen since a failed site might have delivered a write set before failing but none of the other sites deliver this write set. For instance, with the Totem [Moser et al. 1996] protocol, a node with the token could send the write set of a transaction, commit it locally and then fail. If the message is lost, no other site will see this transaction. Note that this scenario cannot occur with uniform reliable delivery since the database will not commit the transaction until it has been received at all sites. As a result, when performing recovery, such spurious transactions must be reconciled.

6.4 Serializability (SER), Cursor Stability (CS) or Hybrid

Similar to SI, for the SER, CS and Hybrid protocols there is no difference on available nodes between reliable and uniform reliable delivery. Unlike SI, however, only the owner of a transaction can decide on the outcome of the transaction. Therefore, a failed node can leave *in-doubt* transactions in the system:

Definition 1. Let N be a node failing during view V_i and T a transaction invoked at node N whose write set WS has been sent before or in V_i. T is *in-doubt in* V_i if the V_i-available nodes receive WS but not the corresponding decision message (commit/abort) before installing view V_{i+1}.

From here we can derive the set of transactions for which the outcome is determined before view V_{i+1}. Let $\mathcal{T}_1$ be the set of transactions which are in-doubt in V_i. $\mathcal{T}_2$ is the set of transactions that have been invoked at a V_i-available node and both write set and decision message (commit/abort) have been sent before or in view V_i. Finally, $\mathcal{T}_3$ is the set of transactions that have been invoked at a node failing during V_i and both write set and decision message have been received by at least one V_i-available node before N fails. Let $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2 \cup \mathcal{T}_3$.

THEOREM 2. *The SER, CS and Hybrid protocols either with reliable or with uniform reliable delivery, guarantee transaction atomicity of all transactions in* $\mathcal{T}$ *on all* V_i*-available nodes.*

PROOF. (Proof Sketch) For both reliable and uniform reliable delivery, all sites available during V_i receive exactly the same messages (and the write sets in the same order) before installing view change V_{i+1}. Hence, they all have exactly the same set of in-doubt transactions and the same set of transactions for which they have received both the write set and the decision message. If they decide on the same outcome for in-doubt transactions, they have the same behavior as a system without failures. Hence, we apply the atomicity property of the SER/CS/Hybrid protocol to guarantee transaction atomicity on all available nodes and with it database consistency. □

Note that there are two types of transactions that are not in $\mathcal{T}$. The first type is transactions of available nodes where the decision message was not sent in V_i. For this group, the decision will simply be made in the next view. The second group is the transactions invoked at a node failing during V_i where the write set was sent but not received at any available site. In the following we refer to this group as $\mathcal{G}$. These are the transactions that will be handled differently depending on whether reliable or uniform reliable delivery is used. As with SI, the reliability of message delivery has an impact on the set of transactions committed at failed nodes and hence, determines whether transaction atomicity is also guaranteed on failed nodes.

If *both the write set and the commit message are uniform reliable*, transaction atomicity is guaranteed in the entire system (including failed nodes). Upon reception of the view change, all available nodes will have the same in-doubt transactions which are safe to abort. These transactions are either active or aborted on the node that failed but never committed, since uniform delivery guarantees that all or no node receives the commit. Furthermore, transactions from $\mathcal{G}$ (that are not visible at available nodes) cannot be committed at failed nodes, since this is only possible when both write set and commit message are delivered. The requirement of

uniform reliability does not apply to abort messages, i.e., they can be sent with the reliable service, since the default decision for in-doubt cases is to abort. For more details, see theorem 10 in Appendix B. We call this approach non-blocking since all available nodes decide consistently in a non-blocking way about all pending transactions and the new view can correctly continue processing.

Some of the overhead of uniform delivery can be avoided by risking not being able to reach a decision about in-doubt transactions. This is the case if *the write set WS_i is sent using uniform reliable delivery*, but *both commit and abort messages are sent using reliable delivery.* After the database module receives a view change excluding node N, it will decide to block all the in-doubt transactions of N and then continue processing transactions. A transaction that is in-doubt at the available nodes can be active, committed or aborted at the failed node. Hence, to achieve transaction atomicity on all nodes, in-doubt transactions must be blocked. For all other transactions, transaction atomicity can be guaranteed with the same arguments as before. A more detailed description is given in theorem 11 in Appendix B. This scenario is similar to the blocking behavior of the 2-phase-commit protocol. Note that 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 must wait. In practice, it might be reasonable to abort these transactions but recovery needs to consider transactions committed at failed nodes and aborted elsewhere.

A last possibility sends *all messages using reliable delivery.* In this case a node could commit a transaction from group $\mathcal{G}$ locally before it is seen by any other node. Consequently, failed nodes may have committed transactions that are non-existing at the other sites and must be reconciled upon recovery. Hence, transaction atomicity cannot be guaranteed on all nodes. However, atomicity and consistency are preserved within the available nodes. As in the previous case, in-doubt transactions can be blocked or aborted.

6.5 Recovery

Upon recovery, the database at the recovering node must be identical to the databases in the rest of the system before the node can start executing transactions. This is also true when a completely new node is added. A recovering node needs to perform a standard undo/redo recovery [Bernstein et al. 1987] and must contact a peer node to obtain the current state of the database. To do this, there exist two possibilities. The peer node can either send an entire copy of the database or it can send the part of its log which contains the transactions executed after the recovering node failed. Which option is best depends on the size of the database and the number of transactions executed during the down-time of the recovering node.

The advantage of the view change protocol is that it provides checkpoints that can be used to determine what a recovering node needs to do. First, there is a last view change V_i being seen by a node before it failed (i.e., it failed during V_{i+1}). Second, when a failed node rejoins the system a new view change V_j is triggered. From then on the recovering node receives all new transactions. Thus, the recovering node can restore the state of the database at the time of view change V_i and replay the log

of the peer node which includes all transactions received after V_i and before V_j. From herein it can execute all transactions received in the new V_j and start its own transactions.

Care has to be taken when the system only uses reliable message delivery. In this case, alluded to before, failed nodes might have committed transactions that have not been seen by other nodes. These updates must be either reconciled or undone in the recovering node.

7. SIMULATION MODEL

The performance of the proposed protocols has been evaluated using similar techniques to those used in other studies of concurrency control [Agrawal et al. 1987; Carey and Livny 1989; Gupta et al. 1997]. The simulation parameters are summarized in Table 2.

The database is modeled as a collection of *DBSize* objects where each of the *NumSite* sites stores copies of all objects. Each site consists of *NumCPU* processors and *NumDisks* data disks. All disks and processors have their own request queues; these are processed in FCFS order. The *ObjectAccessTime* and *DiskAccessTime* parameters capture the time needed to perform an operation on an object and to fetch an object from the disk. The *BufferHitRatio* indicates the percentage of operations on data residing in main memory.

Communication overhead is modeled by several parameters. Each physical message transfer has an overhead of *SendMsgTime* CPU time at the sending site and *RcvMsgTime* CPU time at the receiving site. The times may differ for different message sizes. Furthermore, we assume a time overhead of *BasicNetworkDelay* (for delays taking place at the IP level and lower). Network utilization is calculated by the size of the message and the bandwidth of the network *NetBW*. *MsgLossRate* is the percentage of physical messages that are lost.

We assume a broadcast medium where a message to a group of sites only requires a single physical message. Still, a logical multicast message might involve more than one physical message (handling message loss, total order etc.). Message loss is detected by a combination of acknowledgments and negative acknowledgments similar to the TCP/IP protocol. Upon receiving a multicast from the network, an acknowledgment is multicast. When a gap in the sequence numbers of messages is detected a negative acknowledgment is sent to the sender which then resends the message via a point-to-point message. For the basic service (no order), a single multicast is sent. Using reliable delivery, a message is delivered as soon as it is received from the network. Using uniform reliable delivery a message is delivered once all acknowledgments have been received. The total order uses the Totem algorithm and our implementation follows the description in Moser et al. [1996]: A token is passed through the system and only the token owner may send messages. Acknowledgments are piggybacked on the token. In the case of reliable multicast, a message is delivered immediately after all preceding messages have been delivered. Uniform reliable message delivery additionally waits until all nodes have acknowledged the reception of the message. We model point-to-point communication similar to TCP/IP. Note that we also implemented reliable communication for point-to-point communication, i.e., message loss is always detected in the communication system and lost messages are resent.

