Serializability Theory for Replicated Databases*

PHILIP A. BERNSTEIN† AND NATHAN GOODMAN

Sequoia Systems, Inc., Marlborough, Massachusetts 01752

Received October 13, 1983; revised May 14, 1984

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 are stored at multiple sites. The main motivation for replicated data is improved reliability: by storing important data at multiple sites, the system can tolerate failures more gracefully. This paper presents a theory for proving the correctness of algorithms that manage replicated data. The theory is an extension of serializability theory. We use the theory to give simple correctness proofs for two replicated data algorithms: Gifford's "quorum consensus" algorithm, and Eager and Sevcik's "missing writes" algorithm.

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 [ABDG, HS]. 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 control and concurrency control has the same input/output behavior as a centralized, one-copy database system that executes user requests one at a time [TGGL]. Such behavior is termed one-copy serializability [ABG, BG3].

Many algorithms for managing replicated databases have appeared in the literature [ABDG, ABG, BG2, BL, DS, Ea, ES, Gi, GSCDFR, HS, St, Th]. However, few theoretical tools exist for proving the correctness of these algorithms.

This paper presents a theory for analyzing the correctness of replicated data algorithms, and uses the theory to analyze two replicated data algorithms: "quorum consensus" algorithms as in [BL, DS, Gi, Th], and "missing writes" algorithms in the style of [Ea, ES]. The theory is an extension of serializability theory [BG1, BSW, Pa, SK, SLR, YPK], which is traditionally used for analyzing concurrency control algorithms.

*Research supported by the National Science Foundation, Grant MCS79-07762, and by the Office of Naval Research, Grant N00014-80-C-674.
†Present address: Wang Institute, Tyng Road, Tyngsboro, Mass. 01879.
Section 2 reviews serializability theory for one-copy databases. Section 3 generalizes the theory to replicated databases. Section 4 applies the theory to quorum consensus algorithms; Section 5 considers missing writes algorithms.

The techniques described in this paper are designed to handle clean site failures in which a site simply stops processing operations. We do not consider Byzantine failures [Do, PSL], network failures, or network partitions.

Many important aspects of database system recovery are beyond the scope of this paper. We take centralized database recovery as a given: When a site recovers, it will undo or redo partially completed transactions as necessary. We also take atomic commitment as a given: When a transaction aborts, the system will undo the transaction's updates at all sites, and will abort all transactions that depend on those updates. We take a simplistic view of "copy recovery." When a copy of a data item \(x\) recovers from failure, we bring the copy up-to-date by writing the "current value" of \(x\) into the copy. This approach is acceptable from a theoretical standpoint, but hides important practical problems.

We also pay scant attention to termination issues. The algorithms we present are subject to deadlock and cyclic restart problems. These problems can be attacked by well-known techniques, and we do not treat them here.

2. SERIALIZABILITY THEORY FOR ONE-COPY DATABASES

We assume reader familiarity with serializability theory at the level of [BG1]. This section briefly reviews the main concepts.

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

Serializability theory models a concurrency control algorithm as a scheduler that constrains the order in which reads and write execute. The theory analyzes an algorithm by analyzing the execution orders, called logs, it allows. An algorithm is judged to be correct if all executions it allows are correct.

In our formulation of serializability theory, logs only contain operations from committed transactions. If a transaction aborts, its operations do not appear in the log. This is acceptable, because we take atomic commitment as a given (see the Introduction), and so an aborted transaction has no visible effects.

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 \(\Sigma_i\) is the set of reads and writes issued by transaction \(i\), and \(<_i\) tells the order in which those operations execute.

We use \(r_i[x]\) (resp. \(w_i[x]\)) to denote a read (resp. write) on \(x\) by \(T_i\). To avoid ambiguity, we assume no transaction reads or 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.
We draw logs as diagrams using arrows to depict $<$. Here are five transaction logs.

\[
\begin{align*}
T_0 &= w_0[y] \\
T_1 &= w_1[x] \\
T_2 &= r_2[x] \rightarrow w_2[y] \\
T_3 &= r_3[z] \\
T_4 &= r_4[y]
\end{align*}
\]

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

(i) $\Sigma = \bigcup_{i=0}^{n} \Sigma_i$;

(ii) $< \triangleq \bigcup_{i=0}^{n} <_i$;

(iii) 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

(iv) 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).

