THE FAILURE AND RECOVERY PROBLEM FOR REPLICATED DATABASES

Philip A. Bernstein
Nathan Goodman

Harvard University

ABSTRACT

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. By storing critical data at multiple sites, the system can operate even though some sites have failed. However, few distributed database systems support replicated data, because it is difficult to manage as sites fail and recover.

A replicated data algorithm has two parts. One is a discipline for reading and writing data item copies. The other is a concurrency control algorithm for synchronizing those operations. The read-write discipline ensures that if one transaction writes logical data item x, and another transaction reads or writes x, there is some physical manifestation of that logical conflict. The concurrency control algorithm synchronizes physical conflicts; it knows nothing about logical conflicts. In a correct replicated data algorithm, the physical manifestation of conflicts must be strong enough so that synchronizing physical conflicts is sufficient for correctness.

This paper presents a theory for proving the correctness of algorithms that manage replicated data. The theory is an extension of serializability theory. We apply it to three replicated data algorithms: Gifford's "quorum consensus" algorithm, Eager and Sevcik's "missing writes" algorithm, and Computer Corporation of America's "available copies" algorithm.

Research supported by the National Science Foundation, grant number MCS79-07762, and by the Office of Naval Research, grant number N00014-80-C-674.

Authors' addresses: P.A. Bernstein, Sequoia Systems, Inc., One Metropolitan Corp. Center, Marlborough, MA 01752; N. Goodman, Aiken Computation Lab., Harvard University, Cambridge, MA 02138.

I. INTRODUCTION

In a one-copy distributed database, each data item is stored at exactly one site of a distributed system. In a replicated database, some data items may be stored at multiple sites. In a replicated database, when a user updates data item x, the distributed database system (dbs) must apply the update to one or more copies of x. When a user reads x, the dbs must select an up-to-date copy of x to be read. The main motivation for replicated data is improved reliability [ABDG,HS]: by storing important data at multiple sites, the dbs can tolerate failures more gracefully.

The main correctness criteria for replicated databases are: *replica 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 dbs that achieves replica control and concurrency control has the same input/output behavior as a centralized, one-copy dbs that executes user requests one at a time [TGGL].

The simplest way to handle replicated data is the following. When a user updates x, the dbs applies the update to *all* copies of x stored at "up" sites. When a user wishes to read x, the dbs reads *any* copy of x at an "up" site. Concurrency control is by distributed two phase locking [BG,EGLT].

Unfortunately, this simple algorithm is incorrect. Consider a database with data items x and y and copies x_a, x_b, y_c, y_d. Transaction T_1 reads x and writes y; T_2 reads y and writes x. The following execution obeys the simple algorithm, yet is incorrect.

$$r_1[x_a] \quad \text{d-fails} \quad w_1[y_c]$$
$$r_2[y_d] \quad \text{a-fails} \quad w_2[x_b]$$

'$r_1[x_a]$' denotes a read of x_a by T_1; 'd-fails' denotes the failure of site d; and so forth. Time moves from left-to-right. T_1 and T_2 do their reads at approximately the same time; then sites a and d fail; then T_1 and T_2 do their writes. This execution obeys the algorithm, assuming all sites are initially up, because (i) each transaction reads an up copy; (ii) each transaction

writes *all* up copies of the data item it updates; and (iii) since T_1 and T_2 operate on disjoint copies, the execution is trivially two phase locked. Nonetheless, the execution is incorrect, because, in any serial execution of T_1 and T_2 against a one-copy database, one of the transactions would have read the other's output.

This paper presents a theory for analyzing the correctness of replicated data algorithms and applies the theory to three algorithms: quorum consensus [BL,Gi,Th], missing writes [ES], and available copies [GSCDFR,HS].

Our techniques are designed to handle *clean site failures* in which a site simply stops processing operations. The available copies algorithm makes the further assumption that site failures are *detectable*. (The theory and other algorithms do not need this assumption.) We take centralized dbs recovery as a given: when a site recovers we assume that it can undo or redo partially completed transactions as necessary; see [BGH,Gr,Ve]. We do not consider Byzantine failures [Do,LSP], network failures, or network partitions.

Sections 2 and 3 present the theory. Sections 4-6 apply the theory to replicated data algorithms. Section 7 is the conclusion.

2. SERIALIZABILITY THEORY FOR ONE-COPY DATABASES

The theory developed in this paper is an extension of *serializability theory* for database concurrency control algorithms [BSW,Pa,SLR]. This section reviews this theory. Section 3 generalizes the theory for replicated databases.

2.1 Logs

Serializability theory models executions by *logs*. A log identifies the reads and writes executed on behalf of each transaction, and tells the order in which those operations were executed.

A *transaction log* represents an allowable execution of a single transaction. Formally, it is a partially ordered set (poset) $T_i = (\Sigma_i, <_i)$ where Σ_i is the set of reads and writes issued by transaction i, and $<_i$ tells the order in which those operations execute.

Data items are represented by $\{x,y,z,\ldots\}$. $r_i[x]$ (resp. $w_i[x]$) denotes a read (resp. write) on x by T_i. To avoid ambiguity, we assume no transaction writes a data item more than once. We also assume that if T_i reads and writes x, then $r_i[x] <_i w_i[x]$. These assumptions do not limit our results in any substantive way.

