CMU-CS-86-163

Extending Multiversion Timestamping Protocols to Exploit Type Information

Maurice Herlihy
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213
12 November 1986

Abstract

Atomic transactions are a widely-accepted approach to implementing and reasoning about fault-tolerant distributed programs. This paper shows how multiversion timestamping protocols for atomicity can be extended to induce fewer delays and restarts by exploiting semantic information about objects such as queues, directories, or counters. This technique relies on static preanalysis of conflicts between operations, and incurs no additional run-time overhead. This technique is deadlock-free, and it is applicable to objects of arbitrary type.

Index terms: atomicity, fault tolerance, concurrency control, serializability, multiple versions, abstract data types.

This article will appear in the April 1987 IEEE Transactions on Computers special issue on parallel and distributed computation.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract F33615-84-K-1520.

1. Introduction

The data items managed by distributed applications are often subject to consistency constraints that must be preserved in the presence of concurrency and failures. These constraints apply not only to individual data items, but also to distributed sets of data. For example, a distributed banking system might be subject to the constraint that the books balance: money is neither created nor destroyed, only transferred from one ledger to another. A widely-accepted approach to this problem is to ensure that activities are *atomic*: that is, indivisible and recoverable. *Indivisible* means that activities appear to execute in a serial order [10], and *recoverable* means that an activity either succeeds completely or has no effect. Such atomic activities are called *transactions*.

Many techniques have been proposed for implementing atomicity in distributed systems (for a survey, see [1]). This paper considers one such mechanism, a *multiversion timestamping protocol* due to Reed [13], and shows how it can be extended to induce fewer delays and restarts by exploiting semantic information about the object. While conventional approaches consider only *untyped* objects providing read and write operations, our approach explicitly considers *typed* objects such as queues, directories, or counters, that provide a richer set of operations. We propose a systematic technique for preanalyzing conflicts between operations, a simple and efficient mechanism for detecting these conflicts at run time, and we give proofs of correctness and optimality. This protocol is readily integrated with quorum consensus replication, and it can also be used as the basis for optimistic concurrency control. Like most multiversion timestamping protocols, ours is deadlock-free.

2. Related Work

Two-phase locking [2] is a well-known alternative to multiversion timestamping protocols. A direct comparison of the two approaches is difficult, since each permits interleavings the other does not. Locking is subject to deadlocks, while timestamping is not. Timestamping, however, may be subject to livelock, since unfortunate combinations of events may cause a transaction to be repeatedly aborted and restarted. Multiversion timestamping protocols appear to provide better support for long, read-only transactions.

Timestamping protocols are incompatible with two-phase locking; the two protocols cannot be used together in the same system, because they may impose incompatible serialization orders on transactions at different objects. Weihl [16] has developed analytic techniques for characterizing when atomicity mechanisms are compatible. The technique proposed here satisfies a property called *static atomicity*, and is compatible with Reed's protocol.

Papadimitriou and Kanellakis [11] have studied the effect on concurrency of restricting the number of versions retained by multi-version timestamping schemes, and the author [5] has studied how various atomicity mechanisms affect the availability of replicated data.

3. Assumptions and Definitions

A *distributed system* consists of multiple computers (called sites) that communicate through a network. Distributed systems are typically subject to site crashes and communication link failures. A crash renders a site's data temporarily or permanently inaccessible, while a communication link failure causes messages to be lost. A failure is detected when a site that has sent a message fails to receive a response after a certain duration. The absence of a response may indicate that the original message was lost, that the reply was lost, that the recipient has crashed, or simply that the recipient is slow to respond.

A transaction that completes successfully is said to *commit*, otherwise it *aborts*. A transaction that has neither committed nor aborted is *active*. We rely on standard commitment protocols (e.g., [4, 14]) to ensure that a transaction either commits at all sites or aborts at all sites, and we rely on standard techniques to recover from crashes [15].

The basic containers for data are called *objects*. Each object has a *type*, which defines a set of possible *states* and a set of primitive *operations* that provide the only means to create and manipulate objects of that type. For example, a FIFO queue might be represented by an object of type *Queue* providing the following operations. *Enq* places an item at the end of the queue:

```
    Enq = Operation(i: Item)
```

and *Deq* returns the item at the head of the queue:

```
    Deq = Operation() Returns(Item)
```

Deq is *partial:* it is undefined when the queue is empty. A transaction invoking a partial operation is blocked until the operation can return legally.

