An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases

PHILIP A. BERNSTEIN AND NATHAN GOODMAN
Sequoia Systems, Inc.

In a one-copy distributed database, each data item is stored at exactly one site. In a replicated database, some data items may be stored at multiple sites. The main motivation is improved reliability: by storing important data at multiple sites, the DBS can operate even though some sites have failed.

This paper describes an algorithm for handling replicated data, which allows users to operate on data so long as one copy is "available." A copy is "available" when (i) its site is up, and (ii) the copy is not out-of-date because of an earlier crash.

The algorithm handles clean, detectable site failures, but not Byzantine failures or network partitions.

Categories and Subject Descriptors: H.2.0 [**Database Management**]: General—*concurrency control*; H.2.2 [**Database Management**]: Physical Design

General Terms: Algorithms

Additional Key Words and Phrases: Serializability, distributed databases, replicated databases, continuous operation, transaction processing

1. INTRODUCTION

A replicated database is a distributed database in which some data items are stored redundantly at multiple sites. The main goal is to improve system reliability [1, 20]. By storing critical data at multiple sites, the system can operate even though some sites have failed.

There are two correctness criteria for replicated databases: *replication control*—the multiple copies of a data item must behave like a single copy, insofar as users can tell; and *concurrency control*—the effect of a concurrent execution must be equivalent to a serial one. A replicated database system that achieves replication and concurrency control has the same input/output behavior as a centralized,

This research was supported by the NSF under grant MCS79-07762 and by the Office of Naval Research under grant N00014-80-674.

Authors' address: Sequoia Systems, Inc., 1 Metropolitan Corporation Center, Marlborough, MA 01752.

one-copy database system that executes user requests one at a time [33]. Such behavior is termed 1-serializability [5, 6].

In an ideal world, where sites never fail, there is a simple way to manage replicated data [33]. When a user wishes to read x, the system reads *any* copy of x. When a user updates x, the system applies the update to all copies of x. Concurrency control is by distributed two-phase locking [15].

This paper extends this simple algorithm to an environment where sites fail and recover.

To illustrate the problem, let us extend the simple algorithm in obvious ways and see what goes wrong.

The obvious way to handle site failures is to ignore failed sites. When a user wants to update x, the system applies the update to all copies of x at "up" sites and ignores the copies at "down" sites.

Site recoveries are a little harder. Consider data item x with copies x_a and x_b at sites a and b, respectively. And suppose site a has failed. While the site is down, copy x_a may become out-of-date, because users may update x_b. When site a recovers, the system must bring x_a up-to-date before letting users access it. An obvious way to do this is to copy the value of x_b into x_a.

The examples below show what goes wrong with this algorithm. Example 1 shows a problem caused by site failures; Example 2 shows a problem caused by site recoveries.

Example 1. Consider a database with data items x and y and copies x_a, x_b, y_c, and y_d stored at sites a, b, c, and d, respectively. Consider two transactions: T_1 reads x and writes y; T_2 reads y and writes x. The simple algorithm allows the following execution.

$$\begin{tikzcd} r_1[x_a] \arrow[r] \arrow[dr] & a\text{-fails} \arrow[r] \arrow[dr] & w_1[y_c] \\ r_2[y_d] \arrow[r] \arrow[ur] & d\text{-fails} \arrow[r] \arrow[ur] & w_2[x_b] \end{tikzcd}$$

"$r_1[x_a]$" denotes a read of x_a by T_1; "a-fails" denotes the failure of site a, and so forth. Arrows indicate the order in which events occur. T_1 and T_2 do their reads concurrently; we assume that each read sets a read–lock on the corresponding copy. Then sites a and d fail. Then T_1 and T_2 do their writes. T_1 writes y by writing, and implicitly locking all copies of y at "up" sites. By now, the only such copy is y_c. Note that T_1's write–lock on y_c does not conflict with T_2's read–lock on y_d because y_c and y_d are different copies at different sites. T_2 writes x similarly.

Logically T_1 and T_2 conflict. A correct system must synchronize these transactions to prevent them from running concurrently. But the algorithm does not synchronize them, because sites a and d fail at a crucial moment.

More formally, the execution is incorrect because it does not have the same effect as a serial execution of T_1 and T_2 against a one-copy database. In a serial execution of T_1 and T_2, one or the other comes first. If T_1 comes first, the execution is

$$r_1[x] \longrightarrow w_1[y] \longrightarrow r_2[y] \longrightarrow w_2[x].$$

The important fact to note is that T_2 reads the value of y written by T_1. By contrast, in the original execution, T_2 reads some prior value of y. Since T_2 reads a different value of y in each execution, the executions are not "equivalent"; i.e., in general, they do not have the same effect. A similar argument holds if T_2 comes first in the serial execution.

Example 2. Consider a database with data-items x and y and copies x_a, x_b, x_c, and y_d at sites a, b, c, and d, respectively. Consider two transactions: T_1 writes x then reads y; T_2 reads x then writes y. And suppose site b is recovering from failure as the example begins. The algorithm allows the following execution:

$$\begin{tikzcd}
w_1[x_c] \arrow[rr] & & w_1[x_a] \arrow[r] & r_1[y_d] \\
r_{\text{in}}[x_a] \arrow[r] & w_{\text{in}}[x_b] \arrow[ur] \arrow[dr] & & \\
& & r_2[x_b] \arrow[r] & w_2[y_d] \arrow[uu]
\end{tikzcd}$$

"$r_{\text{in}}[x_a]$" and "$w_{\text{in}}[x_b]$" are operations invoked by the system to bring x_b up-to-date. (The subscript "in" stands for "include"; this is explained in Section 3.1.)

When the example begins, x_a and x_c are the only copies of x at up sites. So to write x, T_1 only writes these copies. However, site b is recovering in parallel with T_1's execution. By the time T_1 finishes, copy x_b is also up. The algorithm makes no provision for this case.

Proceeding as in Example 1, we can show that the execution is not equivalent to a serial execution of T_1 and T_2 on a one-copy database. The serial execution in which T_1 comes first is

$$w_1[x] \longrightarrow r_1[y] \longrightarrow r_2[x] \longrightarrow w_2[y].$$

Note that T_2 reads the value of x written by T_1, whereas in the original execution, T_2 reads a prior value of x (copied from x_a into x_b). Since T_2 reads a different value of x in each execution, the executions are not equivalent. A similar argument holds if T_2 comes first.

This paper presents an embellished form of this simple algorithm, called an *available copies algorithm.* For each data item x, the algorithm maintains a directory listing the copies of x that are "available for use." If a transaction operates on x, the site running the transaction must store a copy of this directory (except as discussed in Section 3.7). When a transaction reads x, the algorithm consults the directory and reads some copy listed there. When a transaction writes x, the algorithm consults the directory and writes all copies. The algorithm runs special *status transactions* to keep directories up-to-date as sites fail and recover.

Related work includes [1–3, 5, 9, 11, 13, 14, 17, 18, 20, 30, 32, 33].