Let $T = \{T_0,\ldots,T_n\}$ be a set of transaction logs. A *dbs log* (or simply a *log*) over T represents an execution of $T_0,\ldots,T_n$. Formally, a log over T is a poset $L = (\Sigma, <)$ where

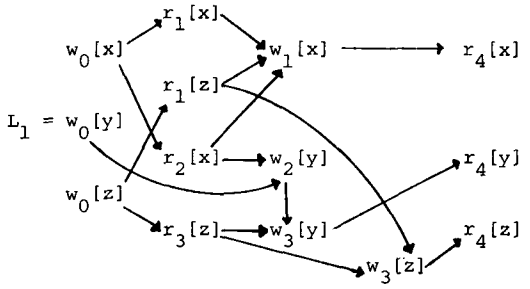
(1) $\Sigma = \bigcup_{i=0}^{n} \Sigma_i$; (2) $< \supseteq \bigcup_{i=0}^{n} <_i$; (3) every $r_j[x]$ follows at least one $w_i[x]$ ($r_j[x]$ *follows* $w_i[x]$ if $w_i[x] < r_j[x]$); and (4) all pairs of conflicting operations are $<$ related (two operations *conflict* if they operate on the same data item, and at least one is a write).

We draw logs as diagrams using arrows to depict $<$. Given transaction logs

$T_0 = w_0[x]\ w_0[y]\ w_0[z]$ $\qquad$ $T_1 = r_1[x] \rightarrow w_1[x]$, $r_1[z] \rightarrow w_1[x]$

$T_2 = r_2[x] \rightarrow w_2[y]$ $\qquad$ $T_3 = r_3[z] \rightarrow w_3[y]$, $r_3[z] \rightarrow w_3[z]$ $\qquad$ $T_4 = r_4[x]\ r_4[y]\ r_4[z]$

the following is a log over $\{T_0,T_1,T_2,T_3,T_4\}$.



Let L be a log over $\{T_0,\ldots,T_n\}$. Transaction T_j *reads-x-from* T_i in L if (1) $w_i[x]$ and $r_j[x]$ are in L; (2) $w_i[x] < r_j[x]$; and (3) no $w_k[x]$ falls between these operations. Two logs are *equivalent*, denoted $\equiv$, if they have the same reads-from's; i.e. for all i,j, and x, T_j reads-x-from T_i in one log iff this condition holds in the other.

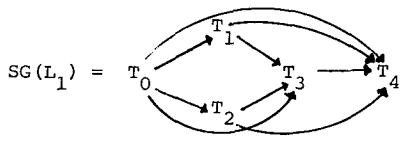
2.2 Serializable Logs

A *serial log* is a totally ordered log such that for every pair of transactions T_i and T_j, either all of T_i's operations precede all of T_j's, or vice versa. For example,

$$L_2 = w_0[x]w_0[y]w_0[z]r_2[x]w_2[y]r_1[x]r_1[z]w_1[x]r_3[z]w_3[y]w_3[z]r_4[x]r_4[y]r_4[z]$$

A log is *serializable* (*SR*) if it is equivalent to a serial log. E.g., log L_1 is SR because it is equivalent to L_2.

The serialization graph of log L, $SG(L)$, is a directed graph whose nodes represent transactions and whose arcs are $\{T_i \rightarrow T_j \mid \exists\ op_i$ in T_i and op_j in T_j such that op_i conflicts with op_j and $op_i < op_j\}$



THEOREM 1. [BSW,EGLT,Pa] *If* $SG(L)$ *is acyclic then* L *is* SR. □

All standard concurrency control algorithms ensure that $SG(L)$ is acyclic.

3. SERIALIZABILITY THEORY FOR REPLICATED DATABASES

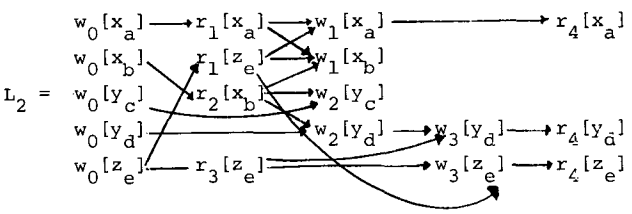
3.1 Replicated Data Logs

In a *replicated database*, each data item x has one or more *copies*, $x_a, x_b, \ldots$, at different sites. We sometimes use the terms *logical* data item and *physical* data items to emphasize the distinction between a data item x and its copies $x_a, x_b, \ldots$.

Let T be a set of transaction logs. To execute T in a replicated database, the dbs applies a *translation function*, t. This function maps each $r_i[x]$ into $r_i[x_a]$ for some copy x_a of x, and each $w_i[x]$ into $w_i[x_{a1}], \ldots, w_i[x_{a\ell}]$ for some copies $x_{a1}, \ldots, x_{a\ell}$ of x.

A *replicated dbs log* (or *rd log*) over T is a poset $< = (\Sigma, <)$ where (1) $\Sigma = t(\cup_{i=0}^{n} \Sigma_i)$, for some translation function t; (2) For each T_i, and all operations op_i and op_i', if $op_i <_i op_i'$ then every operation in $t(op_i)$ is $<$ related to every operation in $t(op_i')$; (3) Every $r_j[x_a]$ follows at least one $w_i[x_a]$; and (4) All pairs of conflicting operations are $<$ related (two operations *conflict* if they operate on the same *copy* and at least one is a write).

The following is an rd log over transactions $T_0, \ldots, T_4$ of Section 2.



In the sequel, we use L to be an arbitrary rd log over $T = \{T_0, \ldots, T_n\}$.