Table 2. Simulation Model Parameters

General	*NumSite*	Number of sites in the system
	DBSize	Number of objects of the database
Database	*NumCPU*	Number of processors per site
	NumDisks	Number of disks per site
	ObjectAccessTime	CPU object processing time
	DiskAccessTime	Disk access time
	BufferHitRatio	% of object accesses that do not need disk access
Communication	*SendMsgTime*	CPU overhead to send message
	RcvMsgTime	CPU overhead to receive message
	BasicNetworkDelay	Basic delay for IP level and lower
	NetBW	Network Bandwidth
	MsgLossRate	Message Loss Rate
Transaction Type	*TransSize*	Number of op. of a transaction
	WriteAccessPerc	% of write op. of the transaction type
	RWDepPerc	% of read op. on objects that will be written later
	TransTypePerc	% of the workload that belongs to this type
	Timeout	Timeout for the distributed 2PL algorithm
Concurrency	*InterArrival*	Average time between the arrival of two transactions at one node

We distinguish between different transaction types where each type is determined by a number of parameters. Each transaction performs *TransSize* operations. We distinguish between read and write operations. *WriteAccessPerc* is the percentage of write operations of a transaction. *RWDepPerc* (ReadWriteDependency) determines the percentage of read operations on objects that will be written later. These require read locks that are kept until the end of the transaction even when using cursor stability. If *WriteAccessPerc* is zero, the transaction type describes read-only transactions that can be performed locally. *TransTypePerc* gives the percentage of the workload that belongs to this transaction type. The objects accessed by a transaction are chosen randomly from the database.

Transaction execution and concurrency control are modeled according to the algorithms described in the previous sections. When a transaction is initiated, it is assigned to one node. All read operations are performed sequentially at that node. At the end of the transaction, the write set is sent to all nodes. For comparison purposes, we have also implemented a traditional distributed locking protocol using ROWAA with strict 2-phase-locking (2PL). In this case, each read operation is executed locally and each write operation is multicast to all sites using the simple reliable multicast. At the remote sites, whenever the lock for the operation is acquired, an acknowledgment is sent back (note that this is an optimization to the standard protocol where the acknowledgment is not sent before the entire operation is executed). When the local site has received all acknowledgments and executed the operation the next operation can start. Whenever a deadlock is detected at a site, a negative acknowledgment is sent back to the local node which, in turn, will multicast an abort message to the remote nodes. When all operations are successfully executed the local site sends a commit message to the remote sites.

To deal with the deadlock behavior of 2PL, we have implemented two versions of distributed deadlock detection. As a first possibility, distributed deadlocks are

detected via timeout as it is usually done in commercial systems. The parameter *Timeout* sets the timeout interval. As a second possibility, we have implemented a global deadlock detection mechanism. Whenever a node requests a lock and the lock must wait, the detection algorithm is run. In our simulation, this algorithm is "ideal" in the sense that no message delay or CPU overhead is associated with it.

Each operation includes sometimes a disk I/O to read the object from the disk and a subsequent period of CPU usage for processing the object access. A transaction that is aborted is restarted immediately and makes the same data accesses as its original incarnation. We use an open queuing model. At each node, transactions are started according to an exponential arrival distribution with a mean determined by *InterArrival*. The *InterArrival* parameter determines the throughput (transactions per second) in the system (e.g., small arrival times lead to high throughput).

8. EXPERIMENTS AND RESULTS

We have conducted an extensive set of simulation experiments analyzing the behavior of the protocols and systems with respect to the settings of various parameters. We present a comprehensive summary of the most important results showing the general behavior and most relevant differences between the protocols. Some of the parameters are fixed for all experiments discussed since their variation led only to changes in absolute but not in relative behavior of the protocols. Their baseline settings are shown in Table 3.

The database consists of 10,000 objects. The number of processors per site is two: one for transaction processing and one for communication. We use a dedicated processor for communication in order to differentiate between transaction processing and communication overhead. The number of disks is 10. CPU time per operation is 0.2 ms and disk access takes 20 ms. The buffer hit ratio is fixed to 80% and the network has a bandwidth of 100Mb/s. Note that bandwidth was never a limiting factor in our experiments. This matches results from previous studies [Friedman and van Renesse 1995a; Moser et al. 1996]. Furthermore, we assume a 2% message loss rate for all of our experiments.

We use three different transaction types. Two of these are update transactions: *short* transactions, consisting of 10 operations, and *long* transactions, consisting of 30 operations. Both types have an update rate of 40% and a read/write dependency of 30%. Short transactions represent a workload with low data contention (conflict rate), whereas long transactions show higher data contention. The third transaction type is a read-only transaction with 30 operations. The timeout interval for the distributed 2PL algorithm is 1000 ms for short update transactions. For long update transactions and read-only transactions the global deadlock detection mechanism is used. In the following experiments we set the inter-arrival times of transactions according to the transaction type. This is done in such a way that the pure overhead of executing operations (CPU/disk) at each single site is about the same for all experiments and within reasonable boundaries (we did not want the transaction processing CPU/disk to be the bottleneck resource). For instance, since read-only transactions are only executed locally while the write operations of update transactions are executed everywhere, a system can achieve a higher throughput with read-only transactions. Hence, in experiment 2, we set the inter-arrival times of a mixed read-only/update workload (80 ms) smaller than for the long update

Table 3. (a) Baseline parameter settings and (b) transaction types

DBSize	10000
NumCPU	2
NumDisks	10
ObjectAccessTime	0.2 ms
DiskAccessTime	20 ms
BufferHitRatio	80%
NetBW	100Mb/s
MsgLossRate	2%

(a)

	Short	Long	Read-only
TransSize	10	30	30
WriteAccessPerc	40%	40%	0%
ReadWritePerc	30%	30%	0%
Timeout	1000 ms	–	–

(b)

transactions (120 ms) in experiment 1.

The main performance metric is the average response time, i.e., the average time a transaction takes from its start until completion. These average response times are with 95% confidence within a 5% interval of the shown results. The response time of a transaction consists of the execution time (CPU + I/O), waiting time and the communication delay. In addition, abort rates of each protocol are also evaluated to provide a more complete picture of the results. In the performance figures and the discussions, the following abbreviations are used: SER for serializability, CS for cursor stability, SI for snapshot isolation and HYB for the hybrid protocol. Furthermore, NBL indicates the non-blocking protocols where all messages (write set WS and decision message c/a) are uniform reliable. BL indicates the blocking protocols where WS is uniform reliable, c/a are reliable (does not exist for SI) and RB refers to the reconciliation based versions where all messages are only reliable.

In all the following figures the order of the labels corresponds to the order of the curves with the label of the highest curve always on the top followed by the label of the second highest curve and so on.

8.1 Experiment 1: Communication Overhead vs. Concurrency Control

The protocols differ in the number of messages and their delay. Whereas the SI protocol uses a single totally ordered message multicast per transaction, the SER/CS/Hybrid protocols send one message using the total order service and one message using the simple order multicast per transaction. The protocols using uniform reliable message delivery suffer from a higher message delay than the protocols using reliable message delay. In addition, the protocols use different concurrency control mechanisms providing different conflict profiles (SER aborts readers, SI aborts writers and CS avoids aborts).

In the first experiment we want to analyze the interplay between these aspects. To do so we vary the communication parameters to model both efficient communication with small message delays and little overhead (typical for group communication systems working in LANs) and slow communication with long delays and high overhead (as in WANs). Using this, we want to analyze the sensitivity of the protocols to the communication overhead. By looking at two different workloads (short and long transactions) we are able to judge which optimization is more effective: reducing message overhead or reducing conflict rates.