Our algorithm is a close relative of the one used in Computer Corporation of America's Adaplex system [10, 18]; other related algorithms appear in [5]. Our algorithm is an intellectual descendent of Alsberg's et al. primary site algorithm [1, 2], the primary copy algorithm proposed for distributed Ingres [30], and of SDD-1's reliable network [20].

Two other approaches to replicated data are *quorum consensus* [9, 11, 17, 32] and *missing write* [13, 14]. In the simplest form of quorum consensus, a transaction reads x by accessing a majority of copies and reading the most up-to-date one; a transaction writes x by writing a majority of copies. In the missing writes algorithm, a transaction executes differently, depending on whether sites are up or down. While all sites are up, a transaction reads x by reading any copy and writes x by writing all copies; while any site is down, transactions use quorum consensus.

The main features of our algorithm are

(1) A transaction can operate on a data item so long as one or more copies are available. To tolerate k failures, the system only needs $k + 1$ copies. In particular, to tolerate single-site failures, the system only needs two copies. (By contrast, quorum consensus and missing writes need three copies to tolerate single-site failures.)

(2) A transaction can read a data item by accessing a single copy. If a user has a copy of x at his local site, he can read it without involving other sites.

(3) The algorithm provides an integrated mechanism for adding and removing data item copies from the database. In this paper we only use the mechanism for failure and recovery purposes, but it could also be used to reconfigure the database as users' needs change.

(4) The algorithm is biased in favor of routine, predictable transactions at the expense of ad hoc queries. With proper database design, our algorithm processes routine transactions as efficiently as the simple algorithm does. The algorithm assumes that site failures occur infrequently compared to the transaction processing rate. Major costs are only paid when a site fails or recovers.

There are also some weaknesses:

(1) The algorithm handles a limited class of failures, namely, *clean, detectable* site failures. The algorithm does not handle Byzantine failures, network failures, or network partitions. See Section 2 for a discussion of this point. (Quorum consensus and missing writes can handle more failures.)

(2) Our presentation is abstract. We describe basic concepts and mechanisms for replicated data; we do not explain how to build these mechanisms into a complete distributed database system. See [18] for one approach to this issue.

(3) We do not prove the correctness of our algorithm, although we sketch a correctness proof. See [5] for a complete proof of a similar algorithm.

This paper has five sections. Section 2 defines basic terms and assumptions, and explains the types of failures our algorithms can handle. Section 3 presents the algorithm. Section 4 sketches a correctness proof. Section 5 is the conclusion.

2. BACKGROUND

2.1 Database Model

Logically, a *database* is a set of *data items*, denoted by $x, y, z, \ldots$. The database system (DBS) processes *read* and *write* operations on data items. Operation read(x) returns the current value of x. Operation write(x) assigns a new value to x. Users interact with the DBS by running programs, called *transactions*, that issue reads and writes.

Physically, each data item x has one or more copies, denoted x_a, x_b, ... Each copy is stored at a *site*. When a user transaction issues a read(x) operation, the DBS reads some copy of x. When a user transaction issues a write(x) operation, the DBS writes one or more copies of x.

Data items are an abstraction. They do not correspond directly to real database objects, such as records and files. Data items are record-like in that they can be read and written by basic operations. They are file-like in that they are named objects visible to the user. To build our algorithm into a system, one must recast this abstraction in terms of database objects that exist in the system.

Users expect the DBS to behave as if it executes transactions one-at-a-time against the logical, one-copy database. Physically, though, the DBS executes many transactions at a time against the physical, multicopy database. The problem addressed in this paper is to ensure that every physical execution is equivalent to a logical execution.

2.2 Failure Assumptions

The components of a distributed DBS can fail in many ways. No algorithm can survive all possible failures. This section describes the failures our algorithm is designed to survive.

We assume that site failures are *clean*: when a site fails it simply stops running; when the site recovers, it "knows" that it failed and initiates a recovery procedure. We do not consider Byzantine failures [12, 24], in which a site continues to run but performs incorrect actions.

Clean failures and Byzantine failures are two ends of a spectrum. Most real failures lie between these extremes. When a fault occurs, the site runs incorrectly for some time until it detects the fault. By assuming that failures are clean, we are assuming that faults are detected before serious damage is done. This assumption is implicit in all centralized database recovery algorithms [19].

We assume that site failures are *detectable*: while a site is "down," other sites can detect this fact. Some networks provide this feature internally (e.g., the early ARPANET NCP protocol [31] and SNA's virtual circuit protocol [21]). Others do not (e.g., the current ARPANET TCP/IP protocol [31]). If the network does not provide this feature, the DBS must implement it through high-level time-outs—which are not completely satisfactory, but are workable in practice.

This assumption is controversial. Quorum consensus and missing writes algorithms do not need it. We feel it is justified for two reasons. First, algorithms that do not make this assumption pay dearly. Quorum consensus and missing writes need three copies of a data item to tolerate a single failure; we feel this is too expensive. Second, failure detection is a generally useful feature. There is compelling theoretical evidence [16] that every reliable, distributed algorithm needs this feature or something comparable (namely, high-level time-outs).

Our third assumption is that the network never becomes *partitioned*: if two sites are "up," they can always communicate. This assumption is appropriate for some networks, but not others. It is appropriate for most local area networks and for long-haul networks such as ARPANET that have multiple paths between sites and automatic routing. This assumption is not needed by quorum consensus and missing writes algorithms.

Finally, we assume that routine communication errors—lost, duplicate, and garbled messages—are handled by the network. If site a sends a message to site b, site b eventually receives the message, unless it fails first. We make no assumption about message ordering.

In summary, our algorithm is designed to survive clean, detectable site failures; it is not designed to handle Byzantine failures, network partitions, or network errors.

2.3 System Assumptions

Many important aspects of DBS reliability are beyond the scope of this paper.

We assume that every site runs a centralized DBS recovery algorithm [7, 19]. When a transaction commits, the site installs the transaction's updates into the permanent database. When a transaction aborts, the site undoes the transaction's updates. When a site recovers, the site undoes uncommitted updates and redoes committed updates, as necessary.

We assume that the distributed DBS runs a distributed atomic commit algorithm, such as two-phase commit [22, 27]. When a transaction commits (or, respectively, aborts), the system commits (or, respectively, aborts) the transaction at all sites where it was active.

Finally, we assume that the system runs a "total failure" algorithm [18, 28]. A *total failure* of data item x occurs when all sites storing copies of x fail. During a total failure of x, no one can operate on x because there are no "up" copies. As sites recover, the total failure algorithm determines which site (or, sites) failed last. When the last site to fail has recovered, transactions can resume using x, because that site holds the most up-to-date copy of x. There is an important case of total failure: if x is not replicated, a total failure of x occurs every time the site storing x fails.

Total failure is orthogonal to this paper. Total failure cannot cause executions to violate 1-serializability.

