Appears in the *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC '97), Santa Barbara, CA, August 1997*

Lazy Consistency Using Loosely Synchronized Clocks

Atul Adya **Barbara Liskov**

Laboratory for Computer Science,
Massachusetts Institute of Technology,
545 Technology Square, Cambridge, MA 02139
{adya,liskov}@lcs.mit.edu

Abstract

This paper describes a new scheme for guaranteeing that transactions in a client/server system observe consistent state while they are running. The scheme is presented in conjunction with an optimistic concurrency control algorithm, but could also be used to prevent read-only transactions from conflicting with read/write transactions in a multi-version system. The scheme is lazy about the consistency it provides for running transactions and also in the way it generates the consistency information. The paper presents results of simulation experiments showing that the cost of the scheme is negligible.

The scheme uses multipart timestamps to inform nodes about information they need to know. Today the utility of such schemes is limited because timestamp size is proportional to system size and therefore the schemes don't scale to very large systems. We show how to solve this problem. Our multipart timestamps are based on real rather than logical clocks; we assume clocks in the system are loosely synchronized. Clocks allow us to keep multipart timestamps small with minimal impact on performance: we remove old information that is likely to be known while retaining recent information. Only performance and not correctness is affected if clocks get out of synch.

1 Introduction

This paper describes a new algorithm that insures transactions running at clients in a client/server system always view a consistent state as they run. The algorithm is useful in conjunction with optimistic concurrency control mechanisms (e.g., [1]), and also for ensuring that read-only transactions can commit without interfering with read/write transactions in a multi-version system (e.g., [2, 7, 30]).

The algorithm is intended for use in a distributed environment in which servers provide reliable persistent storage for online information (e.g., a database or objects). Applications run at client machines that are distinct from the servers. Clients maintain caches that store copies of persistent objects; clients run application transactions locally on cached copies of persistent objects and send modifications to the servers when transactions commit. The scheme is designed to work in a very large system, e.g., tens of thousands of servers and hundreds of thousands of clients.

The new algorithm propagates information about consistency using multipart timestamps or *multistamps*. Today the utility of multistamps is limited because their size is proportional to system size and therefore they don't scale to very large systems. We show how to reduce the size of multistamps. Unlike other multistamp (or vector clock) schemes, e.g., [4, 17, 25, 8, 15], our scheme is based on time rather than on logical clocks: each entry in a multistamp contains a timestamp representing the clock time at some server in the system. Using time rather than logical clocks allows us to keep multistamps small by removing old information. As a result, multistamps need little storage at nodes and little space in messages. Furthermore, because we prune based on time, the discarded information is likely to already be known to interested parties; therefore discarding it has little impact on system performance. We assume that clocks are loosely synchronized; such an assumption is realistic in today's environment [23]. The correctness of the scheme is not affected if information of interest is pruned too early or clocks get out of synch (although performance may be).

This paper describes the new scheme in conjunction with an optimistic concurrency control algorithm, AOCC, which was developed for use in a distributed client/server environment. AOCC has been shown to outperform other concurrency control mechanisms for all common workloads and realistic system assumptions [1, 13]. In particular, AOCC outperforms the best locking approach, adaptive callback locking [6]. One reason for AOCC's good performance is that it allows transactions to use cached information without communication with servers; locking mechanisms require such communication at least when objects are modified. However, the reduced communication comes at a cost: unlike with locking, with AOCC transactions can view an inconsistent state while they run. Such transactions cannot harm the persistent state since they will abort, but

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136.

it is nevertheless desirable to provide transactions with a consistent view, which allows application programmers to depend on invariants. This means code need not check whether invariants hold (as it would need to do if it could not depend on invariants) and it can display consistent information to users.

The paper describes how to augment AOCC to provide running transactions with consistent views. The consistency provided by our scheme is weaker than serializability; we discuss this point further in Section 3. We call it "lazy" consistency because it is what we can provide with very little work. We believe lazy consistency is appropriate for our system since we commit in the standard way (by communicating with the server); a transaction may still abort even though it viewed a consistent state.

The consistency mechanism uses multistamps to warn clients of potential violations of consistency. Multistamps are sent to clients on messages that are already flowing in the system. We guarantee they arrive at clients before a transaction might observe an inconsistency. Clients are also lazy; they act on the multistamp information only if it might affect the current transaction. Being lazy buys time so that the needed consistency information is highly likely to be present by the time it is needed.

The paper presents results of simulation experiments to evaluate the cost of the lazy scheme. Our results show that this cost is very small. The cost is manifested by "fetch stalls" in running transactions; these are events where a fetch done by a client because of a cache miss causes it to communicate with another server before continuing to run the transaction, where the communication would not have been required in the basic AOCC scheme. Our results show that when contention is low, fewer than one in a thousand fetches lead to stalls; even in stressful workloads, less than one in one hundred fetches lead to stalls. Therefore we believe the cost of the scheme is negligible. The studies also evaluate the effectiveness of pruning multistamps and show that small multistamps are as effective as larger ones in preventing stalls.

Thus the paper makes three contributions:

1. It presents a new efficient scheme that provides consistency for running transactions.

2. It presents a new way of implementing multistamps that allows them to be pruned safely and effectively. This keeps multistamps small so that they can be used even in very large systems where otherwise they would not be practical.

3. It presents the results of simulation studies that show the scheme provides consistency with almost no impact on system performance.

The remainder of the paper is organized as follows. Section 2 describes related work. Section 3 defines lazy consistency and discusses how it relates to serializability and causality. Section 4 describes our system and how AOCC works; Section 5 describes our implementation of lazy consistency; and Section 6 presents our performance results. We conclude with a summary of our results.

2 Related Work

Lazy consistency is similar to "degree-2 isolation"; using the terminology of Gray and Reuter [12], our system provides degree-3 isolation for all committed transactions and degree-2 isolation for running transactions.

Earlier research has investigated ways of providing consistency for read-only transactions [3, 30] using serializability or weaker notions of consistency that are different from lazy consistency. For example, some schemes ensure that a read-only transaction is serializable with all read-write transactions but the whole system may not be serializable when all read-only transactions are taken into account.

Chan and Gray [7] propose lazy consistency as a correctness criterion but their work is concerned with committing read-only transactions more cheaply as opposed to providing consistency for transactions that abort. They also propose an implementation technique for a distributed client/server system using a kind of multistamp scheme but ignore all details that would make it a practical scheme.

Our work is related to orphan detection schemes [20]. Such schemes abort transactions before they can observe an inconsistency; they guarantee a stronger property than lazy consistency since they detect anti-dependencies also (discussed in Section 3).