Transaction T_j *reads-x-from* T_i in L if for some copy x_a (1) $w_i[x_a]$ and $r_j[x_a]$ are operations in L; (2) $w_i[x_a] < r_j[x_a]$; and (3) no $w_k[x_a]$ falls between these operations. *Log equivalence*, *serial log*, *serialization graph*, and *SR log* are defined exactly as for one-copy logs. In addition, we extend the definition of log equivalence. Two rd or one-copy logs over T are *equivalent*, denoted $\equiv$, if they have the same reads-from's.

3.2 One-Copy Serializable Logs

An rd log is *one-copy serializable* (1-SR) if it is equivalent to a serial one-copy log. One-copy serializability is our correctness criterion for managing replicated data.

An SR rd log (or even a serial rd log) need not be 1-SR; e.g.,

$T_0 = w_0[x]$ $\quad T_1 = r_1[x]w_1[y]$ $\quad T_2 = r_2[y]w_2[x]$

x has copies x_a, x_b; y has copies y_c, y_d

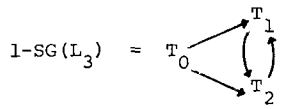
$L_3 = w_0[x_a]w_0[x_b]r_1[x_a]w_1[y_c]r_2[y_d]w_2[x_b]$

In any serial one-copy log over $\{T_0, T_1, T_2\}$, either T_1 or T_2 must read from the other. But in L_3, both T_1 and T_2 read from T_0. Thus L_3 is not 1-SR.

To tell if an rd log is 1-SR, we use a modified serialization graph. We need some graph terminology first. Given a graph G, $<<$ denotes its *precedence order*: $v_i << v_j$ if there is a path from v_i to v_j in G. Let V' be a subset of G's nodes. G *embodies* a total order over V' if for each $v_i, v_j \in V'$, either $v_i << v_j$ or $v_j << v_i$. We are mostly interested in acyclic graphs in which case if G embodies a total order over V', the total order is unique.

A *one-copy serialization graph* of log L, 1-SG(L), is SG(L) with enough edges added so that: (1) For each logical data item x, 1-SG(L) embodies a total order over the transactions that write x. This total order is called a *write order* for x and is denoted $<<_x$. (If $<<_x$ is not unique, arbitrarily pick one.) (2) For each x and transactions T_i, T_j, T_k (i,j,k distinct) such that T_j reads-x-from T_i and $T_i <<_x T_k$, 1-SG(L) contains a path from T_j to T_k. This is called a *reads-before* path; it signifies that T_j reads x "logically before" T_k writes x. In general, a log will have many 1-SG's.

One possible 1-SG for log L_3 is



The edges $T_0 \rightarrow T_1$ and $T_0 \rightarrow T_2$ are in SG(L_3) and also embody $<<_x$ and $<<_y$. The edge $T_1 \rightarrow T_2$ is a reads-before edge: it is needed since T_1 reads-x-from T_0 and $T_0 <<_x T_2$. $T_2 \rightarrow T_1$ is similar.

THEOREM 2. *If* L *has an acyclic* 1-SG, *then* L *is* 1-SR.

Proof. Let L_s be a serial one-copy log induced by a topological sort of 1-SG. We will show that $L_s \equiv L$, by showing they have the same reads-from's. Let T_j read-x-from T_i in L, and let T_k be any other transaction that writes x. If $T_k <<_x T_i$, then T_k precedes T_i in L_s. If $T_i <<_x T_k$, then 1-SG has a reads-beofre path from T_j to T_k and so T_k follows T_j in L_s. In neither case can T_k come between T_i and T_j, and so T_j reads-x-from T_i in L_s, as desired. □

Theorem 2 is our main tool for proving replicated data algorithms correct. We will characterize each algorithm by the logs it produces. Then we will show that every log produced by the algorithm has an acyclic 1-SG.

We conclude this section with a complexity result.

3.3 1-SR is NP-Complete for Serial Logs

Papadimitriou *et al.* have shown that it is NP-complete to decide if a one-copy log is SR [GJ, prob. SR33, Pa]. That result uses a slightly different notion of log equivalence than we use here, but it is straightforward to adapt the result

to our model. The analogous problem for an rd log is to decide if it is 1-SR. This problem is obviously NP-complete, because one-copy logs are a special case of rd logs. We prove a stronger result.

THEOREM 3. *It is NP-complete to decide if a serial rd log in 1-SR.*

Proof (membership in NP). Let L be an rd log over T. Guess a serial one-copy log L_s over T and verify $L_s \equiv L$.

(NP-hardness). The reduction is from the log SR problem.

A log L has an *acyclic reads-from* if the relation $\{T_i << T_j \mid$ for some x, T_j reads-x-from $T_i\}$ is acyclic. We can test this property in polynomial time; and if L does not have an acyclic reads-from, L is certainly not SR. So, it remains NP-complete to test if a log with acyclic reads-from is SR.

Let L' be a one-copy log with acyclic reads-from. Transform L' into an rd log L by translating each $w_i[x]$ into $w_i[x_i]$ and each $r_j[x]$ into $r_j[x_i]$ such that T_j reads-x-from T_i in L'. L and L' have the same reads-froms, hence $L' \equiv L$, and L has an acyclic reads-from. Let L_s be a serial log induced by a topological sort of the reads-from relation. $L_s \equiv L \equiv L'$, and so L_s is 1-SR iff L' is SR. □

4. QUORUM CONSENSUS ALGORITHM

4.1 How the Algorithm Works

For each data item x, define two collections of sets of copies of x, *read quorums* and *write quorums*, such that: (1) For each read quorum R and write quorum W, $R \cap W \neq \emptyset$. (2) For each pair of write quorums, W and W', $W \cap W' \neq \emptyset$. For example, the read and write quorums could be all sets containing a majority of copies.