Communication is determined by the parameters *BasicNetworkDelay*, *SendMsgTime* and *RcvMsgTime*. Table 4 depicts their settings in ms for this experiment

Table 4. Communication Settings in ms

Test Run	I	II	III	IV	V	VI	VII	VIII	IX
BasicNetworkDelay	0.01	0.05	0.1	0.2	0.4	0.5	0.6	0.8	1.0
SendMsgTime Small Msg.	0.005	0.025	0.05	0.1	0.2	0.25	0.3	0.4	0.5
SendMsgTime Medium Msg.	0.01	0.05	0.1	0.2	0.4	0.5	0.6	0.8	1.0
SendMsgTime Large Msg.	0.02	0.1	0.2	0.4	0.8	1.0	1.2	1.6	2.0
RcvMsgTime Small Msg.	0.01	0.05	0.1	0.2	0.4	0.5	0.6	0.8	1.0
RcvMsgTime Medium Msg.	0.02	0.1	0.2	0.4	0.8	1.0	1.2	1.6	2.0
RcvMsgTime Large Msg.	0.04	0.2	0.4	0.8	1.6	2.0	2.4	3.2	4.0

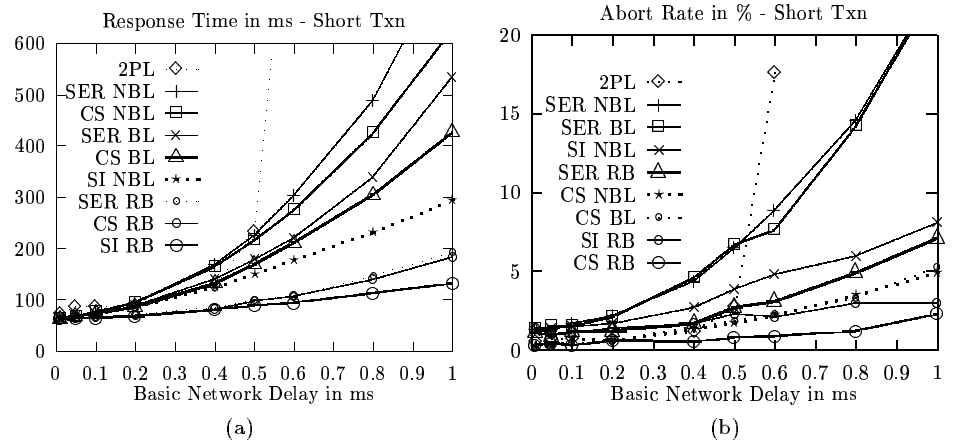

Fig. 6. Experiment 1: *Response time* (a) and *Abort rate* (b) of short transactions

varying them from little to high overhead. Furthermore, we have set the overhead of point-to-point messages to half of the overhead of a multicast message assuming that there is less flow control involved with point-to-point messages. Measurements on real networks using between 6 and 10 nodes show results that are equivalent to ours when we set the parameters to the values of test run IV (resulting in 1 ms for a simple reliable multicast and around 7 ms for reliable, total order). In the performance figures of this experiment we depict the different test runs by using the corresponding setting of the parameter *BasicNetworkDelay*.

Short Transactions. Figure 6 shows the average response times (6.a) and abort rates (6.b) for short transactions for an inter-arrival time of 50 ms per node (i.e. around 200 short transactions per second in the system). At such a workload transaction processing requires few resources and data contention is small. Hence, the message overhead has a greater impact on the overall response time (6.a). With a low communication delay, the response time corresponds to the execution time of the transaction. With an increasing communication overhead, the response time of the different protocols increases depending on the number and complexity of the message exchanges. The communication processors are becoming more and more utilized and are nearly saturated at high communication costs. Hence, when communication is costly and the delay long, the RB protocols show best perfor-

mance (SI outperforming SER and CS since it needs only one multicast message). A little worse is NBL-SI. It performs even better than BL-SER and BL-CS due to the reduced number of messages. Of the suggested protocols, NBL-SER and NBL-CS have the worst behavior since they wait to deliver both the write set and the decision message until all nodes have sent acknowledgments.

2PL has a performance similar to the other protocols when communication is fast. However, when communication is more expensive, 2PL degrades very quickly due to the enormous amount of messages. This saturates the communication processor.

The abort rates (6.b) are generally low for all protocols. 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 and causes transactions to spend more time in the system. As more transactions coexist, the probability of conflicts increases, and with it, the abort rate. Therefore, the RB protocols have a lower abort rate than the NBL and BL versions since they shorten the time from BOT to the delivery of the write set, thereby reducing the conflict profile of the transaction. Generally, since the experiment has a skew towards write operations, CS has the lowest abort rates. SER and SI have similar abort rates with fast communication, but as the communication delay increases the behavior of NBL-SER and BL-SER degrades. This is due the abort of readers 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 since the decision on abort or commit can be done independently at each node and a transaction only acquires locks when it is able to commit. Note also, that the NBL and BL versions of SER (and also the NBL and BL versions of CS) 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 the NBL and BL versions of the protocols.

With low communication costs, 2PL has lower abort rates than SER and SI, since SER and SI sometimes abort readers/writers unnecessarily. However, the abort rates of 2PL quickly degrade when response times become too long due to resource saturation.

Long Transactions. Figure 7 shows the average response times (7.a) and abort rates (7.b) for long transactions for inter-arrival times of 120 ms (i.e. around 80 long transactions per second in the system). Long transactions have higher data contention than short transactions. However, the total number of messages is smaller since less transactions start per time interval. Although the reliability of message delivery is still the dominant factor for high communication costs, the concurrency control method becomes a more important factor in terms of response time (7.a). Looking at the RB protocols, CS outperforms SI and SER due to its low conflict rate. The advantage of SI sending only one message is not the predominant factor, since communication is less saturated and the message delay itself has not a large direct impact on the response time of long transactions. Furthermore, the BL and NBL versions of CS are better than NBL-SI, NBL-SER and BL-SER. The last two protocols do not perform well for slow communication due to the high conflict rate.

Looking at the abort rates (7.b), CS clearly outperforms all other protocols due

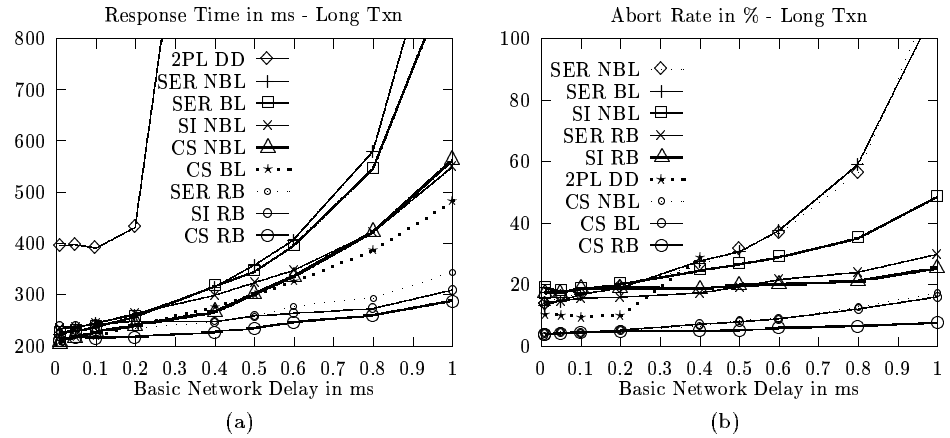

Fig. 7. Experiment 1: *Response time* (a) and *Abort rate* (b) of long transactions

to its low conflict rate. Furthermore there is no degradation when communication becomes slow. NBL-SER, BL-SER and NBL-SI, however, degrade when communication takes longer and here even the RB versions of SER and SI have rather high abort rates. However, these abort rates do not have a large impact on the response time since they usually happen in an early phase of the transaction. Nevertheless, they might cause a problem if the system cannot restart aborted transactions automatically but instead only returns a "transaction failed" notification to the user. If this is the case, CS might be the preferable choice.