3. THE ALGORITHM

Sections 3.1 to 3.3 explain basic concepts and techniques used in the algorithm; Sections 3.4 and 3.5 describe the algorithm itself; Section 3.6 illustrates the algorithm based on the examples from the Introduction; and Section 3.7 discusses a way of tuning performance.

3.1 Basic Concepts

Associated with each data item x is a *directory* $d(x)$. Directories are replicated. We use $d_t(x)$, $d_u(x)$, ... to denote copies of $d(x)$; we leave out the "x," writing d_t, d_u, ... when no ambiguity is possible. Each directory copy $d_t(x)$ stores two kinds of information: a list of available copies of x, denoted $d_t\cdot$*data-items*; and a list of available copies of $d(x)$, denoted $d_t\cdot$*directories*.

The heart of our algorithm consists of special transactions, called *status transactions*, which make copies available and unavailable. These are

INCLUDE(x_a)—make x_a available
EXCLUDE(x_a)—make x_a unavailable
DIRECTORY-INCLUDE(d_t)—make d_t available

There is no DIRECTORY-EXCLUDE transaction; d_t becomes unavailable the instant its site fails.

The DBS invokes EXCLUDE transactions when a site fails, and INCLUDE and DIRECTORY-INCLUDE transactions when a site recovers. There is one exception: the DBS does not exclude copies of data items that have "totally failed"; see Section 2.3 and [18].

It is notationally convenient to treat copies at recovering sites as *new copies* that are joining the system for the first time. Thus, each copy has a well-defined lifetime. It is "born" (i.e., joins the system) at some point. Then it is included, thereby becoming available. Later it dies (i.e., its site fails). Then it is excluded. Once a copy is excluded, it remains unavailable forever.

Each transaction is supervised by a single site, called its *transaction manager*; cf. [4]. (Different transactions may, of course, have different transaction managers.) In most cases, if a transaction operates on x or $d(x)$, its transaction manager must have an available copy of $d(x)$. (Section 3.7 relaxes this requirement.) If the transaction manager for a transaction fails before the transaction reaches its "locked point" (see below), the DBS aborts the transaction.

3.2 Two-Phase Locking

Concurrency control is by two-phase locking (2PL). The algorithm sets some types of locks on data item copies and others on directory copies.

On data item copies, the algorithm sets *read-locks* and *write-locks*. These conflict in the usual way.

On directory copies, the algorithm sets *din-locks*, *in-locks*, *ex-locks*, and *user-locks*. These are set by DIRECTORY-INCLUDE, INCLUDE, EXCLUDE, and user transactions, respectively. As usual, locks on different copies do not conflict. Locks on the same copy conflict as shown below.

	din	in	ex	user
din-lock	x	x	x	—
in-lock	x	x	x	x
ex-lock	x	x	x	—
user-lock	—	x	—	—

— = compatible
x = conflict

It is a basic property of 2PL that, for every transaction, there is a period of time during which it owns all of its locks. The *locked point* of a transaction is an arbitrarily chosen point within this period [8].

We pay no attention to deadlock, which can be handled by well-known techniques.

3.3 Availability Testing

This section describes mechanisms for telling if copies stored at remote sites are available. There is one mechanism for directory copies and another for data item copies.

Suppose site b stores directory copy d_u, and suppose site a knows that d_u was available sometime in the past. Site a determines if d_u is still available as follows:

(1) If site b is "up," site a asks b whether d_u is available.
(2) If site b is "down," site a asserts that d_u is unavailable.

This is correct because directory copies become unavailable the instant their site fails, and remain unavailable thereafter (see Section 3.1). This mechanism depends critically on our assumption that site failures are detectable (see Section 2.2).

Suppose site b stores data item copy x_b, and suppose site a knows that x was available sometime in the past. Site a determines whether x_b is still available by consulting an available directory copy d_t. Copy x_b is still available if x_b is in $d_t \cdot$ data-items.

3.4 Status Transactions

The next two sections present the body of our algorithm. This section describes status transactions, the next section user transactions.

An INCLUDE transaction, INCLUDE(x_a), makes data item copy x_a available. This transaction has two jobs: it initializes x_a to the "current value" of x and it adds x_a to the data-item list of each available copy of $d(x)$.

A DIRECTORY-INCLUDE transaction, DIRECTORY-INCLUDE(d_t), makes directory copy d_t available. This transaction is similar to INCLUDE: it initializes d_t to the "current value" of $d(x)$, and it adds d_t to the directory list of each available copy of $d(x)$.

An EXCLUDE transaction, EXCLUDE(x_a), makes data item copy x_a unavailable. The transaction removes x_a from the data item list of each available copy of $d(x)$.

Each status transaction has one other job. If the transaction discovers that some directory copy d_u has become unavailable, the transaction removes d_u from the directory list of each available copy of $d(x)$.

Special forms of INCLUDE and DIRECTORY-INCLUDE exist to create the first copy of each logical data item. When a user decides to add a new logical data item x to the database, there are two steps. First, he creates an initial copy of $d(x)$ by invoking an *initial* DIRECTORY-INCLUDE transaction. Then, he creates the initial copy of x by invoking an *initial* INCLUDE transaction.

We present the special "initial" transactions first, followed by the normal DIRECTORY-INCLUDE, INCLUDE, and EXCLUDE transactions.

Initial DIRECTORY-INCLUDE

Input: d_t—the initial directory copy for x

1. Set a din–lock on d_t.
2. Write $d_t \cdot$data-items := { }, signifying that no copies of x are available, and $d_t \cdot$directories := $\{d_t\}$, signifying that d_t is the only available copy of $d(x)$. This write implicitly releases the din–lock on d_t.

Initial INCLUDE

Input: x_a—the initial copy of x
x-val—the initial value of x_a

The transaction manager for this transaction must have an available copy of $d(x)$. Let d_t be this copy.

1.1 Set an in-lock on d_t and read $d_t \cdot$directories. Let *dir* be the value read.
1.2 Set a write-lock on x_a.
2. For each d_v in dir–$\{d_t\}$, test whether d_v is still available and, if so, set an in-lock on d_v. Let *avbl* = $\{d_t\}$ union the set of copies locked in this step.
3.1 For each d_v in avbl, test whether d_v is still available and, if so, write $d_v \cdot$directories := avbl and $d_v \cdot$data-items := $\{x_a\}$. Each write implicitly releases the in-lock on d_v.
3.2 In parallel with step 3.1, write x_a := x-val and release the write-lock on x_a.

Normal DIRECTORY-INCLUDE

Input: d_u—a new directory copy for x

The transaction manager for this transaction must have an available copy of $d(x)$. Let d_t be this copy.

1. Set din-locks on d_t and d_u, then read $d_t \cdot$directories and $d_t \cdot$data-items. Let *dir* and *data* be the values read.
2. For each d_v in dir–$\{d_t\}$, test whether d_v is still available and, if so, set a din-lock on d_v. Let *avbl* = $\{d_t, d_u\}$ union the set of copies locked in this step.
3. For each d_v in avbl, test whether d_v is still available and, if so, write $d_v \cdot$directories := avbl. For d_u, also write $d_u \cdot$data-items := data. Each write implicitly releases the din-lock on d_v.