3 Lazy Consistency

This section defines the kind of consistency our new mechanism provides.

We refer to the i^{th} version of object x as x_i. If a transaction creates a version x_i, we assume that it read x_{i-1}. Transaction T is said to *directly depend* on transaction U if it read x_i and U created version x_i; we also say that T has read from U. We say that T *depends* on U if there exist transactions V_1, V_2, ... V_n, where U is V_1 and T is V_n and V_2 directly depends on V_1, V_3 directly depends on V_2, and so on.

A running transaction Q will view a consistent state if the following condition is satisfied:

> If Q depends on T, it must observe the complete effects of T.

For example, suppose that T creates x_i and y_j. If Q reads x_i and also reads y, it must read y_j or a version of y that is later than y_j.

The same condition is used for providing consistent views for read-only transactions by Chan and Gray [7]; that paper proves that if the condition is satisfied, Q will observe a consistent snapshot of the database. Here we present a brief synopsis of the proof. Assume T_1, T_2, ... T_n is a prefix of the serialization history of the system, where T_n is the latest transaction whose updates have been observed by Q.

Suppose Q only depends on transactions T_{i_1}, T_{i_2}, ... T_{i_m} and has missed the updates of transaction T_j. Since we are ensuring that the above condition is satisfied for Q, T_j's updates have no impact on T_{i_1}, T_{i_2}, ... T_{i_m} (they do not depend on T_j). If we consider a history in which we remove (all such) T_j from the set of committed transactions, there will be no change in the database state observed by Q. Since a transaction transforms the database from one consistent state to another and the output of each transaction in T_{i_1}, T_{i_2}, ... T_{i_m} is unaffected by the presence or absence of T_j, their combination must yield a consistent database state.

A transaction Q that views a consistent state is not necessarily serializable with all the committed transactions in the system. For example, suppose there are two objects x and y stored at servers X and Y respectively and suppose that transactions U and T run in parallel and commit and then Q reads x_0 and y_2:

U: Read(x_0) Read(y_1) Write(x_1)
T: Read(y_1) Write(y_2)
Q: Read(x_0) Read(y_2)

In this scenario, Q has observed the effects of T but missed the effects of U. To commit Q, we would need to serialize it both before U and after T, but since U is already serialized before T, this is impossible. The intuition why Q does not observe inconsistencies is that since T does not depend on U, its modifications must preserve system invariants regardless of what U does. A possible invariant that might be preserved is $x \leq y$, where U stores the value of y in x and T increases y.

This problem of transaction Q not being serializable can occur only when there is an anti-dependency: A transaction A *anti-depends* on B if A overwrites an object version that B has read. In this case, T anti-depends on U. Figure 1 shows a cycle that is formed in the dependency graph [3] for this schedule due to anti-dependencies.

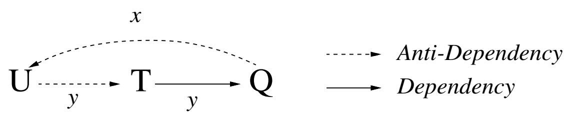

Figure 1: A non-serializable transaction that has observed a consistent database state.

Lazy consistency is independent of causality [18]: a transaction Q might observe the effects of T_2 while missing the effects of T_1, where both T_1 and T_2 ran at the same client in order T_1; T_2. Our implementation guarantees causality for individual clients, which ensures that a client always observes the effects of all its previous committed transactions and the transactions they depended on. We believe such *local causality* is good since it means code within a transaction can depend on invariants holding between objects observed directly and also results computed by its earlier transactions and stored in local variables. Causality is discussed further in Section 6.3, where we show the additional cost incurred over lazy consistency to support local causality. We also show what it would cost to support *global causality*, in which a client that observes the effects of transaction T of some other client also observes effects of all earlier transactions of that client (and the transactions they depended on). Note that locking ensures global causality for active transactions.

4 Base Algorithm

This section provides an overview of our system and AOCC. More information can be found in [1, 10, 13, 19].

Servers store the database objects in pages on disk; objects are typically smaller than pages. Clients maintain a page cache; they fetch missing pages from servers and use approximate LRU for cache management. Clients and servers rely on ordered communication, e.g., TCP.

Transactions run entirely at clients; clients communicate with servers only when there is a miss in the cache, and to commit transactions. The system serializes transactions using AOCC [1, 13] and two-phase commit [11]. (Two-phase commit is avoided for transactions that use objects at only one server.)

AOCC works as follows: While a transaction T is running, client C keeps track of the objects it used and objects it modified. When T is ready to commit, C sends this information together with the new states of modified objects to a server at which some objects used by this transaction reside. This server acts as the coordinator for the commit; other servers where objects used by the transaction reside act as participants.

The participants *validate* T to ensure that it used only up-to-date versions of all objects and that it has not modified any objects used by other prepared or committed transactions, i.e., we use backward validation [14]. Validation uses a *validation queue* or VQ to track read/write sets of prepared and committed transactions; details can be found in [1].

Each participant sends its validation results to the coordinator who commits the transaction iff all participants voted positively. The coordinator sends its decision to client C and the participants. The delay observed by the client is due to phase 1 only; phase 2 happens in the background. In phase two, a participant installs T's updates (makes them available to other transactions).

When updates of a transaction run by client C1 modify objects in the cache of client C2, this causes C2's cache to contain out-of-date information. Furthermore, if C2 then runs a transaction that observes this out-of-date information, that transaction will be forced to abort. To avoid such aborts and keep client caches relatively up to date, servers send *invalidation messages*. The server maintains a per-client *directory* that lists pages cached at the client (the directory may list pages no longer at the client since the server has not yet been informed that they were dropped). The server uses the directories to determine what invalidations to generate;

each invalidation identifies an object that may be out of date at a particular client.

An invalidation message to a client contains all invalidations for that client. When a client receives an invalidation message it discards invalid objects (but not the containing page) and aborts the current transaction if it used them. In addition to keeping caches almost up-to-date, which reduces the number of aborts, invalidation messages also cause transactions that must abort to abort sooner.

Invalidation messages are piggy-backed on other messages that the server sends to the client; the client acks these messages (acks are also piggybacked), which allows the server to discard pending invalidations for the client. However, invalidations are not allowed to remain pending indefinitely. Instead there is a *timeout period* (half a second in our current implementation). If a server has pending invalidations for the client, and if some of these invalidations are "old", i.e., they were generated nearly a timeout period ago, the server sends invalidations to the client on its own account (piggybacked on "I'm alive" messages). Thus invalidations are highly likely to arrive within half a second of their generation.