The following is a log over $\{ T_0, T_1, T_2, T_3, T_4 \}$, given above.
Let $L$ be a log over $\{T_0, \ldots, T_n\}$. Transaction $T_i$ reads-x-from $T_j$ in $L$ if (1) $w_i[x]$ and $r_j[x]$ are operations in $L$; (2) $w_i[x] < r_j[x]$; and (3) no $w_k[x]$ falls between these operations. An obvious, but important fact is that reads-from relationships are unique: if $T_i$ reads-x-from $T_j$ in $L$, then $T_j$ does not read $x$ from any other transaction in $L$.

Two logs over $\{T_0, \ldots, T_n\}$ are equivalent if they have the same reads-from's; i.e., for all $i, j$, and $x$, $T_i$ reads-x-from $T_j$ in one log iff this relationship holds in the other.

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,

$$L_1 = w_0[x] w_0[y] w_0[z] r_2[x] w_2[y] r_1[x] r_1[z] w_1[x] r_3[z]$$

$$w_3[y] w_3[z] r_4[x] r_4[y] r_4[z].$$

A log is serializable (SR) if it is equivalent to a serial log. For example, log $L_0$ is SR because it is equivalent to $L_1$.

Serializability is the correctness criterion for concurrency control in a one-copy database.

The physical serialization graph of log $L$, $PSG(L)$, is a directed graph whose nodes represent transactions and whose arcs are $(T_i \rightarrow T_j)$ if $op_i$ in $T_i$ conflicts with $op_j$ in $T_j$.

**Theorem 1.** [BSW, EGLT, Pa, SLR]. If $PSG(L)$ is acyclic then $L$ is SR.

All standard concurrency control algorithms ensure that $PSG(L)$ is acyclic. Standard algorithms include two phase locking (2PL) [EGLT], and timestamp ordering ($T/O$) [BSR, La, Re, Th].

3. **Serializability Theory for Replicated Databases**

3.1. Basic Concepts

In a replicated database, each data item $x$ has one or more copies, denoted $x_a, x_b, \ldots$, at different sites. Users interact with the system by running trans-
actions that issue reads and writes on data items. When a transaction issues read(x), the db is translating this into a read operation on a copy of x. When a transaction issues write(x), the db translates this into writes on one or more copies of x. Translations may contain operations other than reads and writes, too. But the semantics of a translation are carried fully by its reads and writes, and we deal only with these operations in this section.

An execution of a set of transactions is correct if it is equivalent to a serial execution of the transactions in which replication is transparent. Such an execution is termed one-copy-serializable (1-SR) [ABG, BG3].

3.2. Replicated Data Logs

Let $T$ be a set of transaction logs. To execute $T$ in a replicated database, the db 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 $w_i[x_{a1}], ..., w_i[x_{aL}]$ for some copies $x_{a1}, ..., x_{aL}$ of $x$.

A replicated db log (or rd log) over $T$ is a poset $< = (\Sigma, <)$, where

(i) $\Sigma = t(\cup_{i=0}^{n} \Sigma_i)$, for some translation function $t$;

(ii) for each $T_i$, and all operations $op_i$ and $op'_i$, if $op_i < op'_i$ then every operation in $t(op_i)$ is $<$ every operation in $t(op'_i)$;

(iii) every $r_j[x_a]$ follows at least one $w_j[x_a]$; and

(iv) 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 $T_0, ..., T_4$ of Section 2.

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

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.

We extend the notion of log equivalence given in Section 2. Two rd or one-copy logs are equivalent, denoted $\equiv$, if they have the same reads-from's.

Serial log, SR log, and physical serialization graph are defined as in Section 2.
3.3. 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. The following example illustrates this fact. The database consists of data items $x$ and $y$ with copies $x_a, x_b, y_c,$ and $y_d$. The transactions are

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

The log is

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

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.

3.4. Logical Serialization graphs

To ensure that an rd log is 1-SR, the dbs must ensure that each transaction reads-from the “correct” transaction—namely, the transaction it would have read from had there been only one copy. This notion is captured by a graph called a logical serialization graph (LSG), defined below.