4. Informal Overview

This section gives an informal description of our technique. A formal description appears in Section 5. We begin with a simplified review of Reed's scheme [13] as adapted for the Swallow repository [12]. For brevity, we consider only single-level transactions, and we assume that no transaction writes an object more than once.

When a transaction is created, it is issued a logical timestamp called its *pseudotime*. Transactions will be serializable in pseudotime order. An object is represented as a sequence of *versions*, each labeled with a pseudotime. Versions associated with active transactions are called *possibilities*. Each object also keeps track of *last – read*, the latest pseudotime of a transaction to read the object.

A transaction reads an object in the following steps:

1. Select the version with the latest pseudotime preceding the reader's.
2. If that version is a possibility (i.e., the writer has not yet committed), then delay the reader until the writer commits or aborts, and restart the read.
3. Advance *last – read* to the maximum of its current value and the reader's pseudotime.
4. Return the version to the reader.

The reader must be delayed in Step 2 because the value to be read is ambiguous.

A transaction writes an object in the following steps:

1. If the writer's pseudotime precedes *last – read*, then the write is refused, and the writer is aborted and restarted with a later pseudotime.
2. Create a new version tagged with the writer's pseudotime.

The writer must be restarted in Step 1 because it might invalidate a value already read by another transaction. This protocol is deadlock-free because no transaction ever waits for a transaction with a later pseudotime, thus cycles of dependency cannot arise.

The limitations of this protocol can be illustrated by considering the implementation of a FIFO queue. Because Enq and Deq would each be implemented as a Read followed by a Write, this scheme permits no concurrency at all. Let *A* and *B* be transactions such that *A*'s pseudotime precedes *B*'s. If *A* invokes any operation before *B*, then *B* will be delayed, and if *B* invokes any operation before *A*, then *A* will be aborted and restarted. We will show that these constraints are stronger than necessary, and propose an alternative scheme that permits considerably more concurrency.

First, we introduce some terminology. An *operation execution* is written *op(args*)/term(res*)*, where *op* is an operation name, *args** denotes a sequence of argument values, *term* is a termination condition, and *res** is a sequence of results. We use "Ok" for normal termination. The operation name and argument values constitute the *invocation*, and the termination condition and result values constitute the *response*.

An object is not represented as a sequence of versions; instead it is represented as a sequence of operations called a *history* (see Figure 4-3). To execute an operation, a transaction tentatively inserts the new operation in the history, in a position determined by the transaction's pseudotime. If the transaction commits, the operation become permanent, otherwise it is discarded. Synchronization is based on a predefined *conflict* relation between pairs of operations. Each operation has an associated *ratchet lock*, which ensures that no transaction's results are invalidated by a later transaction with an earlier pseudotime, as well as an associated *possibility lock*, which ensures that a transaction is delayed if it depends on an operation executed by an uncommitted transaction. An operation's ratchet lock is the latest pseudotime of a transaction that executed that operation, and its possibility lock is the set of active transactions that have executed that operation.

A transaction *A* executes an operation in the following steps:

1. When the object receives the invocation, it chooses a response after assembling the subhistory of operations executed by committed operations whose pseudotimes precede *A*'s.

2. Let *p* be the new operation, and let *p* conflict with *q*. *A* is delayed while any transaction with an earlier pseudotime holds a possibility lock for *q*. If *A* is delayed, the operation must be restarted at Step 1, since the new state may produce a different response.

3. If *q* conflicts with *p*, and the ratchet lock for *q* exceeds *A*'s pseudotime, then *A* is aborted and restarted with a later pseudotime.

4. Otherwise, *A* is granted a possibility lock for *p*, *p*'s ratchet lock is advanced to the maximum of its current value and *A*'s pseudotime, and the object returns the response to *A*.

Reed's protocol can be viewed as a special case of ours, where versions correspond to Write operations, *last – read* is the ratchet lock for Read, and possibilities are possibility locks for Write. Because Read operations do not affect the results of later operations, they do not need possibility locks, nor do they need to be recorded. Because Write operations cannot be invalidated by later operations, they do not need ratchet locks.