Although our scheme keeps caches almost up to date, it does not guarantee that they are up to date. When transactions run they can observe old information, and in fact if a transaction T modified objects at two different servers, a transaction running at some other client might observe the new state of one object and the old state of the other. Thus, it is possible for a transaction in our system to observe an inconsistent state. Our new algorithm prevents this situation from happening.

5 The Lazy Scheme

We now present our lazy scheme for providing consistent views for running transactions. Our scheme assumes that server clocks never run backwards, and advance rapidly enough that each transaction can be assigned a distinct timestamp; these assumptions are easy to guarantee (see for example, the discussion in [21]).

The basis of the scheme is the invalidations generated when transactions commit. The fundamental idea is this: if a client running transaction U observes a modification made by transaction T, then it must already have received all the invalidations of T and any transactions T depended on.

The information about invalidations is conveyed to clients using *multistamps*. Each committed transaction has a multistamp that indicates its invalidations and those of all transactions it depends on. A multistamp is a set of tuples <*client, server, timestamp*>; each tuple <*C, S, ts*> means that an invalidation was generated for client C at server S at time *ts*. The timestamp *ts* is the value of S's clock at the time it prepared a transaction that caused invalidations for C.

We assume the obvious merge operation on multistamps: if the two input multistamps contain a tuple for the same client/server pair, the merge retains the larger timestamp value for that pair.

The next two subsections describe the processing at the server and the client, ignoring size issues: multistamps are allowed to grow without bound and so are local tables at the server. Section 5.3 describes how we solve the size problems. Section 5.4 gives an informal argument that the scheme is correct.

5.1 Processing at the Server

Servers have two responsibilities: they must compute multistamps, and they must send them to clients in fetch responses. A fetch response sends a page containing modifications of transactions; at that point we also send the merge of the multistamps of those transactions.

The server maintains the following data structures. The PSTAMP table maps pages to multistamps: the multistamp of a page is the merge of the multistamps of all transactions that modified that page. The ILIST maps clients to invalidation information. Each element of ILIST[C] is a timestamp *ts* and a list of object ids, indicating that these objects were invalidated for C at time *ts*. The VQ records multistamps of committed transactions (along with information about reads and writes of prepared and committed transactions).

Commit Processing. In the prepare phase, if validation of transaction T succeeds, participant S computes multistamp *m* as follows:

1. S initializes *m* to be empty.
2. If the commit of T would cause invalidations for any other clients, S sets *ts* to the current time of its clock. Then for each potentially invalidated client C:
   (a) S adds tuple <*C, S, ts*> to *m*.
   (b) S adds <*ts, olist*> to the ILIST for C, where *olist* contains ids of all objects modified by T that are in pages listed in S's directory for C.
3. For each transaction U that T depends on, S merges VQ[U].mstamp with *m*. The dependencies are determined using S's VQ.

Then S sends *m* in the vote message to the coordinator. If the coordinator decides to commit T, it merges multistamps received from participants to obtain T's multistamp. This multistamp is sent to participants in the commit messages. The participants store it in VQ[T].mstamp. Furthermore, for each page P modified by T, the participant merges this multistamp into PSTAMP[P].

If the coordinator decides to abort, it sends this information to the participants. The participant then removes information about T from the ILIST.

Fetch Processing. When a server receives a fetch message for object *x* on page P, if there is a prepared transaction that modified x, it waits for it to complete. Then it sends the fetch reply, which contains P and also PSTAMP[P]. (Our base system never delays a fetch reply. However, the probability of a delay is very small since the prepare window is very small.)

Invalidations. To produce an invalidation message, the server goes through the ILIST in timestamp order from smallest to largest, stopping when it has processed the entire list, or it reaches an entry for a prepared (but not yet committed) transaction. The ids of all objects in the processed entries are sent in the message along with the largest timestamp contained in the processed entries.

As mentioned, invalidation messages are piggybacked on every message sent to a client and clients acknowledge invalidations. The acknowledgement contains the timestamp *ts* of the associated invalidation message. The server then removes all entries whose timestamps are less than or equal to *ts* from the ILIST.

A client may also request invalidation information. Such a request contains a timestamp *ts*. The server responds by sending back an invalidation message as above except that the timestamp in the message must be greater than or equal to *ts*. It is possible that some entry in the table with a timestamp less than or equal to *ts* exists for a transaction that has not yet committed (it is still prepared); in this case, the server delays the response until the outcome for that transaction is known. Such delays are unlikely because the coordinator sends the phase-two message to participants promptly.

5.2 Processing at the Client

A client C is responsible for using multistamps to ensure that it receives invalidations before their absence could lead to the current transaction viewing an inconsistency.

C maintains two tables that store information about servers it is connected to. LATEST[S] stores the timestamp of the latest invalidation message it has received from server S and REQ[S] is the largest timestamp for S that C is required to hear about. If REQ[S] $>$ LATEST[S], this means S has invalidations for C that C has not yet heard about.

The client also maintains a set CURR that identifies all servers used by the currently running transaction. For each such server S in CURR, it guarantees that LATEST[S] $\geq$ REQ[S]. In other words, for all servers used by the current transaction, the invalidation information is as recent as is required.

When the client receives an invalidation message from server S, it stores the timestamp in the message in LATEST[S].

Client C does the following when a transaction first uses object x:

1. Adds x's server S to CURR.
2. Fetches x if necessary. When the fetch reply arrives it processes the invalidations as described above. Then it updates the information in REQ to reflect the multistamp in the fetch response: for each multistamp entry $< C,\ R,\ ts >$ such that *ts* is larger than REQ[R], it stores *ts* in REQ[R].
3. If LATEST[R] $<$ REQ[R] for some server R in CURR, it sends an invalidation request to R (requesting R to reply with a message timestamped same as or later than REQ[R]), waits for the response, and then processes it.

Note that invalidation processing in steps 2 and 3 can cause the transaction to abort (if the transaction had already used an invalidated object).

The main point to notice here is that the processing overheads associated with multistamps are minimal (assuming multistamps are small). The main impact on system performance is in step 3 when an invalidation request is sent. When this happens there is an extra message compared to AOCC, and the current transaction is delayed, i.e., there is a *stall*. A stall can occur when a fetch completes or when a server is accessed for the first time in a transaction; note that the latter type of stall can be attributed to a fetch that occurred previously at that client. In Section 6, we evaluate the cost in terms of the *stall rate*, i.e., percentage of fetches that resulted in stalls.