Normal INCLUDE

Input: x_b—a new copy of x

The transaction manager for this transaction must have an available copy of $d(x)$. Let d_t be this copy.

1.1 Set an in-lock on d_t and read $d_t \cdot$directories and $d_t \cdot$data-items. Let *dir* and *data* be the values read.
1.2 For some x_a in data, set a read-lock on x_a and read it. Let *x-val* be the value read.
1.3 Set a write-lock on x_b.
2. For each d_v in dir–$\{d_t\}$, test whether d_v is still available and, if so, set an in-lock on d_v.
Let *avbl* = $\{d_t\}$ union the set of copies locked in this step.
3.1 For each d_v in avbl, test whether d_v is still available and, if so, write $d_v \cdot$directories := avbl and $d_v \cdot$data-items := data union $\{x_b\}$. Each write implicitly releases the in-lock on d_v.
3.2 In parallel with step 3.1, write x_b := x-val and release the read-lock on x_a and the write-lock on x_b.

EXCLUDE

Input: x_b—an existing copy of x

The transaction manager for this transaction must have an available copy of $d(x)$. Let d_t be this copy.

1.1 Set an ex-lock on d_t and read d_t·directories and d_t·data-items. Let *dir* and *data* be the values read.
1.2 If data = $\{x_b\}$, then this is a total failure of x, so stop.
2. For each d_v in dir–$\{d_t\}$, test whether d_v is still available and, if so, set an ex-lock on d_v. Let $avbl = \{d_t\}$ union the set of copies locked in this step.
3. For each d_v in avbl, test whether d_v is still available and, if so, write d_v·directories := avbl and d_v·data-items := data minus $\{x_b\}$. Each write implicitly releases the in-lock on d_v.

3.5 User Transactions

We now explain how the algorithm processes user transactions. There are three parts: procedures to process read(x) and write(x) operations and a procedure executed when a user transaction reaches its locked point.

Let T_i be a user transaction. To operate on x, T_i's transaction manager must have an available copy of $d(x)$. Let d_t be this copy.

To read(x)

(1) Read d_t·data-items. Let *data* be the value read.
(2) Set a read-lock on some x_a in data, then read x_a.

To write(x)

(1) Set a user-lock on d_t.
(2) For each x_a in d_t·data-items, test whether x_a is still available and, if so, set a write-lock on x_a.

A copy x_a may be available when this step begins, but becomes unavailable during the step, because we allow EXCLUDEs to run concurrently with user transactions. (Note from Section 3.2 that ex-locks and user-locks do not conflict.) This is not an error.

The step ends when every x_a in d_t·data-items is write-locked or is unavailable. T_i can go on to its next operation before this step ends, leaving this step running in the background. T_i does not reach its locked point until this step ends.

(3) For each x_a locked in step 2, test whether x_a is still available and, if so, write it. If x_a has become unavailable, ignore it.

This step occurs independently for each x_a. For some, it may occur right after step 2 sets the lock, for others, much later, after T_i reaches its locked point.

When T_i reaches its locked point

1. For each x_a read by T_i, if x_a is not in d_t·data-items or, if EXCLUDE(x_a) has an ex-lock on d_t, then abort T_i.
   This test ensures that every x_a read by T_i is still available, and is not in the process of being made unavailable.
2. Do the following in parallel.
2.1 Release all user-locks and read-locks.
2.2 Finish step 3 of the write procedure and release all write-locks.

Two practical notes should be mentioned.

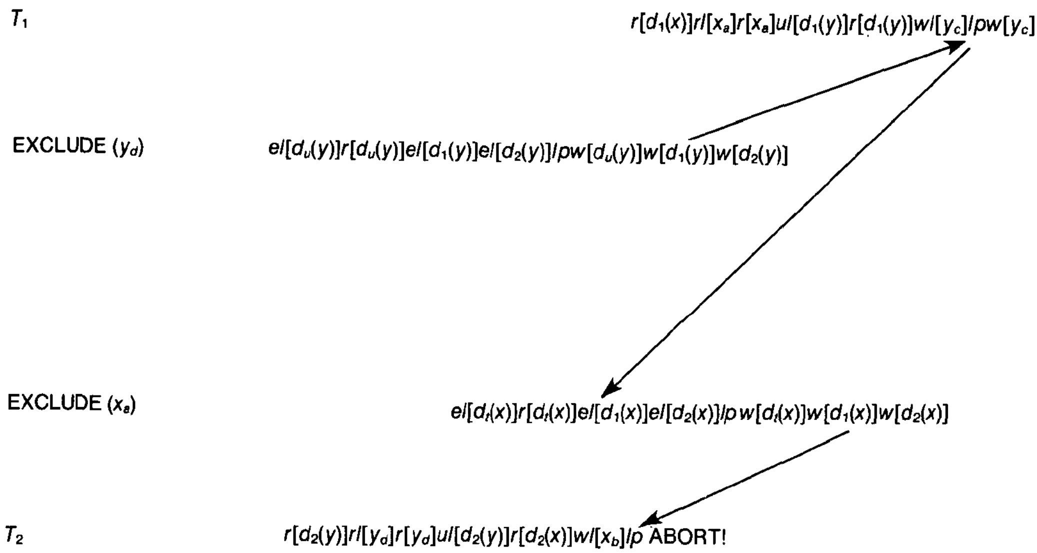

Fig. 1. The execution. Notation: r and w represent read and write operations. rl, wl, el, and ul represent the settings of read-locks, write-locks, ex-locks, and user-locks, respectively. l_p represents a transaction's locked point. We indicate off to the left which symbols go with which transactions. The order of execution is given by the left-to-right order of symbols on the page and by arrows.

(1) If EXCLUDE(x_a) locks d_t while T_i is running, the DBS can abort T_i immediately instead of waiting for T_i to reach its locked point.

(2) Phase 1 of two-phase commit can run before T_i reaches its locked point. Phase 2, however, must not run until step 1 of the locked-point procedure ends. Phase 2 of two-phase commit and step 2 of the locked-point procedure can use the same messages.

3.6 Examples

To see how the algorithm works, let us apply it to the examples from the Introduction. The first example illustrates EXCLUDE, the second illustrates INCLUDE.

Example 1 (revisited). See Figure 1. The database has logical data items x and y, with copies x_a, x_b, y_c, and y_d. There are two user transactions: T_1 reads x then writes y, T_2 reads y then writes x.

We shall need several directory copies: $d_1(x)$ and $d_1(y)$ at T_1's transaction manager; $d_2(x)$ and $d_2(y)$ at T_2's transaction manager; and $d_t(x)$ and $d_u(y)$ for use by EXCLUDEs.

All copies are available when the example begins.