The extent to which this typed scheme induces fewer delays and restarts than the untyped scheme depends on the nature of conflicts. A formal characterization of conflicts is given below; for now we rely on an informal definition: an operation *p* conflicts with an operation *q* if *p* can be invalidated by inserting *q* earlier in a history. For example, two valid but incomparable conflict relations for the FIFO Queue data type are shown in Figure 4-1 and 4-2. In the first relation, Enq operations conflict with no other operations, but Deq operations conflict with all other operations. Here, any number of transactions can execute concurrent Enq operations, but a transaction with a later pseudotime that attempts to execute a Deq will be delayed. In the second relation, Enq operations conflict with one another, and Deq operations conflict with one another, but Enq operations do not conflict with Deq operations, and vice-versa. Here, one transaction may enqueue items while a transaction with a later pseudotime dequeues other items, as long as there are committed items to dequeue. In both cases, a dequeuing transaction never interferes with a enqueuing transaction with a later pseudotime.

	Enq	Deq
Enq		
Deq	Conflict	Conflict

Figure 4-1: First Conflict Relation for Queue

	Enq	Deq
Enq	Conflict	
Deq		Conflict

Figure 4-2: Second Conflict Relation for Queue

```
Enq(x)/Ok()
Enq(y)/Ok()
Deq()/Ok(x)
```

Figure 4-3: A History for a FIFO Queue Object

```
Begin A
Enq(x)/Ok() A
Begin B
Enq(y)/Ok B
Commit A
Deq()/Ok(x) B
```

Figure 4-4: A Schedule for a FIFO Queue Object

5. Formal Model

This section presents formal definitions and correctness arguments.

5.1. Histories and Schedules

In the absence of failures and concurrency, an object's state is modeled by a sequence of operations called a *history*. For example, a history for a FIFO Queue is shown in Figure 4-3. A *specification* for an object is the set of permissible histories for that object. For example, the specification for a Queue consists of histories in which items are dequeued in first-in-first-out order. A *legal* history is one that is included in the object's specification.

In the presence of failures and concurrency, an object's state is given by a *schedule*, which is a sequence of operation executions, *transaction begins*, *transaction commits*, and *transaction aborts*. To keep track of interleaving, a transaction identifier is associated with each step in a schedule. For example, a schedule for a FIFO Queue appears in Figure 4-4, where *A* and *B* are transaction identifiers. The ordering of operations in a schedule reflects the order in which the object returned responses, not necessarily the order in which it received invocations.

We assume schedules satisfy the following *well-formedness* conditions.

1. No transaction executes an operation before it has begun.
2. No transaction executes an operation after it has committed.
3. No transaction both commits and aborts.
4. No transaction begins, commits, or aborts more than once.

(Serial) histories and (concurrent) schedules are related by the notion of *atomicity*. Let $\rightarrow$ denote a total order on committed and active transactions, and let *H* be a schedule. The *serialization* of *H* in the order $\rightarrow$ is the history *h* constructed by reordering the operations in *H* so that if $A \rightarrow B$ the subsequence of operations associated with *A* precedes the subsequence of operations associated with *B*. *H* is *serializable in the order* $\rightarrow$ if *h* is a legal (serial) history. *H* is *serializable* if it is serializable in some order. *H* is *atomic* if the subschedule associated with committed transactions is serializable, and it is *on-line* atomic if committing any set of active transactions yields an atomic schedule.

The schedules produced by the concurrency control mechanism proposed in this paper are *static atomic* [16]: transactions are serialized in the order of their *begin* operations, as observed by a system of logical clocks [9]. Weihl has shown that static atomicity is a *local* property: if each object in a system encompassing multiple objects is static atomic, the system as a whole is atomic.

5.2. Conflict and Serial Dependency

This section gives a formal characterization of what it means for operations to conflict. Let $\succ$ be a relation between pairs of operations, and let h be a legal history. A legal subhistory g of h is $\succ$*-closed* if whenever it contains an operation p it also contains every earlier operation q of h such that $p \succ q$. A subhistory g is a $\succ$*-view* of h for p if g is $\succ$-closed, and if g contains every q of h such that $p \succ q$. Informally, $\succ$ is a *serial dependency* relation if whenever an operation is legal for a $\succ$-view, it is legal for the complete history. More precisely, let "•" denote concatenation:

> **Definition 1:** A relation $\succ$ is a *serial dependency* relation if for all operations p and all legal histories h and g such that g is a $\succ$-view of h for p, $g \bullet p$ is legal implies that $h \bullet p$ is legal.