5.3 Truncation

Now we discuss how to keep multistamps small and also how to keep the VQ and PSTAMP tables small. All of our optimizations rely on time: we remove "old" tuples from multistamps. To account for removed tuples, a multistamp *m* also contains a timestamp *m.threshold*; *m.threshold* is greater than or equal to timestamps of all tuples that have been removed from *m*. The threshold allows us to compute an *effective multistamp* EFF(*m*) containing a tuple $< C,\ S,\ ts >$ for every client/server pair, where *ts* is the timestamp in the tuple for C/S in *m* if one exists and otherwise, *ts* is *m.threshold*.

As mentioned in Section 4, clients receive invalidation information from servers within the timeout period of when it was generated. This communication implies that tuples containing a timestamp *ts* that is older than a server's current time by more than this period are not useful, since by now those old invalidations will almost certainly have been propagated. Therefore, such old entries are *aged*, i.e., automatically removed from multistamps.

Simple aging of tuples may not be enough to keep multistamps small, so we also *prune* the multistamp by removing tuples that are not old. The system bounds the size of multistamps: whenever a multistamp *m* exceeding this size is generated, it is immediately pruned by removing some tuples. Pruning occurs in two steps: First, if there is more than some number of tuples concerning a particular server, these tuples are removed and replaced by a *server stamp*. Then, if the multistamp is still too large, the oldest entries are removed and the threshold is updated. A server stamp is a pair <*server, timestamp*>; EFF(m) expands a server stamp into a tuple for each client for that server and timestamp.

The VQ and PSTAMP are also truncated. Retaining information in the VQ about transactions that committed longer ago than the timeout period is also not useful. Whenever the multistamp of a transaction contains no tuples (i.e., it consists only of a threshold), it is dropped from the VQ. The VQ has an associated multistamp *VQ.m* that is greater than or equal to the (effective) multistamps of all transactions dropped from VQ. Information is dropped from PSTAMP in the same way, with information about multistamps of dropped entries merged into PSTAMP.*m*, a multistamp associated with PSTAMP. Note that for both the VQ and PSTAMP, we can remove entries earlier if we want; all that is needed is to update the associated multistamp properly. Thus, a server only maintains entries for recently committed transactions in the VQ and only information about recently modified pages in PSTAMP.

A server initializes the multistamp m for a transaction to *VQ.m*. When a fetch request for page P arrives and PSTAMP contains no entry for P, the fetch response contains PSTAMP.*m*.

When a client receives a multistamp m, it computes EFF(m) and proceeds as described in Section 5.2.

The result of these optimizations is a loss of precision in multistamps, which may in turn lead to more fetch stalls. For example, in pruning the multistamp, the server may have removed an entry <*D, S, ts*> concerning some client D different from C. When that multistamp arrives at C, C may block while waiting for invalidation information from S even though S has no invalidations for it.

When an invalidation request arrives at a server it might contain a timestamp *ts* that is larger than what is stored in any entry in the ILIST for that client. To handle such a situation, the timestamp in the invalidation response will be the value of the server's current clock. If the current value of the clock is less than *ts*, the server will wait for it to advance. This situation is extremely unlikely, since it occurs only if clocks are out of synch.

Every message sent to the client contains a piggybacked invalidation message. This message is as described earlier except that if the whole ILIST is being sent, or if the ILIST is empty, the current clock value is sent as the timestamp. This timestamp allows the clients to avoid stalls.

5.4 Correctness

The correctness of our scheme depends on two properties: (i) multistamps reflect dependencies properly; (ii) multistamps flow and are acted upon in a timely way.

Step (2a) of commit processing in Section 5.1 ensures that the multistamp for a transaction T contains tuples for all invalidations caused by it and step (3) ensures that T's multistamp contains all tuples in multistamps of transactions T depends on. The fact that we delay a client that fetches an object modified by T while T is prepared and the fact that the coordinator sends the merged multistamp in its commit decision ensures that all of T's read-write participants have the complete invalidation information for T and the transactions it depends on. By induction on the dependency relation we know that T's multistamp contains tuples reflecting all invalidations of T or any transaction it depends on. Truncation preserves dependencies because, for *m'*, the truncated form of m, the timestamp for any client/server pair in EFF(*m'*) is greater than or equal to the timestamp for the pair in EFF(m).

Multistamps flow in a timely way. The only way a transaction U comes to depend on some other transaction T is by reading a modification of T. This can occur only if its client fetches a page modified by T. But when such a fetch occurs, the multistamp of T is sent to the client (in the multistamp of the returned page); the client records this information (in REQ). The REQ and LATEST data structures are used by a client to maintain the invariant that its cache is up-to-date with respect to all servers in the current transaction. Whenever the cache gets new information (i.e. a page is fetched) or a server is accessed for the first time in the current transaction, the client ensures that REQ[S] $\leq$ LATEST[S] for all servers S used by the current transaction.

6 Performance Evaluation

It would be nice to determine stall rates and the effects of truncation analytically, but these effects are dependent on workloads and also connectivity among clients and servers, and are not amenable to analysis. Instead, we study the effects via simulation. This section describes the simulation experiments and results. The results show that the stall rate is low even in workloads that stress the system, that truncation keeps multistamps small with little impact on stall rate, and that aging alone is not sufficient for truncation in these workloads. The section also discusses the cost of supporting causality and presents an optimization that further reduces stalls.

6.1 Simulator Model

The simulator allows us to control the number of clients and servers, their connectivity, and also workload issues such as how the database is distributed among the servers, how clients share, and what transactions are like. The details of the algorithm are simulated precisely; we replaced client-server tuples in multistamps with server stamps when there were at least 10 entries for the same server. We constructed the simulator and workloads starting from earlier concurrency control studies [13, 6]. These studies were performed for a single-server, multi-client system; we extended them to a distributed database with multiple servers. We ran experiments with varying values of simulator parameters; we present results for a particular setup and mention results of other experiments along the way. Figure 2 describes the parameters of the study.