The dbs processes write(x) by selecting a write quorum W and executing writes on all copies in W. To process read(x), the dbs uses a new operation, called *access*. The dbs processes read(x) by selecting a read quorum R, executing access operations on all copies in R, and then reading the most up-to-date copy accessed. (The next paragraph explains how the dbs can tell which copies are most up-to-date.) Access operations on a copy x_a conflict with writes on x_a, but don't conflict with reads. Access, read, and write operations are synchronized by a standard concurrency control algorithm.

Each copy has a *version number*, $VN(x_a)$. Version numbers are initially $\emptyset$. When the dbs processes write(x) on quorum W, it calculates $VN = \max\{VN(x_a) \mid x_a \in W\}$, and updates each version number to $1 + VN$. When the dbs processes read(x) on quorum R, each access returns its copy's version number, and the dbs reads the copy with largest version number.

4.2 Analysis Using Serializability Theory

To analyze the algorithm, we must formalize its behavior in terms of logs. A *quorum consensus (qc) log* is an rd log L such that

(i) If T_i writes x, then L contains $w_i[x_{a1}],\ldots,w_i[x_{a\ell}]$ for some write quorum $\{x_{a1},\ldots,x_{a\ell}\}$ of x;

(ii) If T_j reads x, then L contains

$$\begin{matrix} a_j[x_{a1}] \\ \vdots \\ a_j[x_{a\ell}] \end{matrix} \longrightarrow r_j[x_{ak}] \quad , \quad 1 \leq k \leq \ell$$

for some read quorum $\{x_{a1},\ldots,x_{a\ell}\}$ of x;

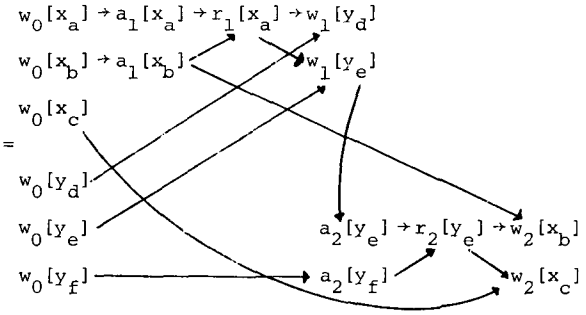
(iii) Every $r_j[x_a]$ or $a_j[x_a]$ follows at leas one $w_i[x_a]$ $(i \neq j)$; and

(iv) All pairs of conflicting operations are $<$ related, where writes on copy x_a conflict with writes, reads, and accesses on x_a.

For example, consider a database with data items x and y, with copies x_a, x_b, x_c, y_d, y_e, and y_f. Let the read and write quorums be all majority sets. Consider transactions

$T_0 = w_0[x]$ $T_1 = r_1[x]$ $w_1[y]$ $T_2 = r_2[y]$ $w_2[x]$ $w_0[y]$

A possible qc log over these transactions is



The serialization graph of a qc log embodies a write order, $<<_x$, for each x, because all write quorums intersect. Let T_i write x, and let $VN_x(T_i)$ be T_i's index in $<<_x$; $VN_x(T_i)$ is the version number that T_i assigns to each copy of x that it writes. The version number mechanism for reading is formalized by the following:

VN-Rule: Let R_x be the read quorum of x that T_j accesses. Let $last(T_j,x) = \{T_k \mid$ for some $x_a \in R_x$, $w_k[x_a]$ is the last write $x_a < a_j[x_a]\}$. T_j reads-x-from T_i such that $VN_x(T_i)$ is maximum over all $T_k \in last(T_j,x)$.

THEOREM 4. *Let* L *be a* qc *log that satisfies the* VN-*rule. If* SG(L) *is acyclic, then* L *is* 1-SR. *Thus, quorum consensus is a correct replicated data algorithm.*

Proof. We prove that SG(L) is a 1-SG. The result then follows by Theorem 2. We have already seen that SG(L) embodies a write order for each x. It remains to prove that it contains all needed reads-before paths: specifically, if T_j reads-x-from T_i and $T_i <<_x T_k$, we prove that SG(L) contains the edge $T_j \rightarrow T_k$.

By the VN-rule, $VN_x(T_i)$ is maximum over last(T_j,x), and, since $T_i <<_x T_k$, $VN_x(T_i) < VN_x(T_k)$. Thus, for all $T_h \in$ last(T_j,x), $T_h <<_x T_k$. Since SG(L) is acyclic, $<<_x$ is unique and is given by SG's precedence order. Thus, for all $T_h \in$ last(T_j,x), T_h precedes T_k in SG(L).

There exists a copy x_a that T_j accesses and T_k writes, because every read and write quorum intersect. Let T_h be the last transaction to write x_a before T_j accesses it. By the preceding paragraph, T_k writes x_a after T_h does. Thus L contains $w_h[x_a] < a_j[x_a] < w_k[x_a]$. The conflict between a_j and w_k creates the edge $T_j \rightarrow T_k$ in SG(L), as desired. □

5. MISSING WRITE ALGORITHM

5.1 How the Algorithm Works

Transactions run in two modes, *normal* and *failure*. If T_i runs in normal mode, the dbs processes write(x) by writing all copies of x and read(x) by reading any copy. If T_i runs in failure mode, the dbs processes it using a quorum consensus algorithm. Copies have version numbers. Transactions (in both modes) update version numbers as described in the previous section.