We show below that our technique is static atomic if and only if the conflict relation between operations is a serial dependency relation. Of primary interest are *minimal* relations having the property that no smaller relation is also a serial dependency relation. The reader can verify that the relations shown in Figures 4-1 and 4-2 are minimal serial dependency relations. It is easy to show that failure to commute is a serial dependency relation, but as illustrated by the FIFO queue example, it is not necessarily minimal.

The notion of serial dependency arises in other contexts as well. For example, serial dependency has also been used for validation in optimistic concurrency control mechanisms [7]. For quorum consensus replication [6, 8], constraints on availability and concurrency are governed by a slightly modified form of serial dependency, defined between invocations and operations rather than between operations.

5.3. The Automaton Model

An object is modeled by an automaton that accepts certain schedules. The automaton's state is defined using the following primitive domains: TRANS is the set of transaction identifiers, OP is the set of operations, PSEUDOTIME is a totally ordered set of pseudotimes. The derived domain HISTORY is the set of sequences of operations. X $\rightarrow$ Y denotes the set of partial maps from X to Y.

An object has the following state components:

Intentions: TRANS $\rightarrow$ HISTORY

Ratchet: OP $\rightarrow$ PSEUDOTIME

Possibility: OP $\rightarrow$ 2^{TRANS}

Clock: PSEUDOTIME

Active: 2^{TRANS}

Committed: 2^{TRANS}

Aborted: 2^{TRANS}

Begin: TRANS $\rightarrow$ PSEUDOTIME

Intentions(A) is the history of operations executed by transaction *A*, initially none. *Ratchet(p)* is the value for *p*'s ratchet lock, initially $-\infty$. *Possibility(p)* is the set of active transactions that hold a possibility lock for *p*, initially none. The *Clock* component models a system of logical clocks, and *Begin(A)* is the pseudotime assigned to *A*. *Active*, *Committed* and *Aborted* keep track of the transactions that are active, committed, and aborted; each is initially empty.

Let $S \subseteq$ Active $\cup$ Committed be a set of transactions $\{A_0,\ldots,A_n\}$ such that $\text{Begin}(A_{i-1}) < \text{Begin}(A_i)$. Define State(S) to be the history constructed by concatenating the intentions lists of the transactions in S in pseudotime order:

$$\text{State}(S) = \text{Intentions}(A_0) \bullet \ldots \bullet \text{Intentions}(A_n)\ .$$

If *A* is a transaction, define:

$$\text{Before}(A) = \{B \mid B \in \text{Committed} \cup \text{Active} \wedge \text{Begin}(B) < \text{Begin}(A)\}$$

$$\text{View}(A) = \text{State}(\text{Before}(A) \cap \text{Committed}) \bullet \text{Intentions}(A)$$

Before(A) is the set of committed and active transactions with pseudotimes earlier than *A*'s, and View(A) is the committed state "observed" by *A*.

Each transition has a precondition and a postcondition. In postconditions, primed component names denote new values, and unprimed names denote old values. To reduce the complexity of the preconditions, we assume all schedules are well-formed. When *A* begins execution:

Post: $\text{Clock}' > \text{Clock}$

$\text{Begin}(A)' = \text{Clock}$

$\text{Active}' = \text{Active} \cup \{A\}$

The clock is advanced, *A* is assigned a pseudotime, and *A* is noted as active.

For *A* to execute operation *p*:

Pre: If $q \succ p$, then Ratchet(q) < Begin(A).

If $p \succ q$, then (Before(A) ∩ Possibility(q))−{A} = ∅.

View(A) • p is legal.

Post: Intentions'(A) = Intentions(A) • p

Ratchet'(p) = max(Ratchet(p),Begin(A))

Possibility'(p) = Possibility(p) ∪ {A}

There must be no conflicting ratchet or possibility locks, and the operation must appear to be legal. The new operation is appended to the intentions list, and the operation's ratchet and possibility locks are set.

For *A* to commit:

Post: Active' = Active − {A}.

Committed' = Committed ∪ {A}.

For p in Intentions(A), Possibility'(p) = Possibility(p) − {A}.