Connectivity Parameters	
Servers in cluster	2
Client in cluster	20
Total clusters in system	10
Client connections with servers	4
Preferred servers for each client	2
Network Parameters	
Network Bandwidth	155 Mbps
Fixed network cost	6000 instr.
Variable network cost	7168 instr./KB
Client/Server Parameters	
Client CPU speed	200 MIPS
Server CPU speed	300 MIPS
Client cache size	50% of IUP
Server cache hit ratio	50%
Disk read time	16 msec
Database and Transaction Parameters	
Object size	64 bytes
Page size	4 KB
Objects per page	64
In-use pages or IUP	1250
Single server transactions	80%
Read think time	64 μsec/object
Write think time	128 μsec/object
Access per transaction	200 objects
Write accesses	20%
Timeout period	0.5 sec

Figure 2: Summary of Parameter Settings

6.1.1 Connectivity Model

We divided clients and servers into clusters each containing 2 servers and 20 clients. There are 10 clusters in the system. Each client is connected to 4 servers of which 2 are "preferred": most accesses by this client go to these servers. A client's preferred servers are in its cluster; the non-preferred servers are chosen randomly from all the clusters. Thus, each server has 20 clients that prefer it and 20 (on average) other clients. In our experiments each server receives about 1 transaction from a non-preferred client for every 5 transactions received from preferred clients; thus non-preferred clients are not very idle from a server's point of view. Figure 3 shows a typical setup.

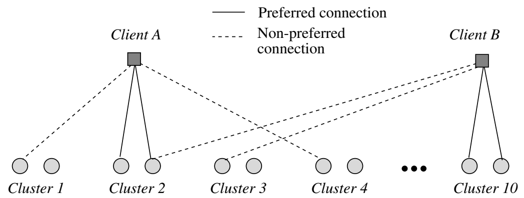

Figure 3: Client connections in a simulator run.

This setup models a realistic situation where clients mostly access their local server (e.g., on their LAN). To stress our scheme, we used 2 preferred servers per client instead of one: as a client switches between the 2 preferred servers, the likelihood of stalls increases due to inter-transaction dependencies created for the client by local causality. Furthermore, the connectivity of the system is high; each client has two random connections to non-preferred servers in other clusters. Thus, a client can "spread" the multistamp from its clusters to other clusters and vice-versa. Increase in multistamp propagation can lead to eager truncation and unnecessary stalls. We chose 10 clusters to model a system in which multistamps must be truncated; otherwise, they would contain 800 entries (since there are 800 client-server connections in the setup).

6.1.2 Network Model

We used an abstract model of the network as in [13]. Apart from time spent on the wire, each message has a fixed and variable processor cost for sending/receiving the message. Our network parameters were obtained from the experimental results reported for U-Net [29] running over a 155 Mbps ATM.

6.1.3 Client/Server Parameters

Client processors run at 200 MIPS and the server CPU at 300 MIPS; these values reflect typical processor speeds expected in the near future. Disks are not modeled explicitly but we assume a server cache hit ratio of 50%. This value is much higher than hit ratios observed in real systems [5, 24]. A lower server cache hit ratio would reduce the stall rate since transactions would take longer to execute and thus dependencies would spread more slowly (we ran an experiment to confirm this). Furthermore, a lower hit rate would reduce the relative impact of stalls on overall execution time. The costs for reading and writing an object are about 64 μsec and 128 μsec respectively. These times are based on the observation that a transaction operates on an object for some time when it accesses it, e.g., the time spent per object in the OO7 benchmark by Thor [19] is around 35 μsec (and OO7 methods do not perform any significant amount of work). Other costs such as those for validation, cache lookup, etc., are not shown due to lack of space; they are negligible compared to the access and fetch costs. The timeout period is chosen to be 0.5 second. Clock skews are not modeled since they are insignificant compared to the timeout period; the Network Time Protocol [22] maintains clock skews that are less than 10 milliseconds on LANs and WANs [23].

6.1.4 Database and Transaction Parameters

The database is divided into pages of equal size. Each simulator run accesses a small subset of the database; this set is called *in-use pages* of the run. As shown in Figure 2, there are 1250 database in-use pages per server.

In all workloads, 80% of the transactions are single server and the rest are multi-server (with a bias towards fewer number of servers in a transaction). To generate a transaction's accesses, we first generate the number of servers for the transaction and then choose the servers. A preferred server is chosen 90% of the time for transactions with one or two servers. For a higher number of servers, we choose both preferred servers and then choose randomly from the remaining non-preferred servers. Each transaction accesses 200 objects on average; 10 objects are chosen on a page resulting in 20 pages being accessed in a transaction. In a multi-server transaction, the number of accesses are equally divided among the servers. 20% of the accesses are writes.

The client cache size is 875 pages; cache management is done using LRU. The client can potentially access 5000 pages (1250 from each server). In single-server studies [13, 6] the client cache was 1/4 of the in-use pages, so it might seem that our cache is too small. However, we observed that more than 85% of accesses are to preferred servers, i.e., to 2500 pages. Thus, a client cache size between 625 and 1250 pages, with a bias towards 625, is in line with earlier work.

The total in-use database in our system is more than twice the size used in [13, 6]. This means we have lower contention (less than half) than what was observed in those studies. We therefore also ran experiments with a smaller database (to make contention levels similar) and observed a stall rate increase of about 50-75%.

As before, our parameters are designed to stress the lazy scheme. Although real systems are dominated by read-only transactions [9, 26], we don't have any since otherwise invalidations would be generated with very low frequency making stalls highly unlikely. Also, we have a relatively high percentage of multi-server transactions (20%); 11.5% of our transactions use two servers and 8.5% use more than two servers. Benchmarks such as TPC-A and TPC-C [28] have fewer than 10-15% multi-server transactions and these transactions involve two servers; other researchers have also reported two-server transactions to be common for distributed transactions [26].

6.1.5 Workloads

We now discuss the different workloads used in our study; these workloads have been used in the past to compare the performance of various concurrency control mechanisms [13, 6]. LOWCON is intended to be a realistic low-contention workload; the others are intended to stress the system.

LOWCON. This workload models a realistic system with low contention (like those observed in [16, 27]). Each client has a private region of 50 pages at each preferred server. Each server has a *shared* region of 1200 pages. 80% of a client's accesses go to its private region; the rest go to the shared regions at its connected servers. Thus, in this workload, a client can access 1250 pages at a preferred server and 1200 pages at a non-preferred server (note that private regions of other clients are not accessed by a client).

SKEWED. As in LOWCON, each client has a private region of 50 pages at each preferred server; together these regions consume 1000 pages. Each server has a *shared* region containing 250 pages. 80% of a client's accesses go to its private region; 20% of the accesses go to the rest of the database (or RDB) region consisting of all other pages at its connected servers, including private regions of other clients. This workload models the case when a client has affinity for its own objects and is used for checking contention with skewed sharing patterns.

HOTSPOT. This is the same as SKEWED except there is a small region of 50 pages added to the database at each server (the shared region is thus 200 pages at each server). 10% of accesses go to the small region, 80% to the private region and 10% to the RDB. The small region might be the top of a naming hierarchy that is accessed often by clients; to capture the fact that such a region will not be modified often, only one in every ten transactions modifies the small region. This workload has the richest access patterns; it includes both uniform sharing (all clients access the small region uniformly) and skewed sharing (one client accesses its private region more frequently than other clients). The workload is intended to stress the lazy scheme because of high contention in the small region.