Let us consider T_1 first. The DBS processes read(x) by reading $d_1(x)$, then locking and reading x_a. It processes write(y) by setting a user-lock on $d_1(y)$, reading $d_1(y)$, and trying to lock all copies listed on d_1·data-items, namely, y_c and y_d. As in the Introduction, we assume that y_d's site fails during this activity, and only y_c is locked. T_1 cannot reach its locked point until y_d becomes unavailable. This cannot happen until EXCLUDE(y_d) writes $d_1(y)$. So, let us examine the execution of EXCLUDE(y_d) next.

EXCLUDE(y_d) locks and reads $d_u(y)$ and then locks all copies listed in $d_u\cdot$directories, namely, $d_1(y)$ and $d_2(y)$. After reaching its locked point, EXCLUDE(y_d) writes $d_u(y)$, $d_1(y)$, and $d_2(y)$. As observed above, the write on $d_1(y)$ must precede T_1's locked point. There is an arrow in Figure 1 to reflect this. EXCLUDE(x_a) executes similarly.

Now, let us return to T_1. When T_1 reaches its locked point, the DBS tests whether x_a is still available. More precisely, it ensures that EXCLUDE(x_a) has not yet locked $d_1(x)$. There is an arrow in Figure 1 to reflect this.

Finally, let us see how T_2 executes. The DBS processes read(y) by reading $d_2(y)$, then locking and reading y_d. It processes write(x) by setting a user-lock on $d_2(x)$, reading $d_2(x)$, and trying to lock all copies listed in $d_2\cdot$data-items, namely, x_a and x_b. As in the Introduction, x_a's site fails during this activity, and only x_b is locked. T_2 cannot reach its locked point until x_a becomes unavailable. This cannot happen until EXCLUDE(x_a) writes $d_2(x)$. There is an arrow in Figure 1 to reflect this.

When T_2 reaches its locked point, the DBS tests whether EXCLUDE(y_d) has locked $d_2(y)$. Following paths in Figure 1, we find:

(1) EXCLUDE(y_d) locks $d_2(y)$ before writing $d_1(y)$;
(2) EXCLUDE(y_d) writes $d_1(y)$ before T_1 reaches its locked point;
(3) T_1 reaches its locked point before EXCLUDE(x_a) locks $d_1(x)$;
(4) EXCLUDE(x_a) locks $d_1(x)$ before writing $d_2(x)$; and
(5) EXCLUDE(x_a) writes $d_2(x)$ before T_2 reaches its locked point.

Therefore, EXCLUDE(y_d) locks $d_2(y)$ before T_2 reaches its locked point, and T_2 aborts. This is the outcome we want, because the example represents an incorrect execution.

Example 2 (revisited). The database has logical data items x and y, with copies x_a, x_b, x_c, and y_d. There are two user transactions: T_1 writes x then reads y, T_2 reads x then writes y.

We need the following directory copies: $d_1(x)$ and $d_1(y)$ at T_1's transaction manager; $d_2(x)$ and $d_2(y)$ at T_2's transaction manager; and $d_t(x)$ for use by INCLUDE(x_b).

All copies except x_b are available when the example begins. Copy x_b becomes available during the example.

The problem is to synchronize INCLUDE(x_b) with T_1. There are three possible correct outcomes:

(1) T_1 logically precedes INCLUDE(x_b). In this case, the INCLUDE initializes x_b to the value of x written by T_1.
(2) INCLUDE(x_b) logically precedes T_1. In this case, copy x_b is available when T_1 runs, and T_1 writes all three copies of x.
(3) Deadlock. One or the other transactions aborts.

The algorithm achieves this synchronization by locking. T_1 sets a user-lock on $d_1(x)$; INCLUDE(x_b) sets an in-lock on $d_1(x)$; these locks conflict. If T_1 sets its

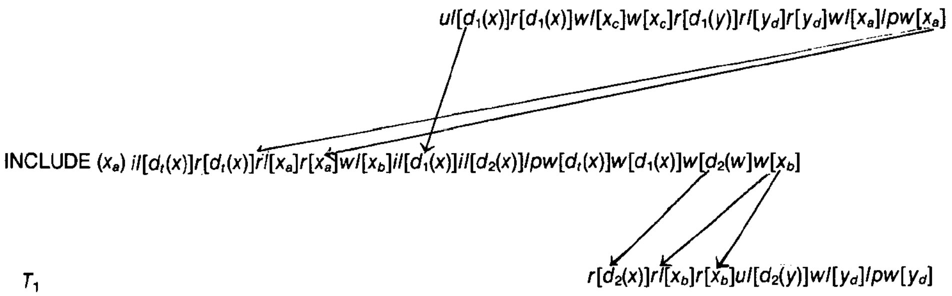

Fig. 2.

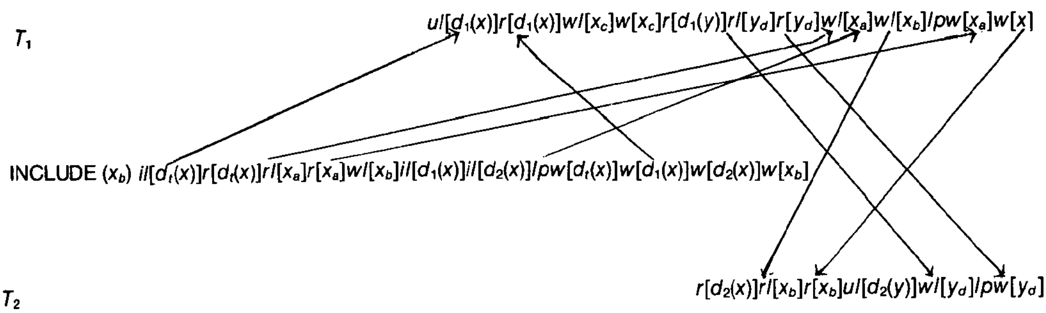

Fig. 3.

lock first, we get the first outcome (or deadlock). If the INCLUDE sets its lock first, we get the second outcome (or deadlock).

Figures 2 and 3 illustrate the first two outcomes. The only new notation is: *"il"* represents the setting of an in-lock.

In both cases, T_2 reads the value of x produced by T_1. Both outcomes are equivalent to a serial execution of T_1 followed by T_2 on a one-copy database.

3.7 Performance Tuning

For site a to serve as the transaction manager for T_i, a must store a directory copy for each data item used by T_i. The allocation of directory copies to sites is an important optimization problem.

With most reasonable allocations, cases may arise in which a user wants to run T_i at site a, even though a does not have copies of all needed directories. We can handle such cases in two ways.

(1) We can move the missing copies to a by running DIRECTORY-INCLUDE transactions. When the copies are no longer needed, we would like to exclude them. This requires that the system support explicit DIRECTORY-EXCLUDE transactions.

(2) We can let T_i use nonlocal copies of the missing directories. This requires a change to the algorithm: when T_i reaches its locked point, the algorithm must

test whether each nonlocal directory copy is still available, and abort T_i if any is not.