For *A* to abort:

Post: Active' = Active − {A}.

Aborted' = Aborted ∪ {A}.

For p in Intentions(A), Possibility'(p) = Possibility(p) − {A}.

5.4. The Proof

We use the following two Lemmas, which are proved elsewhere [7].

Lemma 2: If $\succ$ is a serial dependency relation, *g* and *h* histories, and *q* an operation such that $g \cdot q$ and $g \cdot h$ are legal, and there is no *p* in *h* such that $p \succ q$, then $g \cdot q \cdot h$ is legal.

Lemma 3: If $\succ$ is not a serial dependency relation, then there exist a history *h* and an operation *p* such that *h* has a $\succ$-view *g* of *p* missing exactly one operation, $g \cdot p$ is legal, but $h \cdot p$ is not.

The following invariant is an immediate consequence of the rules for granting ratchet and possibility locks:

Lemma 4: If Begin(A) < Begin(B), and *A* is active, then no operation executed by *B* conflicts with any operation executed by *A*.

Proof: By a simple inductive argument. The property holds initially, so we show that it is preserved by each transition. Suppose *p* conflicts with *q*. If *A* has executed *q*, then *B* is prevented from executing *p* by the conflicting possibility lock for *q*. If *B* has executed *p*, then *A* is prevented from executing *q* by the conflicting ratchet lock for *p*.

We are now ready for the basic correctness result.

Theorem 5: If the conflict relation is a serial dependency relation, then every schedule accepted by the automaton is on-line static atomic.

Proof: The automaton is on-line static atomic if State(S) is a legal history for every S such that Committed $\subseteq$ S $\subseteq$ Active $\cup$ Committed. The proof is by induction on the length of the accepted schedule. The base case, where the schedule is empty, is trivial. Assume as the induction hypothesis that the automaton has accepted an on-line static atomic schedule.

We now show that the next transition preserves on-line static atomicity. Begin, Commit, and Abort events are immediate. The non-trivial case is when A executes an operation p. Pick an S that includes A, let $h_1 \bullet h_2$ = State(S), and $h_1 \bullet p \bullet h_2$ = State'(S), the state following the transition. $h_1 \bullet h_2$ is legal by the induction hypothesis. View(A) is a $\succ$-view of h_1 for p (Lemma 4). Because the conflict relation is a serial dependency relation, $h_1 \bullet p$ is a legal history. Moreover, no operation in h_2 conflicts with p (Lemma 4), thus $h_1 \bullet p \bullet h_2$ is legal (Lemma 2).

Our scheme is optimal in the sense that no conflict relation that is not a serial dependency relation suffices to ensure static atomicity.

Theorem 6: Any automaton whose conflict relation is not a serial dependency relation will accept a schedule that is not on-line static atomic.

Proof: Assuming $\succ$ is not a serial dependency relation, there exist an operation p and legal histories g and h such that g is a $\succ$-view of h for p missing exactly one operation q, $g \bullet p$ is legal, but $h \bullet p$ is not (Lemma 3). Let $g = a \bullet c$ and $h = a \bullet q \bullet c$. Let A, B, and C be transactions such that Begin(A) $<$ Begin(B) $<$ Begin(C). Transaction A executes a and commits. C then executes $c \bullet p$, and B executes q. If S = {A,B,C}, State(S) is the illegal history $a \bullet q \bullet c \bullet p = h \bullet p$.

6. Remarks

A practical difficulty with all multiversion protocols is the apparent need to keep a potentially unlimited volume of data. In Reed's protocol, one would keep a complete history of versions, and in ours, one would keep a complete history of the object's operations. Reed suggests that although this data might be useful as an audit trail, in practice one could discard sufficiently old versions, rejecting transactions that attempt to read or write with very old pseudotimes. Our scheme can be optimized in a similar way: a prefix of the object's history can be periodically compacted into a single version, and transactions with older pseudotimes are rejected. For example, a Queue can be represented by a single, compact version (i.e., a list of the elements present in the queue) followed by a short sequence of Enq and Deq operations, which are treated as a *differential file* [15]. Other optimizations are straightforward: there is no need to create entries or possibility locks for operations that do not modify the object's state, and there is no need for ratchet locks for operations that return no information.