We could have added a small region to LOWCON, but in practice modifications to such a region are extremely rare and therefore its impact on lazy consistency would be insignificant.

HICON. This is very unrealistic workload used in concurrency control studies to model high contention. There are two regions at each server — a "hot" 250 page region that is accessed 80% of the time and a 1000 page "cold" region that is accessed 20% of the time. Both hot and cold regions are shared uniformly among clients.

6.2 Basic Experimental Results

Figure 4 shows results of experiments as the maximum size for multistamps increases from zero entries (just the threshold timestamp) to the size reached when just aging is used. (A system that uses just the threshold timestamp is using Lamport clocks [18].) The X axis represents increasing multistamp size; the Y axis shows the stall rate. The number in parenthesis after the workload name gives the number of fetches per transaction. The results show that even for small multistamps, the percentage of stalls is low. Low stalls translate into extremely low cost as discussed in Section 6.2.2. Furthermore, we can decrease stall rates by a factor of two, as shown in Section 6.4.

LOWCON has relatively few misses (all due to the shared region) and few stalls. Since contention is low on the shared region, the stall rate is extremely low. Most misses in SKEWED and HOTSPOT are due to accesses to the RDB region; HOTSPOT also has coherency misses due to the

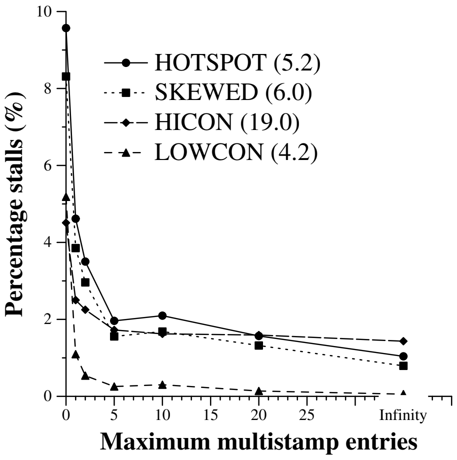

Figure 4: Percentage of fetch stalls as the maximum multistamp size is increased

small region. The stall rate in HOTSPOT is higher than in SKEWED because there is more contention on the small region. Furthermore, since the client can cache a higher percentage of this region than the RDB region in SKEWED, more invalidation entries are generated in HOTSPOT. As a result, there is more truncation, which leads to a higher stall rate.

HICON has very high contention and a high number of stalls. However, the number of fetches is also high because there is a large number of coherency misses in the hot region and capacity misses in the cold region and therefore the stall rates are low. If we had a large enough cache that there were no capacity misses in HICON, the stall rate would be about 2.6% (it will be lower than this since some of the capacity misses will be replaced by coherency misses).

Figure 5 shows multistamp size when only aging is used. These multistamps are large and therefore pruning is needed. As shown in Figure 4, pruning to a reasonable size does not increase stall rate significantly.

Workload	Multistamp entries
LOWCON	27.5
SKEWED	117.9
HOTSPOT	135.8
HICON	201.7

Figure 5: No. of multistamp entries with aging

6.2.1 Additional Experiments

This section discusses results of some other experiments we ran to determine what happens to the stall rate as parameters vary. These experiments used HOTSPOT since it has uniform and skewed sharing patterns, and because it stresses the system.

We changed the topology so that one of the 2 preferred servers was in a different cluster. The stall rate decreased since the likelihood of a client depending on a multi-server transaction that involved the *same* servers decreased (the probability of two clients sharing multiple preferred servers is lower). In another experiment, we decreased the percentage of accesses to the preferred servers; the stall rate decreased as the percentage of accesses to the preferred servers was reduced, for similar reasons.

We varied client cache sizes and observed that the stall rate did not change. As cache sizes were increased, the number of fetches remained the same; capacity misses were replaced by coherency misses.

We ran an experiment where a client had more than 2 non-preferred servers; we varied the number of non-preferred servers from 3 to 9, and allowed transactions to use all connected servers. The stall rate for the 5-entry multistamp increased steadily from 2% to 6.3%. Additional experiments showed that this effect was due to transactions that used large numbers of servers rather than the number of connections per client. Of course, as discussed in Section 6.1.4, transactions that use 8 or 9 servers are highly unlikely in practice.

We ran an experiment where there were 15 clusters instead of 10 (giving 300 clients and 30 servers). The stall rates were the same. In another experiment, we changed the number of clients per cluster from 20 to 30, i.e., each server had an average of 60 client connections. The stall rates for this setup were also similar.

6.2.2 Impact on Performance

We ran experiments with lazy consistency enabled and disabled and observed that the transaction latency/throughput were the same. Very small stall rates have little impact on transaction cost since other costs dominate. Each fetch involves a roundtrip with the reply being a big message (4KB); in contrast, a stall results in a small message roundtrip. On an ATM network, a stall takes 300 μsecs, a fetch from the server cache is approximately 650 μsecs (using U-Net numbers), and a disk access is around 16 msec. For example, consider HOTSPOT with a 5-entry multistamp and assume 50% of fetches hit in the cache. Using the stall rate of 2% shown in Figure 4, we have an increase in the cost of running transactions of less than 0.08%; with the simple optimization discussed in Section 6.4, we get a stall rate of only 0.8% (see Figure 8), and the increase is less than 0.03%.

Lazy consistency also has little cost in space or message size. A multistamp entry requires approximately 12 bytes of storage, and multistamps typically contain fewer entries than a given bound (e.g., in HOTSPOT, a 20-entry multistamp actually contains approximately 10 entries). Therefore, multistamps are cheap to send in messages and the memory requirements for storing them are low. The memory cost for other data structures are also low since we add only a small amount of information over what is needed for concurrency control purposes.

We compared the abort rate with and without lazy consistency and discovered that it did not change. This indicates that in these workloads invalidation requests are almost always due to false sharing rather than true sharing, and that transactions would rarely observe inconsistencies in our base system. Nevertheless, lazy consistency is worth supporting since it provides a much cleaner semantics to programmers: they can count on never seeing inconsistencies.

6.3 The Cost of Causality

This section evaluates the cost of supporting local causality by comparing its stall rate with what happens in a system that supports lazy consistency without any causality. It also evaluates the additional cost of supporting global causality. The results are shown in Figure 6; the results are for a 5-entry multistamp. We can see that the stall rate approximately doubles when support for local causality is added to lazy consistency, and it increases further when support for global consistency is added.