The response time of 2PL degrades very fast. Even with fast communication, it behaves significantly worse than the other protocols. The delay created by acquiring locks on all sites increases the conflict rate extremely. Especially problematic are the waiting periods once a transaction has to wait for a lock. All the protocols proposed in this paper avoid this problem – CS and SI mostly avoid read/write conflicts and SER aborts and restarts readers in a very early phase of transaction execution. Note however, that as long as no degradation takes place, the abort rates of 2PL are better than those of SER and SI. However, choosing the right timeout interval for deadlocks is very difficult and we choose to implement a no-cost deadlock detection mechanism (which is unrealistic in a real system) in order to figure out the "true" abort rate.

Analysis. For the proposed protocols the general behavior can be summarized as follows. In "ideal" environments, the behavior of the protocols is very much the same in all cases and serializability and uniform reliable message delivery can be used to provide full consistency and correctness. However, as soon as conflict rates or network delay increase, both serializability and uniform reliable message delivery might be bad choices: they result in increasing abort rates and longer response times. The results show which strategy to follow depending on the characteristics of the system. Thus, 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. Similarly, if data contention is high, lower levels

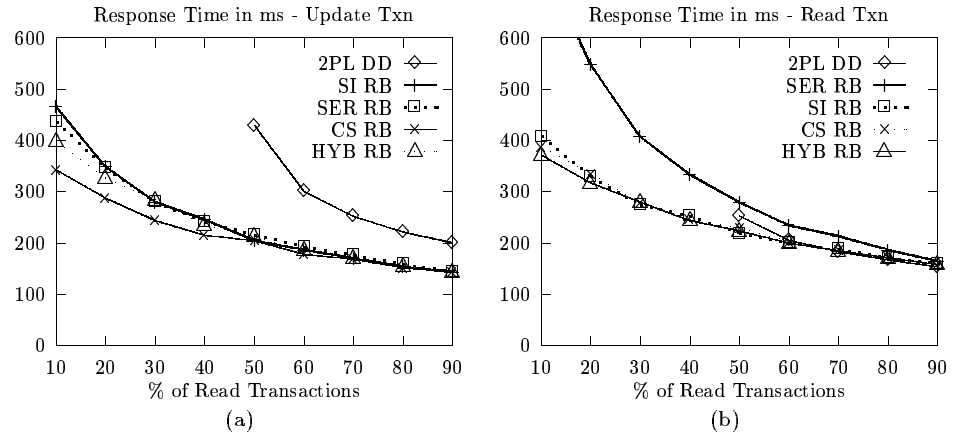

Fig. 8. Experiment 2: Response Time of (a) *update* and (b) *read-only* transactions

of isolation are the only alternative in order to achieve small abort rates.

Generally, 2PL shows worse performance than any of the proposed algorithms. It is very sensitive to the capacity and performance of the communication system, it is not able to handle high conflict rates and it degrades very fast when conditions worsen.

8.2 Experiment 2: Read-Only Transactions

In practice, replication pays off when the majority of the transactions are queries which can be executed locally without any communication costs. We have analyzed the behavior of the protocols using a mixed workload consisting of long update and read-only transactions. The percentage *TransTypePerc* of both transactions types are varied between 10% to 90%. Since we want to investigate pure resource and data contention and not the impact of the communication overhead, the communication costs are set to be low (see values of test run III in Table 4) and we only show the results for the RB protocols. We also analyze the hybrid protocol using SER for updating transactions and a snapshot for read-only transactions.

Figure 8 shows the response time for the update transactions (8.a) and read-only transactions (8.b) as a function of the percentage of read-only transactions in the workload for an inter-arrival time of 80 ms (i.e., around 125 transactions per second in the system). The response times for both transaction types decrease when the percentage of read-only transactions increases due to less resource contention (less replicated write operations, more local read operations) and less data contention (shorter lock waiting times, lower abort rates). The differences in the protocols directly reflect the different abort rates for writers and readers of the different protocols. For update transactions (8.a) CS behaves better than the others for low read-only rates since data contention is rather high and CS has less aborts than the others. SER, HYB and SI behave similarly having very similar abort rates. 2PL does not admit low read-only rates. For read-only rates smaller than 50% the system degrades both for update and read-only transactions. In both cases, the response times of update transactions are longer than with the other protocols due

to the additional message exchange and longer waiting times.

For read-only transactions (8.b) SER has worse response times than the other protocols. SER is the only protocol that aborts readers (even 2PL has very low abort rates for read-only transactions). If many update transactions are in the system, SER has very high abort rates and hence high response times. However, the abort rate, and with it the response times, decrease very fast with an increasing number of read-only transactions. Here, response times start to be comparable with the results of the other protocols.

Analysis. This experiment clearly shows that read-only transactions need a special treatment to avoid unnecessary aborts. The rather simple SER approach, where potential deadlocks are resolved by aborting transactions, results in an unacceptable high abort rate for read-only transactions (40% in the case of 40% read-only transactions). Therefore, the hybrid protocol seems a good alternative. Since readers are never aborted, read-only transactions yield good performance and do not require excessive resources. For updating transactions, the hybrid protocol provides serializability, unlike SI or CS. However, transactions must be declared read-only in advance to allow this special treatment. Although CS has the best performance results, it does not provide repeatable reads; this may be problematic in certain applications.

This experiment again shows that the applicability of 2PL is restricted to low conflict, low workload configurations and that it degrades very fast if the conditions are not optimal.

8.3 Scalability

The ability to scale up the system depends on the number of update operations in the system since they are executed on all sites. To analyze this factor we run an experiment with a workload of 20% short update transactions and 80% read-only transactions with an inter-arrival time of 40 ms per node. The scalability is limited with such a workload. Since all update operations are executed on all sites, increasing the number of nodes in the system results in an increasing number of write operations. This leads in the end to resource (CPU, disk) saturation. In our configuration using the proposed algorithms, this resource saturation starts at 60 nodes. Further increase in the number of nodes leads to performance degradation. This degradation is only due to the enormous amount of transaction processing power needed to perform the write operations at all nodes. 2PL, on the other hand, scales up only to 20 due to data contention.

In a second experiment we chose a decreasing update rate in order to analyze other factors affecting scalability. For example, the number of sites plays an important role since it influences the number of messages involved and, above all, the calculations involved in determining the total order. Again, the workload is a combination of short update and long read-only transactions. The inter-arrival time of update transactions is kept constant (4 ms) for the entire system, i.e., at a 10-node-system the inter-arrival time is 40 ms per node, whereas in a 100-node-system the rate is 400 ms per node (representing 250 transactions per second in the system). The inter-arrival time of read-only transactions is always 100 ms per node (i.e. 10 read-only transactions per second per node). The workload represents an applica-

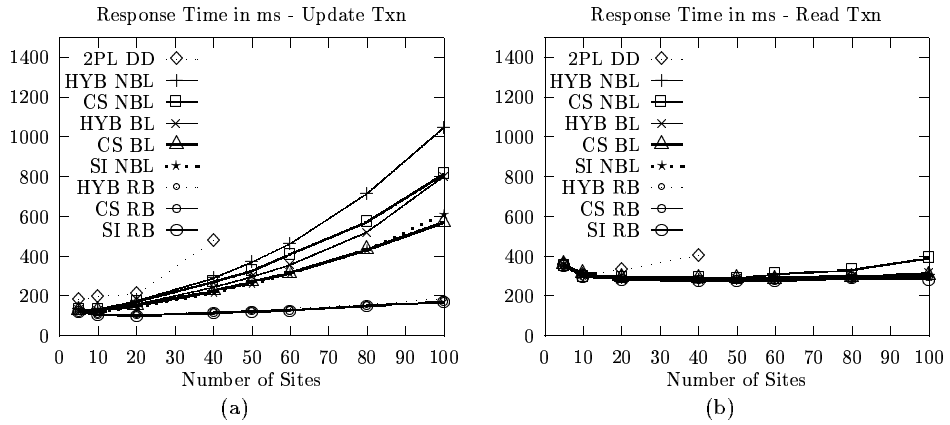

Fig. 9. Experiment 3: Response Time of (a) *update* and (b) *read-only* transactions for different number of nodes

tion where a formerly centralized OLTP database is used in a distributed manner (that means the same amount of write accesses is distributed over more nodes) and the analytical access (read operations) increases with the number of nodes. The communication parameters were set to the values of test run IV in Table 4. We skipped the SER protocols since we saw already in the previous experiment that SER aborts read-only transactions too often.