Given an rd log $L$, let $G$ be a directed graph whose nodes represent the transactions in $L$. Let $\preceq$ denote $G$'s precedence relation, i.e., $T_i \preceq T_j$ if $T_i$ precedes $T_j$ in $G$.

$G$ induces a write order for $L$ if for all data items $x$, and transactions $T_i$ and $T_k$ ($i \neq k$) that write $x$, either $T_i \preceq T_k$ or $T_k \preceq T_i$. This definition just says that if two transactions write $x$, one transaction must precede the other. Equivalently, the transactions that write $x$ are totally ordered by $G$ (assuming $G$ is acyclic).

$G$ induces a read order for $L$ if for all $x$: (i) if $T_j$ reads-$x$-from $T_i$, then $T_i \preceq T_j$; and (ii) if $T_j$ reads-$x$-from $T_i$, $T_k$ writes $x$ ($i, j, k$ distinct), and $T_i \preceq T_k$, then $T_j \preceq T_k$. This definition says that $T_j$ follows the transaction, $T_i$, from which it reads $x$, and precedes all transactions, $T_k$, that subsequently write $x$.

$G$ is a logical serialization graph (LSG) for $L$ if it induces a write order and read order for $L$.

One possible LSG for $L_3$ is

```
\begin{center}
\begin{tikzpicture}
    \node (T0) at (0,0) {$T_0$};
    \node (T1) at (2,2) {$T_1$};
    \node (T2) at (2,-2) {$T_2$};
    \draw[-stealth] (T0) to[bend right] (T1);
    \draw[-stealth] (T0) to[bend left] (T2);
    \draw[-stealth] (T1) to[bend left] (T2);
\end{tikzpicture}
\end{center}
```
The edges $T_0 \rightarrow T_1$ and $T_0 \rightarrow T_2$ induce a write order for $L_3$ (wrt data items $y$ and $x$, respectively). Those edges also satisfy part (i) of the read order definition. The edge $T_1 \rightarrow T_2$ is needed for part (ii) of the read order definition, since $T_1$ reads-x-from $T_0$, $T_2$ writes x, and $T_0 \less T_2$. Edge $T_2 \rightarrow T_1$ is similar.

For a one-copy log $L$, every acyclic PSG is also an LSG. Let us see why this is so. If $T_j$ and $T_k$ write $x$, a one-copy log will contain the operations $w_j[x]$ and $w_k[x]$. These operations conflict, forcing $PSG(L)$ to have an edge connecting $T_j$ and $T_k$. Thus $T_j \less T_k$ or $T_k \less T_j$, and so $PSG(L)$ induces a write order. If $T_j$ reads-x-from $T_i$, then $L$ contains $w_j[x] < r_j[x]$. These operations conflict, forcing $PSG(L)$ to contain $T_i \rightarrow T_j$. If $T_k$ also writes $x$ and $T_i \less T_k$, then since the PSG is acyclic, $L$ contains $w_i[x] < r_j[x] < w_k[x]$. The conflict between $r_j[x]$ and $w_k[x]$ forces $PSG(L)$ to contain $T_i + T_k$, and so $PSG(L)$ induces a read order. Since $PSG(L)$ induces a write order and a read order, it is an LSG as claimed.

For an rd log, the PSG need not be an LSG. For example,

\[ PSG(L_3) = T_0 \]

\[ T_2 \]

This PSG does not induce a read order for $L_3$, hence is not an LSG.

**Theorem 2.** $L$ is 1-SR iff there exists an acyclic LSG for $L$.

**Proof** (if). Let $G$ be an acyclic LSG for $L$. Let $G_s$ be a topological sort of $G$ and let us use $G_s$ to construct a serial one-copy log $L_s$ in the obvious way: for each $T_i$, construct a serial transaction log by listing $T_i$’s (logical) operations in any order consistent with $<_i$; construct $L_s$ by concatenating the serial transaction log in $G_s$ order.

We prove $L_s \equiv L$ by proving they have the same reads-from’s. The proof has two steps.