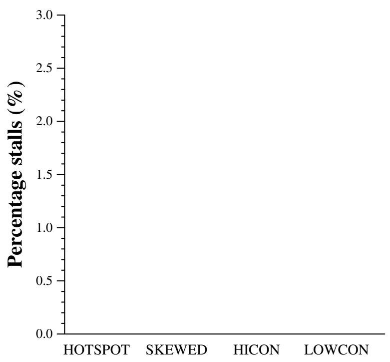

Figure 6: Breakdown of stall rate for local and global causality

Lazy consistency without local causality is implemented by having the client maintain a table SSTAMP that maps servers to multistamps; every time the client receives a fetch reply from server S, it merges the multistamp in the message into SSTAMP[S]. Furthermore, rather than a global REQ table, REQ stores information only for the current transaction. REQ is empty when the transaction starts. The first time the transaction accesses an object from server S, it uses SSTAMP[S] to update entries for other servers used by the transaction in REQ, and then sends invalidation requests to these other servers if necessary. After commit, REQ is merged into SSTAMP[S] for all servers S accessed by the current transaction T (to capture T's dependencies).

In a system with just lazy consistency, single-server transactions cannot stall, and even in a multi-server transaction, a client's requirements are due only to the servers used in that transaction. Thus, the following situation must occur for a client to stall: Client C fetches page P at server S in a multi-server transaction that also involves server R. Page P must have been modified recently by a multi-server transaction that also involved server R and that caused an invalidation for C. The likelihood of a multi-server transaction (fetching P) depending on another "temporally close" multi-server transaction is low. The fact that we keep track of exact dependencies at validation ensures that a preparing transaction depends on very few recent transactions (in HOTSPOT, this was less than 1.5 and in HICON it was around 4). Also, 80% of these transactions are single-server so that the multistamp of a preparing transaction will mostly contain entries for this server. As a result, when P is fetched, P's multistamp contains entries for other servers mostly due to the last transaction that modified it.

When local causality is supported, there are more stalls since a transaction can stall due to requirements generated in an earlier transaction. For example, without causality single-server transactions never stall but now they do; we observed that with local causality about half the total stalls in HICON and HOTSPOT were stalls in a single-server transaction.

Global causality is supported by adding the following to the implementation described in Section 5: the coordinator sends a transaction's multistamp to the client in the commit reply; the client sends its current multistamp to the server at the next commit request; and the server merges it into the multistamp generated for that transaction. Global causality further increases the stall rate because clients now act as "propagators" of multistamps. Without global causality, multistamp information was propagated across servers only through multi-server transactions. Now when a client "switches" servers, it propagates the multistamps (generated due to its previous commits) to the new servers. Also, faster propagation causes more pruning to occur (since the multistamp has more entries); the threshold is raised, resulting in more stalls.

6.4 Optimizations

6.4.1 Reducing Switch Stalls

With local causality, we observed that 70-80% of stalls were *switching stalls* that occurred the first time client uses a server it has not used in the recent past. Switches are frequent in our system since we select servers randomly for every transaction. In a real system, a client will use a server or a set of servers for some time before switching to a different server; thus switching will be relatively infrequent and stalls are likely to be smaller than what we observed.

Workload	**NORMAL**		**ALL**		**PREF**	
	Stall Rate	Msgs	Stall Rate	Msgs	Stall Rate	Msgs
HOTSPOT	1.96 %	0.10	0.23 %	0.39	0.82 %	0.15
HICON	1.73 %	0.33	0.10 %	0.95	0.37 %	0.54
SKEWED	1.56 %	0.10	0.16 %	0.34	0.57 %	0.14
LOWCON	0.26 %	0.01	0.02 %	0.05	0.11 %	0.02

Figure 7: Tradeoff involved in sending background invalidation requests to reduce the stall rates

A good way to reduce switching stalls is to send invalidation requests in the background, for example, in parallel with a commit request. That way, the client will be up to date for all servers when the next transaction starts. However, since a client does not access non-preferred servers frequently, it may not be worthwhile sending installation requests to them. Figure 7 shows the results of this optimization (for 5-entry multistamps). Here ALL is the case in which invalidation requests are sent to all out-of-date servers at commit, and PREF is the case in which these requests are sent only to out-of-date preferred servers; NORMAL is the case where no background invalidation requests are sent. The "Msgs" column gives the total number of invalidation messages (both background and foreground) sent per transaction for each scheme. The main point to note here is that PREF works very well; only slightly more messages are sent (over NORMAL), yet the stall rate drops significantly.

The results for all multistamp sizes for PREF are given in Figure 8; we can see that the optimization reduces stall rate by approximately a factor of two (compared with Figure 4). This figure shows the true performance of our scheme. For all the stressful workloads, the stall rate is less than 1%; for the realistic workload, the stall rate is around 0.1%.

Note that a client can use heuristics based on its history of previous transactions to determine whether a server is preferred or non-preferred.

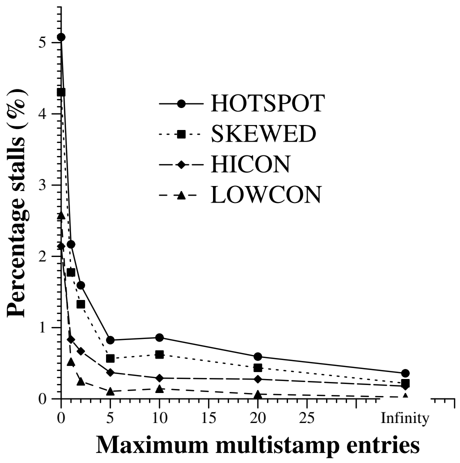

Figure 8: Percentage of fetch stalls when messages are sent to preferred servers at end of transaction

6.4.2 Using Server Stamps

The current system uses both client-server stamps and server stamps in multistamps; it replaces client-server stamps with server stamps when the number of stamps for a server exceeds a bound (10 in our system). We also ran experiments where only server stamps were used. We found that the two systems ran identically when multistamp size (in number of entries) was held constant. However, multistamps containing only server stamps are smaller than those containing client-server stamps, and therefore we can get the same low stall rates with less space and smaller messages using server stamps.