Our protocol is readily integrated with quorum consensus replication [3, 8]. A *replicated object* is an object whose state is stored redundantly at multiple sites. Replicated objects are implemented by two kinds of modules: *repositories* and *front-ends*. Repositories provide long-term storage for the object's state, while front-ends carry out operations for clients. A client executes an operation by sending the invocation to a front-end. The front-end merges the data from an *initial quorum* of repositories, performs a local computation, and records the response at a *final quorum* of repositories. A *quorum* for an operation is any set of sites that includes both an initial and a final quorum for that operation. Each operation's availability is determined by its set of quorums, thus constraints on quorum assignment determine the range of availability properties realizable by replication.

Because state information for replicated objects is distributed, it is convenient to define conflicts between invocations and events rather than between pairs of events. Each repository keeps a set of ratchet and possibility locks, and the replicated object will be on-line static atomic if and only if the intersection of the conflict relation with the quorum intersection relation must be an (invocation/event) serial dependency relation. (In practice, they would be identical.) This requirement ensures that any conflict will be detected at the non-empty intersection of two quorums. A similar protocol in which the serialization order is determined dynamically is described in detail elsewhere [6].

Techniques proposed elsewhere [7] could be adapted to transform our protocol into a type-specific *optimistic* protocol, in which transactions execute without synchronization, relying on commit-time validation to ensure serializability. Such optimistic protocols would preserve static atomicity, thus they could be used in conjunction with static atomic pessimistic protocols.

This paper has proposed a technique that extends multiversion timestamping protocols to exploit type-specific properties of data objects to induce fewer delays and restarts than conventional techniques employing a simple model of read/write conflicts. The problem of identifying a correct and minimal set of conflicts for an object is shown to be equivalent to the algebraic problem of identifying a minimal serial dependency relation for the data type. This technique relies on static preanalysis of conflicts between operations, and incurs no additional run-time overhead.

References

[1] P.A. Bernstein and N. Goodman.
Concurrency control in distributed database systems.
ACM Computing Surveys 13(2):185-222, June, 1981.

[2] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger.
The Notion of Consistency and Predicate Locks in a Database System.
Communications of the ACM 19(11):624-633, November, 1976.

[3] D.K. Gifford.
Weighted Voting for Replicated Data.
In *Proceedings of the Seventh Symposium on Operating Systems Principles*. ACM SIGOPS, December, 1979.

[4] J. Gray.
Notes on Database Operating Systems.
Lecture Notes in Computer Science 60.
Springer-Verlag, Berlin, 1978, pages 393-481.

[5] M.P. Herlihy.
Comparing how atomicity mechanisms support replication.
In *Fourth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. August, 1985.

[6] M.P. Herlihy.
Availability vs. atomicity: concurrency control for replicated data.
Technical Report CMU-CS-85-108, Carnegie-Mellon University, February, 1985.
Submitted for publication.

[7] M.P. Herlihy.
Optimistic concurrency control for abstract data types.
In *Fifth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 206-217. August, 1986.

[8] M.P. Herlihy.
A quorum-consensus replication method for abstract data types.
ACM Transactions on Computer Systems 4(1), February, 1986.

[9] L. Lamport.
Time, clocks, and the ordering of events in a distributed system.
Communications of the ACM 21(7):558-565, July, 1978.

[10] C.H. Papadimitriou.
The serializability of concurrent database updates.
Journal of the ACM 26(4):631-653, October, 1979.

[11] C.H. Papadimitriou and P. Kanellakis.
On concurrency control by multiple versions.
ACM transactions on database systems 9(1):89-99, March, 1984.

[12] D.P. Reed and L. Svobodova.
SWALLOW: a distributed data storage system for a local network.
In *Proceedings of the International Workshop on Local Networks*. August, 1980.

[13] D.P. Reed.
Implementing atomic actions on decentralized data.
ACM Transactions on Computer Systems 1(1):3-23, February, 1983.

[14] M.D. Skeen.
Crash Recovery in a Distributed Database System.
PhD thesis, University of California, Berkeley, May, 1982.

[15] J.S.M. Verhofstad.
Recovery Techniques for Database Systems.
ACM Computing Surveys 10(2):167-196, June, 1978.

[16] W.E. Weihl.
Data-Dependent concurrency control and recovery.
In *Proc. 2nd Annual Symposium on Principles of Distributed Computing*. August, 1983.