Figure 9 shows the response time for update transactions (9.a) and read-only transactions (9.b) as the number of sites in the system increases from 5 to 100.

For update transactions (9.a) the 5-node system behaves slightly worse than the 10-node system due to the higher load (each node executes the read operations of an update transaction each 20 ms while in the 10-node system each node executes the reads of an update transaction only each 40 ms). For ten nodes and higher we can observe a very similar behavior as observed when increasing communication overhead (see Figure 6). The response time for all protocols increases with the number of nodes due to the increased message delay (determining the total order and uniform reliable message delay take significantly more time when the number of nodes increase). The RB protocols behave better than their fault-tolerant counterparts and if complex communication protocols are used (RB and NBL), concurrency control methods with less conflict rate (CS) or less messages (SI) show better results. Compared to the results in Figure 6, only SI behaves a little bit worse, i.e., sending only one message does not have an impact since the communication processor is not overloaded. 2PL has worse behavior than the other protocols already for a small number of nodes. The problem is that each write operation has to wait for all the nodes to respond and with each additional node the probability increases that a write operation has to wait on one of the sites for a read-only transaction to release its locks.

The response time for read-only transactions (9.b) is similar for all evaluated protocols. Since none of the protocols considered in this experiment aborts read-only transactions, they are barely influenced by the behavior of update transactions. Only NBL-CS gets slightly worse when the number of sites increases since CS ac-

quires read locks for read-only transactions and has to wait for update transactions to release their write locks. When communication takes longer the NBL version keeps write locks longer than the BL and RB versions. Write locks are kept until all write operations have been performed and the commit message has been delivered. Since BL and RB send the commit message only with reliable delivery the message is delivered as soon at it is physically received. With NBL however, the commit message is only delivered when all acknowledgments have been received. Hence, when the number of sites in the system increases it takes longer to receive all acknowledgments and therefore, NBL blocks waiting read locks longer than BL or RB. Note that neither the HYB nor the SI protocol acquire read locks and hence have the same results for read-only transactions for both NBL and RB versions. For 2PL, even for 20 nodes the response time of read-only transactions is worse compared to the other protocols since read-only transactions have to wait long for update transactions to release their locks. From 40 nodes on, the response time increases due to the degrading response times of update transactions.

Analysis. The proposed algorithms scale up fairly well, providing good performance across a wide range of configurations. The main factor to consider is the communication delay (which, of course, increases with the number of sites in the system). Thus, our protocols scale well as long as the communication system scales well, i.e., it provides short message delays although the number of nodes increases.

9. CONCLUSION

In this paper, we have provided a comprehensive solution to eager, update everywhere database replication. By exploiting advanced communication semantics we optimize message exchange, support the ordering of transactions, and avoid the problem of distributed deadlocks, thereby addressing many of the concerns of database designers. A flexible choice of different levels of isolation and fault-tolerance allows the handling of data contention and to achieve high performance even in the case of a slow network. The architecture described above and the implementation details provided show the applicability and feasibility of the approach.

We have also presented a quantitative analysis of the compared performances 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. Summarized, our experiments demonstrate several points:

—Efficient communication plays a major role in replicated transaction processing. Severe performance problems can only be avoided by reducing the message overhead. Our technique minimizes the number of messages per transaction. For snapshot isolation, we were able to reduce the message overhead to one message per transaction. Another way to reduce the communication costs is to reduce the message latency. Our reconciliation based protocols use communication protocols with low message delay. They guarantee consistency, but 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.

—With our protocols, the message overhead is constant and does not increase with the transaction size.

—Long message delays increase the response times of the transactions. This leads to more concurrent transactions in the system, and hence, to higher data contention. 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.

—High updates rates can become a severe problem in a replicated system due to their extensive resource requirements. To alleviate the problem, conflict rates can be reduced by using cursor stability or snapshot isolation.

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

—Our protocols show good results even when the number of nodes in the system is high. The main factor is communication delay. As long as message delay is reasonable short our protocols show very good scalability.

—A comparison with a standard distributed 2PL protocol shows that the proposed protocols are much more stable and allow for a much broader range of workloads and configurations. They all avoid the fast and abrupt degradation experienced with 2PL/ROWA.

We believe eager update everywhere replication is feasible for a wide spectrum of applications and configurations. A fault-tolerant, fully serializable protocol can be used under certain conditions: fast communication, low system load, low conflict rates and the percentage of read-only transactions is reasonable high. If the system configuration is not ideal, as it will happen in most cases, the optimizations in terms of lower levels of isolation and fault-tolerance help to maintain reasonable performance while still guaranteeing replica consistency and replication transparency.

As part of future work, we are working on the implementation of these algorithms in a database management system to test their performance in a real environment. We have also explored a more sophisticated and tighter integration of the group communication primitives with transactional execution [Kemme et al. 1999]. Initial results indicate we can hide most of the communication overhead behind the transaction by using an optimistic approach that allows a transaction to execute before its total order is known. If the total order does not alter the existing serialization order, the transaction can commit. If the final total order violates the serialization order, the transaction is aborted and rescheduled in its proper position in the total order. This technique will allow to increase the feasibility range of the protocols in terms of acceptable communication delays.

Finally, we are also working on versions of the protocols that deal with partial replication. Using partial replication, objects need not be replicated on all sites and can even exist only locally on one site. As long as all transactions accessing replicated data (whether partially or fully replicated) are globally ordered by the total order, and concurrency control enforces some known criteria like *order preserving serializability* [Beeri et al. 1989], our protocols can be enhanced in a rather straightforward way to support partial replication.

ACKNOWLEDGMENTS

We would like to thank A. Schiper and F. Pedone for many helpful discussions on the topic. We are also grateful to Y. Breitbart and G. Weikum for their useful comments and suggestions on how to improve the paper.

APPENDIX

A. PROOFS OF THE PROTOCOLS

This appendix contains the proofs of the protocols presented in section 5. For each protocol, we prove that transaction atomicity is guaranteed in the failure free case. That is, a transaction is committed or aborted at all sites when no node failures or network partitions occur. Furthermore, we show that SER provides indeed serializability, while CS and SI guarantee the levels of isolation described in Section 4.2.

A.1 Serializability (SER)

Before we prove the atomicity and serializability characteristics of SER, we show that SER is deadlock free.

LEMMA 1. *The SER protocol is deadlock free.*

PROOF. We show that each lock of a transaction T_j will eventually be granted. To do so, we analyze the behavior of the protocol in all conflict situations. We have to distinguish the following situations:

—T_i has a read lock $r_i(X)$, WS_i has not yet been processed (i.e. T_i is still in its reading or send phase). Now T_j requests $w_j(X)$. Then T_i is aborted and T_j receives the lock. (Step 3.a.iii of the protocol in Figure 2).

—T_i has a read lock $r_i(X)$ or a write lock $w_i(X)$, WS_i has already been processed and T_j requests $w_j(X)$. T_j waits for T_i to finish (step 3.a.ii of the protocol). We denote this as $T_i \rightarrow T_j$. Now assume there is a deadlock, i.e., there is a sequence of transactions so that $T_j \rightarrow T_k \rightarrow T_1 \rightarrow \ldots \rightarrow T_i \rightarrow T_j$. T_k cannot wait for a write lock $w_j(Y)$ of T_j to be released since all write locks of T_j are requested in one atomic step together with $w_j(X)$. But if T_k waits for a read lock $r_j(Y)$ to be released then the conflicting operation of T_k must be a write operation. This, however, means, that WS_k was processed before WS_j and therefore at the time of processing WS_k, T_j would have been aborted according to step 3.a.iii of the protocol. Therefore, such a cycle cannot exist, and T_j will finally receive the lock.

—T_i has a write lock $w_i(X)$ and T_j requests a read lock $r_j(X)$. The two previous points show that once the write set WS_i has been processed, T_i will eventually receive all needed locks and therefore be able to terminate and release the locks. Then $r_j(X)$ can be granted.