The choice of modes depends on whether T_i is "aware of missing writes." Informally, T_i is aware of a missing write on x if (i) T_i writes x, but does not write all copies of x; or (ii) the "last" T_k before T_i in SG(L) that writes x does not write all copies of x. (In case (ii), T_i need not read or write x itself.) We formalize this definition in the next subsection. If T_i is aware of a missing write, it must run in failure mode, else it may run in either mode. For this rule to be effective, the dbs must propagate missing write information along SG edges. See [ES] for details.

Transaction T_i may begin executing in normal mode and become aware of missing updates as it runs. When this happens, T_i can abort and re-execute in failure mode, or it can try to upgrade to failure mode on the fly. (In [ES], upgrades are done on the fly when T_i commits.)

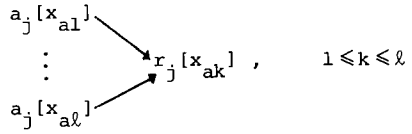
5.2 Analysis Using Serializability Theory

Transaction T_i is *maximal* for T_j w.r.t. x in log L, if T_i writes x, and there is a path from T_i to T_j in SG(L) such that no T_k ($k \neq i,j$) along the path also writes x. T_j is *aware of a missing write* if for some x, (i) T_j writes x but does not write all copies of x, or (ii) some T_i that is maximal for T_j w.r.t. x does not write all copies of x. (In case (ii), T_j need not operate on x.)

A *missing write (mw) log* is an rd log L such that

(i) If T_i writes x, then L contains $w_1[x_{a1}],\ldots,w_i[x_{a\ell}]$ for some write quorum $\{x_{a1},\ldots,x_{a\ell}\}$ of x;

(ii.1) If T_j reads x and is aware of a missing write, then L contains



for some read quorum $\{x_{a1},\ldots,x_{a\ell}\}$ of x. When L contains this structure, T_j obeys the VN-rule of Section 4.

(ii.2) If T_j reads x and is not aware of any missing writes, L may contain the above structure, or simply $r_i[x_a]$ for some copy x_a.

(iii) Every $r_j[x_a]$ or $a_j[x_a]$ follows at least one $w_i[x_a]$ ($i \neq j$).

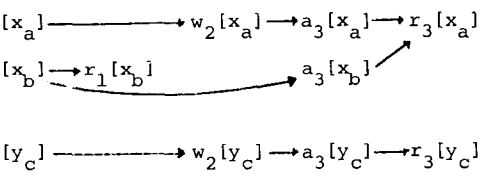
(iv) All pairs of conflicting operations are < related, where writes on copy x_a conflict with writes, reads, and accesses on x_a.

Here is an example of an mw log. The database has data items x and y, with copies x_a, x_b, and y_c. The quorums for x (both read and write) are $\{x_a\}$ and $\{x_a,x_b\}$. The quorum for y is, of course, $\{y_c\}$. The transactions are

$T_0 = w_0[x]\ w_0[y]$ $T_1 = r_1[x]$ $T_2 = w_2[x]\ w_2[y]$ $T_3 = r_3[x]\ r_3[y]$

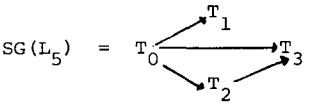
A possible mw log is

$L_5 =$



Note that T_1 runs in normal mode, but T_3 runs in failure mode. T_3 *must* run in failure mode because it is aware of T_2's missing write.

The serialization graph of an mw log embodies a write order for each x, because all write quorums intersect. However, the graph need not contain all necessary reads-before paths. For example,



T_1 reads-x-from T_0, T_2 writes x, and $T_0 <<_x T_2$, and so a 1-SG for L_5 must have reads-before path from T_1 to T_2. We can extend SG(L_5) into an acyclic 1-SG by adding the edge $T_1 \rightarrow T_2$. To prove the algorithm correct, we show that this extension works in general.

LEMMA 1. *Let* L *be an mv log. Let* T_j *read* x, T_k *write* x, *and suppose they are not connected in* SG(L). *Then* T_j *ran in normal mode (i.e., it read one copy of* x*), and* T_k *ran in failure mode and did not write all copies of* x.

Proof. If T_j accessed a quorum, it would conflict with every writer of x. If T_k wrote all copies, it would conflict with every reader of x. □

LEMMA 2. *Let* L *be an mv log with* SG(L) *acyclic. Let* T_j read-x-from T_i, *let* T_k *write* x $(k \neq i,j)$, *and let* $T_i <<_x T_k$. *Then* T_k *does not precede* T_j *in* SG(L).

Proof. There are two cases.

1. T_j read x using quorum consensus. The VN-rule forces T_j to precede T_k by an argument similar to that in Theorem 4.

2. T_j read one copy of x. Then T_i wrote all copies of x, and for T_k to precede T_j, T_k must precede T_i. This contradicts the assumption $T_i <<_x T_k$. □

THEOREM 5. *Let* L *be an mv log. If* SG(L) *is acyclic, then* L *is* 1-SR. *Thus, missing writes is a correct replicated data algorithm.*

Proof. Let $G(L) = SG(L) \cup \{T_j \rightarrow T_k \mid$ for some x and T_i, T_j reads-x-from T_i, and $T_i <<_x T_k\}$; i.e. G(L) is SG(L) augmented with all reads-before edges. G(L) is clearly a 1-SG. We prove that it is acyclic. The result then follows by Theorem 2.

We need two facts about directed graphs.