**Step 1.** If $T_j$ reads-x-from $T_i$ in $L$, then this relationship holds in $L_s$.

**Reason.** Let $T_k$ be any other transaction that writes $x$. Since $G$ is an LSG, it induces a write order and a read order. The write order forces $T_i \less T_k$ or $T_k \less T_i$ in $G$. In the first case, the read order forces $T_j \less T_i \less T_k$. In the second case, the read order (together with the write order) forces $T_i \less T_j \less T_k$. In both cases, $T_i$ precedes $T_j$ in $L_s$, and $T_k$ does not come between $T_i$ and $T_j$. Therefore $T_j$ reads-x-from $T_i$ in $L_s$, as desired.

**Step 2.** If $T_j$ reads-x-from $T_i$ in $L_s$, then this relationship holds in $L$.

**Reason.** By definition of rd log, $T_j$ reads-x-from some transaction in $L$, say $T_k$. By Step 1, $T_j$ reads-x-from the same transaction in $L_s$. Since reads-from’s are unique, this transaction must be $T_i$. 

\[ T_1 \]

\[ PSG(L_3) = T_0 \]

\[ T_2 \]
Let $L_\alpha$ be a serial one-copy log equivalent to $L$. Let $G$'s edge-set be $\{ T_i \rightarrow T_j | T_i$ precedes $T_j$ in $L_\alpha \}$. $G$ is acyclic since all edges go "left-to-right" in $L_\alpha$. $G$ induces a write order, since it totally orders all transactions. $G$ induces a read order, because if $T_j$ reads-x-from $T_i$ then (i) $T_j$ follows $T_i$ in $L_\alpha$, and (ii) $T_j$ precedes all transactions after $T_i$ that also write $x$. 

Theorem 2 is our main tool for proving the correctness of replicated data algorithms. We use it in Sections 4 and 5 in proving the correctness of quorum consensus and missing writes algorithms.

3.5. A complexity Result

Papadimitriou has shown that it is NP-complete to decide if a one-copy log is SR [GJ, problem SR22; 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 is 1-SR.

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

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

A log $L$ has an *acyclic reads-from* if the relation $\{ T_i \ll T_j |$ 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 an acyclic reads-from is SR.

Let $L'$ be a one-copy log with an acyclic reads-from. Transform $L'$ into an rd log $L$ by translating each $w_j[x]$ into $w_i[x_j]$ and each $r_j[x]$ into $r_j[x_i]$ for each $T_j$ and $T_i$ 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 $L_\alpha$ be a serial log induced by a topological sort of the reads-from relation. Clearly, $L_\alpha \equiv L \equiv L'$, and so $L_\alpha$ is 1-SR iff $L'$ is SR.

4. QUORUM CONSENSUS ALGORITHM

4.1. How the Algorithm Works

For each data item $x$, let us define two collections of sets of copies of $x$, *read quorums* and *write quorums*, satisfying two properties:

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, if $x$ has five copies, the read quorums could be all sets containing two or more copies, and the write quorums could be all sets containing four or
more copies. Alternatively, the read and write quorums could be all sets containing a majority of copies.

The db system processes write \( x \) by selecting a write quorum \( W \) and executing writes on all copies in \( W \). To process read \( x \), the db system uses a new operation called access. The db system 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 db system can tell which copies are most up to date.) Access operations on a copy \( x_a \) conflict with writes on \( x_a \), but do not conflict with reads. Access, read, and write operations are synchronized by any concurrency control algorithm that produces an acyclic PSG (e.g., 2PL or T/O).

Each copy has a version number, initially 0. When the db system processes write \( x \) on quorum \( W \), it calculates \( VN = \) the maximum version number over all \( x_a \in W \), and updates each version number to \( 1 + VN \). When the db system processes read \( x \) on quorum \( R \), each access returns its copy's version number, and the db system reads the copy with largest version number.