□

In proving that SER is deadlock free, we have not used the total order property but several specific facts of the algorithm:

—Read/write conflicts cannot cause deadlocks since the reading transaction gets aborted or will not acquire any more locks until it commits.

—Write/write conflicts cannot cause deadlocks since all write locks of a transaction are acquired in one step.

—A write/read conflict alone cannot cause a deadlock.

THEOREM 3. *In a failure free environment, the SER protocol guarantees the atomicity of a transaction.*

PROOF. We have to show that if a node N commits/aborts a local update transaction T then all nodes commit/abort T. In the protocol, the owner N of T always decides on commit/abort. We have to show that the remote nodes are able to obey this decision. An abort decision can easily be obeyed, since remote nodes do not terminate a transaction until they receive the decision message from N. The decision to commit can be obeyed since it follows from the lack of deadlocks that all nodes will eventually be able to grant all write locks for T and execute the operations. □

THEOREM 4. *SER is 1-copy-serializable.*

PROOF. Based on Bernstein et al. [1987], we show that each replicated data history H has an acyclic replicated data serialization graph $RDSG(H)$. A serialization graph $SG(H)$ has a node for each committed transaction and an edge $T_i \rightarrow T_j$ if H orders operation o_i before conflicting operation o_j. $RDSG(H)$ extends $SG(H)$ by adding enough edges such that the graph orders transactions with conflicting operations on the same logical object. Let $SG(H)$ be the serialization graph of a history H produced by the SER protocol. It is easy to see that $SG(H)$ itself is a $RDSG(H)$. SER always performs write operations on all copies (ROWA). Hence, whenever two operations (read or write) conflict on the same logical object they conflict on at least one physical object, and with this, create an edge in $SG(H)$.

To proof that $SG(H)$ is acyclic, we use the fact that all write sets are totally ordered, i.e., if $WS_i < WS_j$ at one site, then $WS_i < WS_j$ at all sites. We show that if $T_i \rightarrow T_j$ in $SG(H)$, then $WS_i < WS_j$. Therefore, $SG(H)$ cannot have any cycles but has dependencies only in accordance to the total order provided by the communication system. An edge $T_i \rightarrow T_j$ in $SG(H)$ exists due to three different kind of conflicting operations (on the same copy): (r_i, w_j), (w_i, w_j) or (w_i, r_j).

—(r_i, w_j): WS_i must have been delivered before WS_j, otherwise T_i would have been aborted according to step 3.a.iii of the protocol in Figure 2 and would not be in $SG(H)$.

—(w_i, w_j):
This is only possible when $WS_i < WS_j$, since write locks are acquired according to the total order write sets are delivered (step 3, especially 3.a.ii of the protocol).

—(w_i, r_j):
T_j can only read from T_i when T_i has committed, hence after WS_i has been delivered. Since all read operations of T_j must be performed before WS_j is sent according to step 2 of the protocol, WS_j was sent and delivered after WS_i.

To extend this proof to read-only transactions, we use dummy write sets that we assume to be delivered at the same logical time the last read lock is granted. With this we can use the same argument as for update transactions. □

A.2 Cursor Stability (CS)

THEOREM 5. *The CS protocol is deadlock free and, in case of a failure free environment, it guarantees the atomicity of a transaction.*

PROOF. The proofs are identical to the proofs of the SER protocol. □

THEOREM 6. *CS provides 1-copy-equivalence and regarding the level of isolation, it avoids the phenomena $P1 - P2$, but $P3 - P5$ may occur.*

PROOF. CS does not provide serializability, hence we cannot apply the combined 1-copy-serializability proof using a replicated data serialization graph. 1-copy-equivalence itself means that the multiple copies of an object appear as one logical object. This is true for several reasons. First, we use a read-only/write-all approach, i.e. transactions perform their updates on all copies. Second, transaction atomicity guarantees that all sites commit the updates of exactly the same transactions. Third, using the total order guarantees that all these updates are executed in the same order at all sites. Hence, all copies of an object have the same value if we look at them at the same logical time (for example, let T_i and T_j be two transactions updating object X, WS_i is delivered directly before WS_j and we look at each node at the time the lock for the copy of X is granted to T_j. At that time each copy of X has the value that was written by T_i).

The phenomena P1 and P2 are avoided since:

- P1 Dirty read: $w_1(X_i) \ldots r_2(X_i) \ldots (c_i \text{ or } a_i)$ is not possible since updates become visible only after commit of a transaction. Read operations, whether obtaining short or long locks, must wait for write locks to be released at EOT to receive a committed version of the data.
- P2 Lost update: $r_1(X_i) \ldots w_2(X_i) \ldots w_1(X_i) \ldots c_1$ is not possible since T_1 should obtain a long read lock if it wants to write X (according to step 1 in Figure 4). Therefore, it will be aborted when WS_2 is processed.

All the other phenomena are possible due to the short read locks. □

A.3 Snapshot Isolation (SI)

THEOREM 7. *In a failure free environment, SI guarantees the atomicity of a transaction.*

PROOF. With SI the local node N only sends the write set WS_i and all nodes decide on their own on the outcome of the transaction. To show the atomicity of a transaction we have to show that all nodes make the same decision. This is done by induction on the sequence of write sets that arrive (this sequence is the same at all nodes). Hence, we use the total order to prove the atomicity of transactions.

Assume an initial transaction T_0 with $TS_0(EOT) = 0$. All objects are labeled with T_0. Now assume WS_i is the first write-set to be delivered. The $BOT(T_i)$ must be 0. Therefore all nodes will decide on commit and perform all operations. Now assume, already $n-1$ write-sets have been delivered and all nodes have always

behaved the same, i.e., have committed and aborted exactly the same transactions. Therefore, on all nodes there was the same series of committed transactions that updated exactly the same objects in the same order. Since these transactions have the same EOT timestamps at all nodes, the versions of the objects are exactly the same at all nodes when all these transactions have terminated. Hence, when WS_i is the n'th write-set to be delivered, $\forall nodes, \forall w_i(X) \in WS_i$ the version check in step 3 of Figure 5 will have the same outcome. Although at the time WS_i is processed not all of the preceding transactions might have committed, the check is always against the last version. This version might already exist if all preceding transactions that updated this object have already committed and there is no granted lock on the object (step 3.a of the protocol), or the version will be created by the transaction that is the last still active preceding one to update the object (3.b of the protocol). □

THEOREM 8. *SI provides 1-copy-equivalence and avoids phenomena $P1 - P4$, but $P5$ may occur.*

PROOF. 1-copy-equivalence is guaranteed for the same reasons as it is guaranteed for the CS protocol. Phenomena $P1 - P4$ are avoided due to the following reasons:

- P1 Dirty read: $w_1(X_i) \ldots r_2(X_i) \ldots (c_1 \text{ or } a_1)$ is not possible since updates become visible only after commit of a transaction. In our algorithm, $TS_2(BOT)$ is lower than $TS_1(EOT)$ and therefore T_2 will not read from T_1 (step 1 of the protocol).
- P2 Lost update: $r_1(X_i) \ldots w_2(X_i) \ldots w_1(X_i) \ldots c_1$ is not possible. $TS_2(EOT)$ is bigger than $TS_1(BOT)$ but smaller than $TS_1(EOT)$. Therefore, in the moment in which $w_1(X_i)$ is requested T_1 will be aborted (step 3 of the protocol).
- P3 Non-repeatable read: $r_{11}(X_i) \ldots w_2(X_i) \ldots c_2 \ldots r_{12}(X_i)$ is not possible. $TS_1(BOT)$ is lower than $TS_2(EOT)$ and therefore T_1 will not read from T_2 at its second read but reconstruct the older version (step 1 of the protocol).
- P4 Read skew: $r_1(X_i) \ldots w_2(X_i) \ldots w_2(Y_j) \ldots c_2 \ldots r_1(Y_j)$ is not possible since a transaction only reads data-versions created before the transaction started (step 1 of the protocol).
- P5 Write skew: $r_1(X_i) \ldots r_2(Y_j) \ldots w_1(Y_j) \ldots w_2(X_i)$ is possible since the two write operations do not conflict and read/write conflicts are not considered.