We believe, however, that our workloads are biased towards server stamps since all clients that share a server also share pages at that server, so that page modifications are likely to cause invalidations for many clients. Therefore, there is not much information loss when client-server stamps are replaced by server stamps. Instead, consider two clients C and D that share pages from server S and also both use R but do not share any pages at R. If client C fetches a page from S that was modified in a multi-server transaction by client D, C can stall unnecessarily due to invalidations generated by D for some other client. Depending on the sharing patterns in the system, the client-server stamp system will dynamically replace some client-server stamps with a server stamp. Thus, if the above sharing pattern occurs, the client-server stamp system may be able to handle it better than the server-stamp system. This pattern may be quite likely in practice, and therefore client-server stamps seem like a good idea since they are not very expensive.

6.5 Eager scheme

Lazy consistency can also be provided by an *eager* scheme that delays installing new object versions until all old versions have been removed from client caches. Participants send the list of invalidations to the coordinator in their vote. The coordinator sends the commit decision to the client and initiates an *invalidation phase* in which it sends invalidations to clients on behalf of participants. After it has received replies from these clients, it sends the commit decision to the participants. (The invalidation phase is similar to communication in other concurrency control schemes, e.g., callbacks in [6], except that in those schemes, communication occurs in the foreground.)

The eager scheme avoids fetch stalls entirely, but at the cost of extra messages and an extra commit phase. For example, in HOTSPOT an average of 20 clients are invalidated in every transaction (the number of invalid objects per client is low, e.g., 10), resulting in 20 extra roundtrip messages per transaction with the eager scheme. Since our results show that the stall rate is low using the lazy approach, we believe the extra cost of the eager approach is not justified.

7 Conclusions

This paper has presented a new scheme for guaranteeing that transactions in a client/server system view consistent state while they are running. The scheme is presented in conjunction with an optimistic concurrency control algorithm, but could also be used to prevent read-only transactions from conflicting with read/write transactions in a multi-version system [7].

The scheme is lazy about providing consistency. It simply gathers information as transactions commit, and propagates information to clients in fetch replies. Clients use information only when necessary — to ensure that the current transaction observes a consistent view.

The scheme is based on multipart timestamps. Today the utility of multistamps is limited because their size is proportional to system size and therefore they don't scale to very large systems. Our scheme solves this problem by using real times in multistamps. Using time allows us to keep multistamps small by truncating them without loss of information and with minimal impact on performance: we remove older information that is likely to already be known while retaining recent information. We assume clocks are loosely synchronized; loss of synchronization affects only performance and not correctness.

The paper shows that our approach has extremely low costs. Cost takes the form of stalls — times when a client transaction must be delayed because some dependency information is needed to ensure consistency. Our experiments show that small multistamps are sufficient to make stalls rare in all workloads and extremely rare in realistic low-contention workloads. Furthermore, low stall rates translate into extremely low impact on the cost to run transactions.

Acknowledgements

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136.

We are grateful to Chandrasekhar Boyapati for carefully reading this paper and discussing various aspects of the scheme with us. We would also like to thank Andrew Myers and the referees for their helpful comments.

References

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *SIGMOD*, May 1995.

[2] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. Distributed Multi-version Optimistic Concurrency Control with Reduced Rollback. *Distributed Computing*, 2(1), 1987.

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

[4] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3), August 1991.

[5] M. Blaze. Caching in Large-Scale Distributed File Systems. Technical Report TR-397-92, Princeton University, January 1993.

[6] M. Carey, M. Franklin, and M. Zaharioudakis. Fine-Grained Sharing in a Page Server OODBMS. In *Proc. SIGMOD Int'l Conf. on Management of Data*, pages 359–370. ACM Press, 1994.

[7] A. Chan and R. Gray. Implementing Distributed Read-Only Transactions. *IEEE Transactions on Software Engineering*, 11(2), 1985.

[8] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. Session Guarantees for Weakly Consistent Replicated Data. Technical Report CSL-94-9, Xerox Parc, Palo Alto, CA, 1994.

[9] H. Garcia and G. Weiderhold. Read-Only Transactions in a Distributed Database. *ACM Transactions on Programming Languages and Systems*, 7(2), June 1982.

[10] S. Ghemawat. The Modified Object Buffer: A Storage Management Technique for Object-Oriented Databases. Technical Report MIT/LCS/TR-666, MIT Laboratory for Computer Science, September 1995.

[11] J. N. Gray. Notes on Database Operating Systems. In R. Bayer, R. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, number 60 in Lecture Notes in Computer Science, pages 393–481. Springer-Verlag, 1978.

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

[13] R. Gruber. *Optimism vs. Locking: A Study of Concurrency Control for Client-Server Object-Oriented Databases*. PhD thesis, M.I.T., Cambridge, MA, 1997.

[14] T. Haerder. Observations on Optimistic Concurrency Control Schemes. *Information Systems*, 9(2):111–120, June 1984.

[15] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of 19th Int'l Symp. on Computer Architecture*, pages 13–21, May 1992.

[16] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *SOSP*, 1991.

[17] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing High Availability Using Lazy Replication. *ACM Transactions on Computer Systems*, 10(4), November 1992.

[18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7), July 1978.

[19] B. Liskov et al. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proc. SIGMOD Int'l Conf. on Management of Data*, 1996.

[20] B. Liskov, R. Scheifler, E. Walker, and W. Weihl. Orphan Detection. Programming Methodology Group Memo 53, Laboratory for Computer Science, MIT, February 1987.

[21] Barbara Liskov. Practical Uses of Synchronized Clocks in Distributed Systems. *Distributed Computing*, 6, August 1993.

[22] D. L. Mills. Network Time Protocol (Version 3) Specification, Implementation and Analysis. Network Working Report RFC 1305, March 1992.

[23] D. L. Mills. The Network Computer as Precision Timekeeper. In *Proc. Precision Time and Time Interval (PTTI) Applications and Planning Meeting*, December 1996.

[24] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems - or - your cache ain't nuthin' but trash. In *USENIX Winter Conference*, January 1992.

[25] D. S. Parker et al. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering*, SE-9(3), May 1983.

[26] G. Samaras, K. Britton, A. Citron, and C. Mohan. Two-Phase Commit Optimizations in a Commercial Distributed Environment. *Distributed and Parallel Databases*, 3, October 1995.

[27] M. Spasojevic and M. Satyanarayanan. An Empirical Study of a Wide-Area Distributed File System. *ACM Transactions on Computer Systems*, 14(2), May 1996.

[28] Transaction Processing Performance Council (TPC). *TPC-A Standard Specification, Revision 1.1, TPC-C Standard Specification, Revision 1.0*. Shanley Public Relations, 1992.

[29] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *SOSP*, 1995.

[30] W. E. Weihl. Distributed Version Management for Read-only Actions. *IEEE Transactions on Software Engineering*, SE-13(1), January 1987.