### 4.2. Quorum Consensus Logs

To analyze the algorithm, we formalize its behavior in terms of logs. This log behavior constitutes our formal definition of the algorithm. A quorum consensus (qc) log is an rd log \( L \) such that:

(i) If \( T_i \) writes \( x \), then \( L \) contains \( w_i[x_{a_1}], \ldots, w_i[x_{a_d}] \) for some write quorum \( W = \{ w_{a_1}, \ldots, x_{a_d} \} \) of \( x \). Let \( \text{last}_i(x) = \{ T_h \} \) for some \( x_a \in W \), \( w_h[x_a] \) is the last write on \( x_a \) before \( w_i[x_a] \). Define \( VN_i(x) = 1 + \max (VN_i(x) \mid T_h \in \text{last}_i(x)) \), where \( \max (\emptyset) = 0 \). Intuitively, \( VN_i(x) \) is the version number \( T_i \) assigns to the copies of \( x \) that it writes.

(ii) If \( T_j \) reads \( x \), then \( L \) contains \( a_j[x_{a_1}], \ldots, a_j[x_{a_d}] \) for some read quorum \( R = \{ x_{a_1}, \ldots, x_{a_d} \} \), and \( r_j[x_{a_k}] \) for some \( x_{a_k} \in R \). For each \( x_a \in R \), let \( VN(x_a) = VN_i(x_a) \), where \( w_i[x_a] \) is the last write on \( x_a \) before \( a_j[x_{a_k}] \). The copy, \( x_{a_k} \), read by \( T_j \) must satisfy \( VN(x_{a_k}) = \max (VN(x_{a_k}) \mid x_a \in R) \).

(iii) Every \( r_j[x_{a_k}] \) or \( a_j[x_{a_k}] \) follows at least one \( w_i[x_{a_k}] \) \((i \neq j)\).

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

(v) PSG\( (L) \) is acyclic.

For example, consider a database with data items \( x \) and \( y \), with copies \( x_{a_1}, x_{a_2}, x_{a_3}, y_{a_1}, y_{a_2}, \) and \( y_{a_3} \). Let the read and write quorums be all majority sets. Consider transactions

\[
T_0 = w_0[x], \quad T_1 = r_1[x]w_1[y], \quad T_2 = r_2[y]w_2[x].
\]
A possible qc log over these transactions is

\[ L_4 = \]

\[
\begin{align*}
& w_0 \{x_a\} \rightarrow a_1 \{x_a\} \rightarrow r_1 \{x_a\} \rightarrow w_1 \{y_3\} \\
& w_0 \{x_b\} \rightarrow a_1 \{x_b\} \rightarrow w_1 \{y_c\} \\
& w_0 \{y_a\} \rightarrow a_2 \{y_c\} \rightarrow r_2 \{y_e\} \rightarrow w_2 \{x_b\} \\
& w_0 \{y_c\} \rightarrow a_2 \{y_c\} \rightarrow w_2 \{x_c\} \\
& w_0 \{y_f\} \rightarrow a_2 \{y_f\} \rightarrow w_2 \{x_f\}
\end{align*}
\]

In this example, last_0(x) = last_0(y) = \( \emptyset \), hence VN_0(x) = VN_0(y) = 1; last_1(y) = last_2(z) = T_0, hence VN_1(y) = VN_2(x) = 2. The copies of x that T_1 accesses were both written by T_0, hence have identical VN's and T_1 may read either. The copies of y that T_2 accesses were written by different transactions, hence have different VN's; T_2 reads the copy with larger VN as required.

4.3. Correctness Proof

We prove that quorum consensus is a correct replicated data algorithm by proving that every qc log is 1-SR. We do so by proving that the PSG of a qc log is also an LSG. Then, since the PSG is acyclic (by point (v) of the qc log definition), the log is 1-SR by Theorem 2.

**Lemma 4.1.** Let L be a qc log. Then PSG(L) induces a write order for L.

**Proof.** Let \( T_i \) and \( T_k \) write x. Since all write quorums for a given data item intersect, there exists a copy \( x_a \) that \( T_i \) and \( T_k \) both write. These writes on \( x_a \) conflict, forcing PSG(L) to have an edge connecting \( T_i \) and \( T_k \). Thus \( T_i \ll T_k \) or \( T_k \ll T_i \), and so PSG(L) induces a write order.

**Lemma 4.2.** Let L be a qc log. Then PSG(L) induces a read order for L.

**Proof.** Let \( T_j \) read-x-from \( T_i \). Then, for some copy \( x_a \), L contains \( w_i[x_a] < r_j[x_a] \), forcing PSG(L) to contain \( T_i \rightarrow T_j \).