□

B. PROOFS OF FAULT-TOLERANCE BEHAVIOR

In Section 6 we showed that transaction atomicity and with it data consistency is guaranteed on all available sites. Furthermore, we informally discussed that using uniform reliable delivery atomicity is guaranteed in the entire system, i.e., both on faulty and non-faulty nodes. This means that the set of transactions committed resp. aborted at a failed node is a subset of transactions committed resp. aborted at available nodes. In this appendix, we provide more exhaustive proofs. To do so, we look at a node N which fails during view V_i and at the different states a transaction T_i on node N might be in when N fails. For each case we show that N does not contradict the decision that is made by the V_i-available (or short:

available) nodes. This means that when N has committed/aborted T_i the available nodes do the same. When T_i was still active on N then the other sites only commit the transaction when this decision is correct (the failed node would have been able to commit it too, if it had not failed). Otherwise they abort it.

THEOREM 9. *The SI protocol using uniform reliable message delivery guarantees the atomicity of all transactions on all sites.*

PROOF. Consider a transaction T_i at a failed node N:

(1) T_i is in its reading phase (T_i is local): In this case none of the other nodes knows of T_i. The transaction has no effect neither on N nor on any other node. Hence, T_i can be considered aborted on all sites.

(2) T_i is in its send phase (T_i is local and WS_i has been sent in view V_i but not yet received): Reliable multicast (and hence uniform reliable multicast) guarantees, that all available nodes or none of them will receive T_i's write set. In the latter case, uniform reliable multicast guarantees, that no other node N' failing during V_i has received WS_i. Hence, if it is not delivered at any site the transaction is considered aborted. Otherwise, all available sites have received the same sequence of messages up to WS_i and hence decide on the same outcome of T_i (according to theorem 7 in Appendix A).

(3) T_i, remote or local, is in its lock or write phase: This means that WS_i has already been delivered at N before or during view V_i. Hence, according to the uniform reliable delivery, WS_i will be delivered at all available sites (note that reliable delivery does not provide this guarantee). Again, on all these sites (including N) the same sequence of messages is received before WS_i is received and the version check will have the same results (see theorem 7). Hence, all can decide on the same outcome (although N will not complete T_i before it fails).

(4) Finally, a transaction T_i, local or remote, was committed/aborted on N when N crashed: This means WS_i is delivered on N and due to the uniform reliable message delivery, all available sites will receive WS_i. Hence, the same holds as in the previous case.

This together with theorem 1 of Section 6 provides the atomicity of all transactions on all sites. □

THEOREM 10. *The SER, CS and HYBRID protocols using uniform reliable message delivery for both the WS and commit messages, and aborting all in-doubt messages after a view change guarantee the atomicity of all transactions on all sites.*

PROOF. Consider a transaction T_i at a failed node N:

(1) T_i is in its reading phase: As with SI, this transaction is considered aborted.

(2) T_i is in its send phase: T_i is still active at N and N has not yet decided whether to commit or abort T_i. Since N is the only one who can decide to commit since only N knows about the read locks (this is different to the SI protocol) the others are only allowed to decide on abort. According to the semantics of the reliable/uniform reliable multicast, all or none of the available nodes will deliver T_i's write set. If it is not delivered at any site the transaction is considered aborted. If it is delivered, it is an in-doubt transaction at all sites

(N has not yet sent a decision message) and all sites will abort it. Hence, all make the correct decision.

(3) T_i is a local transaction of N, the write set of T_i was delivered at N but N has not yet sent the decision message. Again, this requires that the transaction is aborted at the other sites. Since WS_i was sent with uniform reliable message delivery, it will be delivered at all sites. However, after the crash of N it is an in-doubt transaction and all sites will abort it.

(4) T_i has processed its lock phase and submitted the commit message to the communication module. However, N fails before any other node physically receives the commit message. In this situation, all available sites have delivered WS_i but none of them delivers c_i. T_i is an in-doubt transaction and will be aborted. We have to show that T_i was still active when N failed. This is the case since the database module of N waits to commit until the communication modules delivers the commit message (step 5 of the SER/CS/Hybrid protocols). However, due to the uniform reliable delivery of the commit message, the communication module only delivers it when it is guaranteed that all sites will deliver the message. Since this is not the case the transaction is still pending at N when N fails.

(5) T_i has processed its lock phase and submitted the abort message to the communication module. Since the abort message is only sent with reliable delivery, the communication module delivers the message back to the database module immediately and does not wait until it is guaranteed that the other nodes will deliver the message. Hence, N has either aborted T_i or T_i was still active at the time of the failure. However, the other nodes will abort N in any case. Either all deliver the abort message and then abort or T_i is an in-doubt transaction and is aborted.

(6) T_i, being local or remote, was committed at N before N failed. This can only happen when both write set and commit message were delivered. However, the messages are only delivered when it is guaranteed that they will be delivered at all available sites. Hence, T_i is no in-doubt transaction at the available sites and will commit. Furthermore, the uniform reliable delivery of the write set excludes the scenario where all nodes of the remaining group deliver the commit message but not the write set.

We have shown that a failed node aborts/commits exactly the same transactions as the available nodes until it fails. After the node failure all available nodes have the same in-doubt transactions which they can safely abort. This together with theorem 2 of Section 6 guarantees the atomicity of all transaction on all sites. □

THEOREM 11. *The SER, CS and HYBRID protocols using uniform reliable message delivery for the WS messages, and blocking all in-doubt messages after a view change have the same properties as the non-blocking versions.*

PROOF. The proof is the same as for the previous theorem except the case which forces to block in-doubt transactions. This is case 4 when T_i has processed its lock phase and submitted the decision message to the communication module but N fails before any other node physically receives the decision message. Since both commit and abort are only sent with reliable delivery, the database does not wait

until it is guaranteed that the other nodes will receive the message. Hence, T_i can be still active, committed or aborted at the time of N's failure. To guarantee T_i's atomicity all available sites must block it. □

REFERENCES

Adya, A. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations of Distributed Transactions.* Ph. D. thesis, Massachusetts Institute of Technology.

Adya, A., Liskov, B., and O'Neil, P. 2000. Generalized isolation level definitions. In *Proc. of the Int. Conf. on Data Engineering (ICDE)* (San Diego, California, March 2000), pp. 67–78.

Agrawal, D., Alonso, G., El Abbadi, A., and Stanoi, I. 1997. Exploiting atomic broadcast in replicated databases. In *Proc. of Europ. Conf. on Parallel Processing (Euro-Par)* (Passau, Germany, August 1997), pp. 496–503.

Agrawal, D. and El Abbadi, A. 1990. The tree quorum protocol: an efficient approach for managing replicated data. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)* (Brisbane, Australia, August 1990), pp. 243–254.

Agrawal, D., El Abbadi, A., and Steinke, R. C. 1997. Epidemic algorithms in replicated databases. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems (PODS)* (Tucson, Arizona, May 1997), pp. 161–172.

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

Alonso, G. 1997. Partial database replication and group communication primitives. In *Proc. of European Research Seminar on Advances in Distributed Systems* (Zinal, Switzerland, March 1997).

Alonso, G., Kamath, M., Agrawal, D., El Abbadi, A., Günthör, R., and Mohan, C. 1996. Advanced transaction models in the workflow contexts. In *Proc. of the Int. Conf. on Data Engineering (ICDE)* (New Orleans, Louisiana, February 1996), pp. 574–581.

Alonso, G., Reinwald, B., and Mohan, C. 1997. Distributed data management in workflow environment. In *Proc. of the Int. Workshop on Research Issues in Data Engineering (RIDE)* (Birmingham, United Kingdom, April 1997).

Alsberg, P. and Day, J. 1976. A principle for resilient sharing of distributed resources. In *Proc. of the Int. Conf. on Software Engineering* (San Francisco, California, October 1976), pp. 562–570.