Fact 1. A cycle $v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_n \rightarrow v_0$ in a graph is *nonminimal* if the graph contains an edge, called a *chord*, $v_i \rightarrow v_{i+k}$ for $1 \leq i \leq n$, $k \geq 2$. That is, a nonminimal cycle can be shortened by substituting the chord $v_i \rightarrow v_{i+k}$ for the path $v_i \rightarrow v_{i+1} \rightarrow \ldots \rightarrow v_{i+k}$. All other cycles are *minimal*. Every cyclic graph has a minimal cycle.

Fact 2. Let v_j and v_k be nodes of an acyclic graph such that v_k does not precede v_j. Adding the edge $v_j \rightarrow v_k$ cannot create a cycle.

The main proof now begins.

Suppose G(L) is cyclic. By Fact 1, G(L) contains a minimal cycle C. By Fact 2 and Lemma 2, any cycle in G(L) must contain two edges not in SG(L); call those *new edges*. Let us write C in the form below, where $T_j \rightarrow T_k$ is a new edge, and $T_\ell \rightarrow T_m$ is the next new edge along the cycle.

$$C = T_j \xrightarrow{\text{new}} T_k \rightarrow \cdots \rightarrow T_\ell \xrightarrow{\text{new}} T_n \rightarrow \cdots \rightarrow T_j \quad .$$

By Lemma 1, T_k runs in failure mode and has a missing write, say, on x. Since C is minimal, no T_{ki} between T_k and T_ℓ writes x, else G(L) would have the chord $T_k \rightarrow T_{ki}$. The path from T_k to T_ℓ has no new edges, hence is in SG(L). No T_{ki} along this path writes x, hence T_k is maximal for T_ℓ w.r.t. x. Therefore T_ℓ is aware of T_k's missing write and runs in failure mode, too. But this contradicts Lemma 1 which claims that T_ℓ runs in normal mode because of the new edge $T_\ell \rightarrow T_m$. This contradiction proves that cycle C cannot exist, and G(L) is acyclic as desired. □

6. AVAILABLE COPIES ALGORITHM

6.1 How the Algorithm Works

The available copies algorithm is an embellished form of the incorrect, simple algorithm given in the Introduction. The idea is to ynchronize failures and recoveries with transactions, by controlling when a copy is deemed "available" for use.

When a site recovers from failure, the dbs runs a special *INCLUDE* transaction for each data item copy stored at the site. INCLUDE(x_a), or simply IN(x_a), brings the value of x_a up-to-date and declares x_a *available*. When a site fails, the dbs runs an *EXCLUDE* transaction for each copy stored there. EXCLUDE(x_a), or EX(x_a), declares x_a *unavailable*.

We make a notational assumption that simplifies the discussion: when a site recovers, we treat its data 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 becomes available through the action of an INCLUDE transaction. Later it dies--i.e. fails--and becomes unavailable through an EXCLUDE transaction. Once a copy is EXCLUDEd, it remains unavailable forever.

The dbs processes read(x) by reading any available copy of x. The dbs process write(x) by writing all available copies. The dbs synchronizes transactions with IN's and EX's to achieve four properties.

1. A transaction can read or write x_a only when it is available.

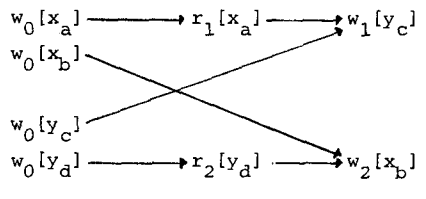
2. Suppose T_i reads x_a and x_a's site subsequently fails. Then T_i must "logically precede" EX(x_a).

3. If T_i writes x, condition (2) must hold for at least one copy T_i writes.

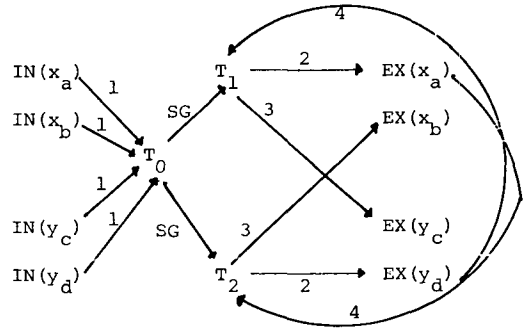
4. If T_i writes x and x_a's site fails before T_i is able to write x_a, then EX(x_a) logically precedes T_i. On the other hand, if a new copy x_b is INCLUDE'd while T_i is running and T_i does not write x_b, then T_i logically precedes IN(x_b).

Let us apply these properties to the example used in the Introduction. Log L_6 reproduces that example in log notation.

$L_6 =$



The diagram below indicates the logical precedence order required by the algorithm. The edge labels tell why each edge is present in the diagram: "1-4" means that the edge is needed to satisfy the corresponding property; "SG" stands for "serialization graph" and means that the edge represents the order of conflicting operations.



This diagram has a cycle: $T_1 \rightarrow EX(x_a) \rightarrow T_2 \rightarrow EX(y_d) \rightarrow T_1$. An available copies algorithm prevents bad logs like L_6 by ensuring that the "logical precedes" relation has no cycles.

6.2 Serializability Theory for Available Copies

This section develops the serializability theory needed to analyze the available copies algorithm. As in Section 3, we focus on the reads and writes executed by the dbs on behalf of *user transactions*, T_0, T_1,... . We treat IN and EX quite abstractly, specifying their behavior with a few properties that relate them to user transactions.

The theory addresses two main problems. One is to model the way INCLUDE's bring copies up-to-date. The other is to model the way IN's and EX's are synchronized with user transactions.

Modelling INCLUDE's

For each x, we designate one copy to be its *initial copy*. The initial copy is the first one to join the system. When this copy is INCLUDE'd, there is no need to bring it "up-to-date," because there is no value of x yet. The first user transaction to operate on the copy must create its initial value by writing it. This transaction must run before any other copies of x can be INCLUDED.