The second choice can be expensive. In some DBS's, a transaction reaches it locked point after the first phase of atomic commit. The availability testing must be done after the first phase ends, but before the second phase begins. In effect, this adds an extra phase to atomic commit.

Intuition suggests that the second approach may be worthwhile for ad hoc queries, as opposed to production transactions. In [5] we describe an algorithm that takes this idea to its extreme, treating all directory copies as nonlocal.

4. CORRECTNESS PROOF

This section sketches a correctness proof for our algorithm. The proof is based on serializability theory [4, 8, 23, 26, 29, 34]. Serializability theory analyzes an algorithm by analyzing the execution orders it allows. An algorithm is correct if all of its execution orders are correct. For a replicated data algorithm, "correct" means "1-serializable."

The proof has two main stages. The first derives "synchronization properties" enforced by the algorithms. The second proves that these synchronization properties imply correctness.

Section 4.1 defines terminology used in the proof; Section 4.2 sketches the first stage; and Section 4.3 sketches the second stage. Complete proofs of related algorithms appear in [5].

4.1 Terminology

An execution order is called a *log*. Logs are like the diagrams we use in Sections 1 and 3. The main differences are (1) logs only contain operations from committed transactions; this is acceptable because an aborted transaction has no visible effects. And, (2) logs contain INCLUDEs and EXCLUDEs for all data item copies used in the log, and DIRECTORY-INCLUDEs and DIRECTORY-EXCLUDEs for all directory copies used in the log.

When we draw logs as diagrams, we indicate the execution order by the left-to-right order of symbols and by arrows; in text, we use "<."

We use the following notation: "r_i" and "w_i" represent reads and writes executed on behalf of transaction T_i; "op_i" means "r_i" or "w_i"; "l_{pi}" represents T_i's locked point. We abbreviate INCLUDE by IN, EXCLUDE by EX, DIRECTORY-INCLUDE by DIN, and DIRECTORY-EXCLUDE by DEX.

Let T_i be a status transaction. Directory copy d_u is *potentially available* for T_i if d_u is listed in the directory copy that T_i reads. Let T_i be a user transaction. Data item copy x_a is *potentially available* for T_i if T_i operates on x, and x_a is listed in the copy of $d(x)$ that T_i reads.

4.2 Synchronization Properties

We analyze the algorithm by studying the logs that result from running the algorithm.

The first step is to derive *synchronization properties* that hold in every log produced by the algorithm. Bear in mind that logs only contain committed transactions; if a transaction violates the synchronization properties, it is aborted. Intuitively, the synchronization properties are

Conflict property—if transactions T_i and T_j execute conflicting operations, the transactions are synchronized so that one transaction "logically precedes" the other.

Write property—if a copy is potentially available for T_i, but T_i does not use the copy, then T_i "logically follows" the copy's exclusion.

Read property—if T_i reads a copy, then T_i "logically precedes" the copy's exclusion.

We now state the synchronization properties precisely, and explain briefly how the algorithm enforces them.

Conflict property. Let T_i and T_j both be status transactions, or let both be user transactions, or let one be a user transaction that writes x and the other an INCLUDE on x. If $op_i < op_j$ and these operations conflict, then $l_{pi} < l_{pj}$.

For the cases listed here, all conflicting operations are synchronized by locks. (This can be verified by inspecting the algorithm.) The conclusion ($l_{pi} < l_{pj}$) is a basic property of two-phase locking.

Write property 1. Let T_i be a status transaction. If d_u is potentially available for T_i, but T_i does not lock d_u, then $\text{DEX}(d_u) < l_{pi}$.

The locked point of a status transaction (except an initial DIN) occurs after step 2. Step 2 tries to lock every potentially available copy, and does not finish until it locks each copy or discovers that the copy is unavailable.

Write property 2. Let T_i be a user transaction that writes x. If x_a is potentially available for T_i, but T_i does not write-lock x_a, then the locked point of $\text{EX}(x_a)$ *is* $<l_{pi}$.

The locked point of T_i occurs after step 2 of the write(x) procedure. Step 2 does not finish until it locks each potentially available copy or discovers that the copy is unavailable.

Read property 1. Let T_i be a status or user transaction. If T_i reads d_t, then $l_{pi} < \text{DEX}(d_t)$.

Every directory copy that T_i reads is stored at its transaction manager. So $\text{DEX}(d_t)$ represents the failure of T_i's transaction manager. If this site fails before T_i reaches its locked point, the DBS aborts T_i (see Section 3.1). This is why we insist that d_t be stored at T_i's transaction manager.

Read property 2. Let T_i be a user transaction that reads x. If T_i reads x_a, then $l_{pi} <$ the locked point of $\text{EX}(x_a)$.

Let d_t be the copy of $d(x)$ at T_i's transaction manager. Step 1 of the locked-point procedure makes sure that $\text{EX}(x_a)$ has not yet locked d_t. $\text{EX}(x_a)$ cannot reach its locked point before this.

4.3 Stage Two

The remaining problem is to show that the synchronization properties, plus other facts apparent from the algorithm, imply 1-serializability.

This stage has three steps. Let L be a log produced by the algorithm. Step 1 finds a serial log L_s, equivalent to L. Step 2 studies the behavior of status transactions in L_s; this step shows that status transactions use directories in the intuitively correct manner. Step 3 uses the result of step 2 to prove that L_s is 1-serializable. It follows that L is 1-serializable since L is equivalent to L_s by step 1.

Step 1. *An Equivalent Serial Log.* Given a log L, we can obtain a serial log L_s by sorting L in "l_p" order. We must show that L_s is equivalent to L. Let $\ll$ denote the left-to-right order of transactions in L_s, and let $\ll=$ mean "$\ll$ or identical to."

Most pairs of conflicting operations in L are synchronized by 2PL. This ensures that the conflicting operations appear in the same order in L and L_s, and so cannot violate equivalence. The only conflicts not synchronized this way are directory reads by user transactions versus directory writes by status transactions. We argue that such conflicts appear in the same order in L and L_s anyway, or else do not matter.

Let T_i be a user transaction that reads x, let x_a be the copy T_i reads, and let d_t be the directory copy at T's transaction manager. The operations that matter *vis a vis* $r_i[d_t]$ are writes on d_t that change the status of x_a or d_t. These are writes on d_t by IN(x_a), DIN(d_t), and EX(x_a).

In L, IN(x_a) writes d_t before T_i reads d_t. IN(x_a) also writes x_a before T_i reads x_a. Although the operations on d_t are not locked, the operations on x_a are locked. These locks force $l_{p\mathrm{IN}(x_a)} < l_p$ in L, hence IN(x_a) $\ll T_i$ in L_s, as desired. In effect, the locks on x_a act as surrogates for locks on d_t.