Anderson, T. A., Breitbart, Y., Korth, H. F., and Wool, A. 1998. Replication, consistency, and practicality: Are these mutually exclusive? In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data* (Seattle, Washington, June 1998), pp. 484–495.

ANSI X3.135-1992. 1992. *American National Standard for Information Systems – Database Languages – SQL.*

Beeri, C., Bernstein, P., and Goodman, N. 1989. A model for concurrency in nested transaction systems. *Journal of the ACM 36*, 2, 230–269.

Berenson, H., Bernstein, P. A., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. 1995. A critique of ANSI SQL isolation levels. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data* (San Jose, California, June 1995), pp. 1–10.

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

Birman, K., Schiper, A., and Stephenson, P. 1991. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems 9*, 3, 272–314.

Breitbart, Y., Komondoor, R., Rastogi, R., Seshadri, S., and Silberschatz, A. 1999. Update propagation protocols for replicated databases. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data* (Philadephia, Pennsylvania, June 1999), pp. 97–108.

Breitbart, Y. and Korth, H. F. 1997. Replication and consistency: Being lazy helps sometimes. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS)* (Tucson, Arizona, May 1997), pp. 173–184.

Bridge, W., Joshi, A., Keihl, M., Lahiri, T., Loaiza, J., and Macnaughton, N. 1997. The Oracle universal server buffer manager. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)* (Athens, Greece, August 1997), pp. 590–594.

Carey, M. and Livny, M. 1989. Parallelism and concurrency control performance in distributed database machines. In *Proc. of the ACM SIGMOD Management on Data* (Portland, Oregon, June 1989), pp. 122–133.

Ceri, S., Houtsma, M. A., Keller, A., and Samarati, P. 1991. A classification of update methods for replicated databases. Technical report, Computer Science Departement, Stanford Universtity, CS-TR-91-1392.

Chandra, T. D. and Toueg, S. 1991. Unreliable failure detectors for asynchronous systems. In *Proc. of the ACM Symp. on Principles of Distributed Computing (PODC)* (Montreal, Canada, August 1991), pp. 325–340.

Chen, S.-W. and Pu, C. 1992. A structural classification of integrated replica control mechanisms. Technical report, Departement of Computer Science, Columbia University, New York, CUCS-006-92.

Cheung, S. Y., Ahamad, M., and Ammar, M. H. 1990. The grid protocol: A high performance scheme for maintaining replicated data. In *Proc. of the Int. Conf. on Data Engineering (ICDE)* (Los Angeles, California, February 1990), pp. 438–445.

Chundi, P., Rosenkrantz, D. J., and Ravi, S. S. 1996. Deferred updates and data placement in distributed databases. In *Proc. of the Int. Conf. on Data Engineering (ICDE)* (New Orleans, Louisiana, February 1996), pp. 469–476.

Dolev, D. and Malki, D. 1996. The Transis approach to high availability cluster communication. *Communications of the ACM 39*, 4, 63–70.

El Abbadi, A. and Toueg, S. 1989. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems 14*, 2, 264–290.

Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. 1976. The notions of consistency and predicate locks in a database system. *Communications of the ACM 19*, 11, 624–633.

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

Friedman, R. and van Renesse, R. 1995b. Strong and weak virtual synchrony in Horus. Technical report, Departement of Computer Science, Cornell University, TR-95-1537.

Gifford, D. 1979. Weighted voting for replicated data. In *Proc. of the ACM SIGOPS Symp. on Operating Systems Principles* (Pacific Grove, California, December 1979), pp. 150–159.

Goldring, R. 1994. A discussion of relational database replication technology. *InfoDB 8*, 1.

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

Gray, J., Lorie, R., Putzolu, G., and Traiger, I. 1976. Granularity of locks and degrees of consistency in a shared database. In *Modeling in Data Base Mangement Systems* (1976). Elseiver North-Holland, Amsterdam.

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

Guerraoui, R. and Schiper, A. 1995. Transaction model vs virtual synchrony model: bridging the gap. In *Theory and Practice in Distributed Systems*, LNCS 938 (September 1995), pp. 121–132. Springer-Verlag.

Gupta, R., Haritsa, J., and Ramamritham, K. 1997. Revisiting commit processing in distributed database systems. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data* (Tucson, Arizona, June 1997), pp. 486–497.

Hadzilacos, V. and Toueg, S. 1993. Fault-tolerant broadcasts and related problems. In S. Mullender Ed., *Distributed Systems*, pp. 97–145. Addison-Wesley.

Holliday, J., Agrawal, D., and Abbadi, A. E. 1999. The performance of database replication with group multicast. In *Proc. of the Int. Symp. on Fault-Tolerant Computing (FTCS)* (Madison, Wisconsin, June 1999), pp. 158–165.

Kemme, B. and Alonso, G. 1998. A suite of database replication protocols based on group communication primitives. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)* (Amsterdam, The Netherlands, May 1998).

Kemme, B., Pedone, F., Alonso, G., and Schiper, A. 1999. Processing transactions over optimistic atomic broadcast protocols. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)* (Austin, Texas, June 1999).

Krishnakumar, N. and Bernstein, A. 1991. Bounded ignorance in replicated systems. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS)* (Denver, Colorado, June 1991), pp. 63–74.

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7, 558–565.

Maekawa, M. 1985. A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems 3*, 2, 145–159.

Moser, L. E., Amir, Y., Melliar-Smith, P. M., and Agarwal, D. A. 1994. Extended virtual synchrony. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)* (Poznan, Poland, June 1994), pp. 56–65.

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

Neiger, G. and Toueg, S. 1988. Automatically increasing the fault-tolerance of distributed systems. In *Proc. of the ACM Symp. on Principles of Distributed Computing (PODC)* (Toronto, Canada, August 1988), pp. 248–262.

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

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

Pacitti, E., Minet, P., and Simon, E. 1999. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)* (Edinburgh, Scotland, September 1999), pp. 126–137.

Pedone, F., Guerraoui, R., and Schiper, A. 1997. Transaction reordering in replicated databases. In *Proc. of the Symp. on Reliable Distributed Systems (SRDS)* (Durham, North Carolina, October 1997).

Pu, C. and Leff, A. 1991. Replica control in distributed systems: An asynchronous approach. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data* (Denver, Colorado, May 1991), pp. 377–386.

Rabinovich, M., Gehani, N. H., and Kononov, A. 1996. Scalable update propagation in epidemic replicated databases. In *Proc. of the Int. Conf. on Extending Data Base Technology (EDBT)* (Avignon, France, 1996), pp. 207–222.

Schiper, A. and Sandoz, A. 1993. Uniform reliable multicast in a virtually synchronous environment. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)* (Pittsburgh, Pennsylvania, 1993), pp. 561–568.

Shasha, D. 1997. Lessons from wall street. In *Tutorial in the Proc. of the ACM SIGMOD Int. Conf. on Management of Data* (Tucson, Arizona, June 1997), pp. 498–501.

Sidell, J., Aoki, P., Sah, A., Staelin, C., Stonebraker, M., and Yu, A. 1996. Data replication in Mariposa. In *Proc. of the Int. Conf. on Data Engineering (ICDE)* (New Orleans, Louisiana, February 1996), pp. 485–494.

Stacey, D. 1994. Replication: DB2, Oracle, or Sybase. *Database Programming & Design 7*, 12.

Stanoi, I., Agrawal, D., and El Abbadi, A. 1998. Using broadcast primitives in replicated databases. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)* (Amsterdam, The Netherlands, May 1998).

STONEBRAKER, M. 1979. Concurrency control and consistency of multiple copies of data in distributed Ingres. *IEEE Transactions on Software Engineering 3*, 3, 188–194.

VAN RENESSE, R., BIRMAN, K. P., AND MAFFEIS, S. 1996. Horus: A flexible group communication system. *Communications of the ACM 39*, 4, 76–83.

WHITNEY, A., SHASHA, D., AND APTER, S. 1997. High volume transaction processing without concurrency control, two phase commit, SQL or C++. In *Int. Workshop on High Performance Transaction Systems* (Asilomar, California, September 1997).