A subsequent INCLUDE, say $IN(x_a)$, must bring x_a up-to-date. It does this by (i) reading the value of x produced by some transaction T_i, and (ii) writing that value into x_a. (Later, we explain how to make sure the value read is up-to-date.) $IN(x_a)$ can read-from T_i by reading a copy of x that T_i wrote. [ABG], or by reading the value of x from a special place called a *spooler* [HS].

Suppose T_j reads-x-from $IN(x_a)$. The value read by T_j is precisely the value written by T_i, and we say that T_j *indirectly reads-x-from* T_i. All other reads-froms are *direct*. We extend our notion of log equivalence. Two logs over the same user transactions are *equivalent* if they have the same direct and indirect reads-from's. Finally, we extend the definition of 1-SG to required that if T_j indirectly reads-from T_i, there is a path from T_i to T_j. It is easy to verify that Theorem 2 remains valid under these extensions.

Suppose T_k writes x_a. We require that this write occurs after the write done by $IN(x_a)$. Therefore, if T_j reads-x-from $IN(x_a)$ and T_k writes x_a, then $r_j[x_a] < w_k[x_a]$.

Available Copies Serialization Graphs

Let L be an rd log over user transactions $T_0,\ldots,T_n$. An *available copies serialization graph (ACSG)* for L is a directed graph whose nodes represent $T_0,\ldots,T_n$, and $IN(x_a)$, $EX(x_a)$ for each copy x_a referenced in L. The edges among user transactions are:

SG Edges. $T_i \rightarrow T_j$ if for some x_a, $r_i[x_a] < w_j[x_a]$, or $w_i[x_a] < r_j[x_a]$, or $w_i[x_a] < w_j[x_a]$;

RF Edges. $T_i \rightarrow T_j$ if for some x, T_j indirectly reads-x-from T_i.

The edges relating user transactions to IN's and EX's are:

AC1. $IN(x_a) \rightarrow T_i$ if T_i reads or writes x_a;

AC2. $T_i \rightarrow EX(x_a)$ if T_i reads x_a;

AC3. $T_i \rightarrow EX(x_a)$ if T_i writes x, for at least *one* copy x_a that T_i writes.

AC4. $EX(x_b) \rightarrow T_i$ or $T_i \rightarrow IN(x_b)$ if T_i writes x, but does not write copy x_b.

To ensure that INCLUDE's read up-to-date values, we place one more constraint on ACSG's. Let $<<$ be the precedence order of an ACSG. Transaction T_i is *up-to-date* for $IN(x_a)$ if T_i writes x, $T_i << IN(x_a)$, and no T_k, $T_i << T_k << IN(x_a)$ also writes x. We require that if $IN(x_a)$ reads-from T_i, then T_i is *up-to-date* for $IN(x_a)$.

We now prove that if L has an acyclic ACSG, then L is 1-SR.

LEMMA 3. *An ACSG for* L *embodies a write order for each data item* x.

Proof. Let T_i and T_k write x. If they write the same copy, they are connected by an SG edge. So, assume they write disjoint sets of copies. Let x_a be a copy T_i writes in observance of AC3. By AC4, $T_k \rightarrow IN(x_a)$ or $EX(x_a) \rightarrow T_k$; by AC1, $IN(x_a) \rightarrow T_i$; by AC3, $T_i \rightarrow EX(x_a)$. Multiplying these conditions together, we get $T_k \rightarrow IN(x_a) \rightarrow T_i$ or $T_i \rightarrow EX(x_a) \rightarrow T_k$. In both cases, T_i and T_k are related. □

Consider an ACSG for L, and let $<<$ be its precedence order. The ACSG embodies a *reads-before order* for x, if for all distinct T_i, T_j, and T_k, if T_j reads-x-from T_i, T_k writes x, and $T_i << T_k$ then $T_j << T_k$.

LEMMA 4. *An acyclic ACSG for* L *embodies a reads-before order for each data item* x.

Proof. Let T_j read-x-from T_i, let T_k also write x, and let $T_i << T_k$. Let x_a be the copy T_j reads

There are four cases.

Case 1. T_i and T_k write x_a. Since $T_i \ll T_k$ and ACSG is acyclic, $w_i[x_a] < w_k[x_a]$. By definition of reads-from, this implies $r_j[x_a] < w_k[x_a]$, which introduces the SG edge $T_j \rightarrow T_k$.

Case 2. T_i writes x_a, but T_k does not. By AC4, $T_k \rightarrow IN(x_a)$ or $EX(x_a) \rightarrow T_k$; by AC1, $IN(x_a) \rightarrow T_i$; by AC2, $T_j \rightarrow EX(x_a)$. Multiplying these conditions, we get $T_k \rightarrow IN(x_a) \rightarrow T_i$ or $T_j \rightarrow EX(x_a) \rightarrow T_k$. The first term is impossible, since $T_i \ll T_k$ and ACSG is acyclic. We are left with the second term which implies $T_j \ll T_k$ as desired.

Case 3. T_i does not write x_a, but T_k does. In this case, T_j's reads-from is indirect, and $r_j[x_a] < w_k[x_a]$ by the observation made earlier in the section. As in Case 1, this gives $T_j \rightarrow T_k$.