In L, DIN(d_t) also writes d_t before T_i reads d_t. In order for T_i to run, d_t must be available, meaning that DIN(d_t) has written d_t, thereby releasing its lock on d_t. This ensures $l_{p\mathrm{DIN}(d_t)} \leq l_{pi}$ in L, hence DIN(d_t) $\ll T_i$ in L_s.

The final case is EX(x_a). In L, EX(x_a) writes d_t after T_i reads d_t. Read property 2 "catches" this conflict, by forcing $l_{pi} < l_{p\mathrm{EX}(x_a)}$ in L, hence $T_i \ll$ EX(x_a) in L_s.

Now let T_i be a user transaction that writes x. The operations that matter *vis a vis* $r_i[d_t]$ are writes on d_t by INCLUDEs on any copies of x and DIN(d_t). Writes by EXCLUDEs do not matter; it is always safe to write excluded copies since they cannot be read.

The conflict between T_i and INCLUDEs is synchronized by 2PL; this is what user-locks are for. The conflict between T_i and DIN(d_t) has already been discussed.

To summarize the argument: All conflicts that matter are synchronized by 2PL or some other mechanism. This means that these conflicts appear in the same order in L and L_s. Therefore L and L_s are equivalent.

Step 2. *Status Transactions.* Status transactions for different data items are disjoint. With no loss of generality, this step considers a single data item x.

Intuitively, copy x_a is available at a point in L_s if x_a was included before the point and excluded after it. Step 2 proves that status transactions use directories in this intuitive manner.

The main part of the argument shows that each status transaction reads a directory copy written by the last status transaction before it. This lemma is proved as follows. Suppose the lemma is false, and let T_j be the first transaction that violates it. That is,

(1) T_j is a status transaction,
(2) T_j reads d_u written by T_h, and
(3) there exists a status transaction between T_h and T_j that does not write d_u.

Let T_i be the first such transaction (i.e., the one immediately following T_h). The construction looks like this:

$$w_h[d_t]\ w_h[d_u]\ \underbrace{\big|\ r_i[d_t] \ldots l_{pi}\ \big|}_{\text{no write on } d_u \text{ here.}}\ r_j[d_u] \ldots l_{pj}$$

Since T_h writes d_u, d_u is potentially available for T_i. Therefore, by write property 1, one of the following must hold.

Case 1. T_i locks d_u. T_j reads d_u after T_i locks it, and so T_i must unlock it. The only way T_i can unlock d_u is to write it. But, by construction, T_i does not write d_u. Contradiction.

Case 2. $\mathrm{DEX}(d_u) < l_{pi}$. T_j reads d_t, and so read property 1 states that $l_{pj} < \mathrm{DEX}(d_t)$. Hence, $l_{pj} < \mathrm{DEX}(d_t) < l_{pi}$. But by construction, $l_{pi} < l_{pj}$. Again, a contradiction.

Since both cases lead to contradiction, the lemma is proved.

To prove the main result of step 2 we use this lemma in a simple induction.

Step 3. *L_s is* 1-*Serializable.* In a serial one-copy log, a transaction that reads x reads the value written by the last transaction before it that writes x. Step 3 proves that the same property holds in L_s, and so L_s is 1-serializable.

The main part of the argument shows that L_s satisfies the following property. Let T_i be a user transaction that writes x but does not write x_a, and let $\mathrm{IN}(x_a) \ll T_i$. If transaction T_j reads x_a, then $T_j \ll= T_i$. That is, if x_a is included before T_i, but T_i does not write x_a, then every transaction that reads x_a comes before T_i.

The proof goes as follows. Suppose the lemma is false, and let T_i and T_j violate it. That is,

(1) T_i is a user transaction that writes x, but not x_a,
(2) $\mathrm{IN}(x_a) \ll T_i$,
(3) T_j reads x_a, and
(4) $T_i \ll T_j$.

Let T_i read its copy of $d(x)$ from T_h.

By an argument similar to case 1 of step 2, we can show that no INCLUDE on x comes between T_h and T_i. Since $\mathrm{IN}(x_a) \ll= T_i$ by assumption (3), it follows that $\mathrm{IN}(x_a) \ll T_h$. We can also show that $T_j \ll \mathrm{EX}(x_a)$: if T_j is a status transaction, it holds by step 2; if T_j is a user transaction, it holds by read property 2. So, L_s looks like this:

$$\mathrm{IN}(x_a) \ll T_h \ll T_i \ll T_j \ll \mathrm{EX}(x_a).$$

By step 2, copy x_a is potentially available for T_i, since x_a is included before T_i and excluded after it. Therefore, by write property 2, one of the following must hold.

Case 1. T_i locks x_a. T_j reads x_a after T_i locks it, so T_i must unlock it. The only way T_i can unlock x_a is to write it. But, by construction, T_i does not write x_a. Contradiction.

Case 2. The locked point of $\mathrm{EX}(x_a) < l_{pi}$. T_j reads x_a, so read property 2 states that $l_{pj} <$ the locked point of $\mathrm{EX}(x_a)$. Hence, $l_{pj} <$ locked point of $\mathrm{EX}(x_a) < l_{pi}$. But, by construction, $l_{pi} < l_{pj}$. Again, a contradiction.

Since both cases lead to contradiction, the desired property of L_s is proved.

Because of this property, it is easy to transform L_s into an equivalent serial one-copy log. For each x, make all operations in L_s refer to a single copy, and discard all noninitial INCLUDEs.

For example, if L_s is

$$w_{\mathrm{IN}(x_a)}[x_a]r_1[x_a]w_1[x_a]r_{\mathrm{IN}(x_b)}[x_a]w_{\mathrm{IN}(x_b)}[x_b]r_2[x_b],$$

the transformation yields

$$w_{\mathrm{IN}(x_a)}[x_a]r_1[x_a]w_1[x_a]r_2[x_a].$$

Thus, L_s is 1-serializable, as claimed.

5. CONCLUSION

We have presented an algorithm for managing replicated databases in a system where sites can fail and recover. Our algorithm extends the simple algorithm given in the Introduction, while preserving its essential character: namely, to read x, a transaction reads a single copy of x; and to write x, a transaction writes all available copies. We have added enough mechanism to make this simple idea work.

We believe our algorithm is an attractive alternative to quorum consensus and missing writes algorithms for systems that can use it. To use our algorithm, a system must employ lower-level mechanisms to make site failures look clean and detectable, and must use a network that does not partition easily. One system that meets these conditions is Computer Corporation of America's Adaplex system [10].

We believe the algorithm will perform well when two further conditions are met.

First, sites must fail infrequently. When a site fails, the algorithm updates all directory copies for all data items stored at the site. When the site recovers, the algorithm updates those directories again and, also, updates all directory copies for all directories stored at site. This is a lot of work. In systems where sites fail too often, quorum consensus and missing write algorithms might be better choices.