Case 4. Neither T_i nor T_k writes x_a. Again, T_j's reads-from is indirect, and $IN(x_a)$ reads-from T_i. By AC4, $T_k \rightarrow IN(x_a)$ or $EX(x_a) \rightarrow T_k$. Since $T_i \ll T_k$, the first term implies that T_i is not up-to-date for $IN(x_a)$, contradicting the definition of ACSG. As in Case 2, the second term plus AC2 gives $T_j \rightarrow EX(x_a) \rightarrow T_k$. □

THEOREM 6. *If* L *has an acyclic ACSG, then* L *is* 1-SR. *Thus, available copies is a correct replicated data algorithm.*

Proof. Let $\ll$ be the ACSG's precedence order. By Lemmas 3 and 4, the restriction of $\ll$ to user transactions is an acyclic 1-SG. The result follows by Theorem 2. □

7. CONCLUSION

We have extended serializability theory to account for failures and recoveries in replicated databases. The main idea is *one-copy serializability*: an execution of transactions in a replicated database is one-copy serializable (1-SR) if it is equivalent to a serial execution of the same transactions in a non-replicated (one-copy) database. A replicated data algorithm is correct if all of its execution are 1-SR. We gave a graph structure, *one-copy serialization graphs (1-SG's)*, for testing if an execution is 1-SR. 1-SG's are analogous to the serialization graphs used in conventional serializability theory.

We applied the theory to three algorithms: quorum consensus, missing writes, and available copies. In the simplest form of quorum consensus, a transaction writes logical data item x by writing a majority of its copies; a transaction reads x by accessing a majority of copies and reading the most up-to-date one. In missing writes, a transaction runs in either of two modes. If the transaction is "aware" of missing writes, it uses a quorum consensus algorithm; else, to write x, it writes all copies of x, and to read x, it reads any copy. In available copies, a transaction writes x by writing all "available" copies of x; and reads x by reading any "available" copy. Copies become available and unavailable in a synchronized manner through special INCLUDE and EXCLUDE transactions.

Our analysis concentrates on the ordering of events imposed by each algorithm. We pay no attention to the mechanisms used to implement these orderings. As a result, our treatment is quite abstract and ignores many algorithmic issues. The gap between analysis and algorithm is modest for the first algorithm we treat (quorum consensus), but widens as the paper proceeds. We regard bridging this gap to be an important area for future research.

REFERENCES

[ABDG] Alsberg, P.A., G.G. Belford, J.D. Day, and E. Grapa, "Multi-copy Resiliency Techniques," *Distributed Data Management* (J.B. Rothnie, P.A. Bernstein, D.W. Shipman, eds.), IEEE, 1978, pp. 128-176.

[ABG] Attar, R., P.A. Bernstein, and N. Goodman, "Site Initialization, Recovery, and Back-up in a Distributed Database System," *Proc. 6th Berkeley Workshop*, Feb. 1982, pp. 185-202.

[BG] Bernstein, P.A., and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys* 13, 2 (June 1981), pp. 185-221.

[BGH] Bernstein, P.A., N. Goodman, and V. Hadzillacos, "Recovery Algorithms for Database Systems," *Proc. 9th IFIPS Congress*, Sept. 1983.

[BL] Breitwieser, H., and M. Leszak, "A Distributed Transaction Processing Protocol Based on Majority Consensus," *Proc. 1st ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, Aug. 1982, pp. 224-237.

[BSW] Bernstein, P.A., D.W. Shipman, and W.S. Wong, "Formal Aspects of Serializability in Database Concurrency Control," *IEEE Trans. on Software Engineering*, SE-5, 3 (may 1979), pp. 203-215.

[Do] Dolev, D., "The Byzantine Generals Strike Again," *J. of Algorithms*, 3, 1 (1982).

[EGLT] Eswaran, K.P., J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Commun. ACM*, 19, 11, Nov. 1976, pp. 624-633.

[Gi] Gifford, D.K., "Weighted Voting for Replicated Data," *Proc. 7th ACM-SIGOPS Symp. on Operating Systems Principles*, Dec. 1979, pp. 150-159.

[Gr] Gray, J.N., "Notes on Database Operating Systems," in *Operating Systems: An Advanced Course*, Springer-Verlag, 1979.

[GJ] Garey, M.J., and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Co., SF, 1979.

[GSCDFR] Goodman, N., D. Skeen, A. Chan, U. Dayal, S. Fox, and D. Ries, "A Recovery Algorithm for a Distributed Database System," *Proc. 2nd ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, March 1983.

[HS] Hammer, M.M., and D.W. Shipman, "Reliability Mechanisms for SDD-1: A System for Distributed Databases," *ACM Trans. on Database Syst.*, 5, 5 (Dec. 1980), pp. 431-446.

[Pa] Papadimitriou, C.H., "Serializability of Concurrent Updates," *JACM*, 26, 4 (October 1979), 631-653.

[PSL] Pease, M., R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *JACM*, 27, 2 (1980), pp. 228-234.

[SLR] Stearns, R.E., Lewis, P.M., II, and Rosenkrantz, D.J., "Concurrency Controls for Database Systems," *Proc. of the 17th Annual Symp. on Foundations of Computer Science*, IEEE, 1976, pp. 19-32.

[Th] Thomas, R.H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Trans. on Database Systems*, 4, 2 (June 1979), 180-209.

[TGGL] Traiger, I.L., J. Gray, C.A. Galtier, and B.G. Lindsay, "Transactions and Consistency in Distributed Database Systems," *ACM Trans. on Database Systems*, 7, 3, (Sept. 1982), pp. 323-342.

[Ve] Verhofstad, J.M.S., "Recovery Techniques for Database Systems," *ACM Computing Surveys*, 10, 2 (1978), pp. 167-196.