Second, data access patterns must be predictable. If a transaction operates on x, its transaction manager must have a copy of the directory for x, or it must use the less efficient techniques discussed in Section 3.7. For routine "production" transactions, it is reasonable to assume that transaction managers will have the needed directories. For ad hoc queries, it is a less reasonable assumption. In systems with too many ad hoc queries, the algorithm in [5] might be better.

For systems that meet these conditions, we believe our algorithm is a practical solution to the replicated data problem.

ACKNOWLEDGMENTS

We thank Umesh Dayal, Vassos Hadzilacos, and Dale Skeen for helping to formulate the ideas in this paper. We thank Jim Gray and an anonymous referee for improving the presentation.

REFERENCES

1. ALSBERG, P.A., BELFORD, G.G., DAY, J.D., AND GRAPA, E. Multicopy resiliency techniques. In *Distributed Data Management*, J.B. Rothnie, P.A. Bernstein, and D.W. Shipman, Eds., IEEE, New York, 1978, 128–176.
2. ALSBERG, P.A., AND DAY, J.D. A principle for resilient sharing of distributed resources. In *Proceedings 2nd International Conference on Software Engineering* (Oct. 1976), 562–570.
3. ATTAR, R., BERNSTEIN, P.A., AND GOODMAN, N. Site initialization, recovery, and back-up in a distributed database system. In *Proceedings 6th Berkeley Workshop* (Feb. 1982), 185-202.
4. BERNSTEIN, P.A., AND GOODMAN, N. A sophisticate's introduction to distributed database concurrency control. In *Proceedings 8th VLDB* (Sept. 1982), 62–76.
5. BERNSTEIN, P.A., AND GOODMAN, N. Concurrency control and recovery for replicated distributed databases. TR-20-83, Center for Research in Computing Technology, Harvard Univ., July 1983.
6. BERNSTEIN, P.A., AND GOODMAN, N. Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst. 8*, 4 (Dec. 1983), 465-483.
7. BERNSTEIN, P.A., GOODMAN, N., AND HADZILACOS, V. Recovery algorithms for database systems. In *Proceedings 9th IFIP Congress* (Sept. 1983), 799–801.
8. BERNSTEIN, P.A., SHIPMAN, D., AND WONG, W.S. Formal aspects of serializability in database concurrency control. *IEEE Trans. Softw. Eng. SE-5*, 3 (May 1979).
9. BREITWIESER, H., AND LESZAK, M. A distributed transaction processing protocol based on majority consensus. In *Proceedings 1st ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Aug. 1982), 224–231.
10. CHAN, H., DAYAL, U., FOX, S., GOODMAN, N., RIES, D., AND SKEEN, D. Overview of an Ada compatible distributed data manager. In *Proceedings 1983 ACM SIGMOD Conference on Management of Data* (May 1983), 228–231.
11. DANIELS, D., AND SPECTOR, A.Z. An algorithm for replicated directories. In *Proceedings 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Aug. 1983), 104–113.
12. DOLEY, D. The Byzantine generals strike again. *J. Algorithms 3*, 1 (1982).
13. EAGER, D.L. Robust concurrency control in a distributed database. TR CSRG U135, Univ. Toronto, Oct. 1981.
14. EAGER, D.L., AND SEVCIK, K.C. Achieving robustness in distributed database systems. *ACM Trans. Database Syst. 8*, 3 (Sept. 1983), 354–381.
15. ESWARAN, K.P., GRAY, J.N., LORIE, R.A., AND TRAIGER, I.L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.
16. FISCHER, M.J., LYNCH, N.A., AND PATERSON, M.S. Impossibility of distributed consensus with one fault process. In *Proceedings 2nd ACM SIGACT–SIGMOD Symposium on Principles of Database Systems* (Mar. 1983).
17. GIFFORD, D.K. Weighted voting for replicated data. In *Proceedings 7th Symposium on Operating Systems Principles* (Dec. 1979), 150–159.
18. GOODMAN, N., SKEEN, D., CHAN, A., DAYAL, U., FOX, S., AND RIES, D. A recovery algorithm

for a distributed database system. In *Proceedings 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Mar. 1983).

19. GRAY, J.N. Notes on database operating systems. In *Operating Systems: An Advanced Course*, vol. 60, Springer-Verlag, 1978, 393–481.
20. HAMMER, M.M., AND SHIPMAN, D.W. Reliability mechanisms for SDD-1: A system for distributed databases. *ACM Trans. Database Syst. 5*, 4 (Dec. 1980), 431–466.
21. LINDSAY, B.G., HASS, L.M., MOHAN, C., WILMS, P.F., AND YOST, R.A. Computation and communication in R*: A distributed database manager. *ACM Trans. Comput. Syst. 2*, 1 (Feb. 1984), 24–38.
22. LINDSAY, B.G., SELINGER, P.G., GALTIERI, C., GRAY, J.N., LORIE, R.A., PRICE, T.G., PUTZULO, F., TRAIGER, I.L., AND WADE, B.W. Notes on distributed databases. In *Distributed Databases*, Drattan and Poole, Eds., Cambridge University Press, New York, 1980, 247–284.
23. PAPADIMITRIOU, C.H. Serializability of concurrent updates. *J. ACM 26*, 4 (Oct. 1979), 631–653.
24. PEASE, M., SHOSTAK, R., AND LAMPORT, L. Reaching agreement in the presence of faults. *J. ACM 27*, 2 (1980), 228–234.
25. REED, D.P. Implementing atomic actions. In *Proceedings 7th ACM Symposium on Operating Systems Principles* (Dec. 1979).
26. SILBERSCHATZ, A., AND KEDEM, Z. Consistency in hierarchical database systems. *J. ACM 27*, 1 (Jan. 1980), 72–80.
27. SKEEN, D. Nonblocking commit protocols. In *Proceedings 1982 ACM SIGMOD Conference on Management of Data*, 133–147.
28. SKEEN, D. Calculating the last process to fail. In *Proceedings 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Mar. 1983).
29. STEARNS, R.E., LEWIS, P.M., II, AND ROSENKRANTZ, D.J. Concurrency controls for database systems. In *Proceedings 17th Symposium on Foundations of Computer Science*. IEEE, New York, 1976, 19–32.
30. STONEBRAKER, M. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Trans. Softw. Eng. SE-5*, 3 (May 1979), 188–194.
31. TANNENBAUM, A.S. *Computer Networks*. Prentice-Hall, Englewood Cliffs, N.J., 1981.
32. THOMAS, R.H. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst. 4*, 2 (June 1979), 180–209.
33. TRAIGER, I.L., GRAY, J., GALTHIER, C.A., AND LINDSAY, B.G. Transactions and consistency in distributed database systems. *ACM Trans. Database Syst. 7*, 3 (Sept. 1982), 323–342.
34. YANNAKAKIS, M., PAPADIMITRIOU, C.H., AND KUNG, H.T. Locking policies: Safety and freedom from deadlock. In *Proceedings 20th IEEE Symposium on Foundations of Computer Science* (1979), 286–297.

Received October 1983; revised June 1984; accepted June 1984