It remains to prove that if \( T_k \) also writes x and \( T_i \ll T_k \), then \( T_j \ll T_k \). Since every read quorum for x intersects every write quorum for x, there exists a copy \( x_b \) that \( T_j \) accesses and \( T_k \) writes. Also, as noted in the previous lemma, there exists a copy \( x_c \).
that $T_i$ and $T_k$ both write; since $T_i \ll T_k$ and $PSG(L)$ is acyclic, $w_i[x_c] < w_k[x_c]$. The picture below summarizes the situation.

\[ w_i[x_a] \rightarrow r_j[x_a] \]

\[ a_j[x_b] \rightarrow w_k[x_b] \]

\[ w_i[x_c] \rightarrow w_k[x_c] \]

The proof proceeds in three steps.

**Step 1.** $VN_i(x) < VN_k(x)$. **Reason.** Let $w_i[x_c]$ be the next write on $x_c$ after $w_i[x_c]$. $T_i \in \text{last}_p(x)$, hence $VN_p(x) \geq 1 + VN_i(x) > VN_i(x)$. Applying this argument inductively proves the claim.

**Step 2.** Let $T_h$ be any transaction such that $w_h[x_b] < a_j[x_b]$. Then $VN_h(x) \leq VN_j(x)$.

**Reason.** Let $w_h[x_b]$ be the last write on $x_b$ before $a_j[x_b]$ (possibly $h = hn$). By Step 1, $VN_h(x) \leq VN_{hn}(x)$, while by point (ii) of the qc log definition $VN_{hn}(x) \leq VN_j(x)$.

**Step 3.** $T_j \ll T_k$.

**Reason.** By Step 1, $VN_i(x) < VN_k(x)$. By Step 2, if $w_i[x_b] < a_j[x_b]$, then $VN_k(x) \leq VN_j(x)$. But this implies $T_j$ reads $x$ from $T_k$, a contradiction. Consequently, $a_j[x_b] < w_k[x_b]$, forcing PSG(L) to contain $T_j \rightarrow T_k$.

This completes the proof that $PSG(L)$ induces a read order. 

The main result of the section follows by Theorem 2.

**Theorem 4.** Every qc log is 1-SR. Thus quorum consensus is a correct replicated data algorithm.

5. **Missing Writes Algorithms**

5.1. **How the Algorithm Works**

In the missing writes algorithm, a transaction can run in either of two modes: normal mode or failure mode. If a transaction runs in normal mode, the dbs processes write($x$) by writing all copies of $x$ and read($x$) by reading any copy. If a transaction runs in failure mode, the dbs processes it by using quorum consensus.

The choice of mode depends on whether the transaction is “aware of any missing writes.” Intuitively, transaction $T_j$ is aware of a missing write on $x_a$ if (i) $T_j$ writes $x$, but does not write $x_a$; or (ii) some transaction that “immediately precedes” $T_j$ is aware of a missing write on $x_a$. (In case (ii), $T_j$ need not read or write $x$ itself.) If $T_j$ is aware of any missing writes, it must run in failure mode; else it may run in either mode.
It is possible for $T_j$ to begin running in normal mode and become aware of a missing write as it runs. When this happens, the dbs can abort $T_j$ and re-execute it in failure mode, or it can try to upgrade to failure mode: For each $x$ that $T_j$ read, the dbs must access a read quorum $R$ and check that the value read by $T_j$ is at least as up-to-date as all copies in $R$. In [ES], upgrades are deferred until $T_j$ completes this execution and tries to commit. We ignore upgrades in our analysis, since the effect is identical to running the transaction in failure mode from the start.

We now formalize the definition of missing writes.

Transaction $T_i$ immediately precedes $T_j$ if for some $x_a$, (i) $r_i[x_a] < w_j[x_a]$, or (ii) $w_i[x_a] < r_j[x_a]$, or (iii) $w_i[x_a] < w_j[x_a]$, and no $w_k[x_a]$ comes between these operations.

The set of missing writes for $T_j$, denoted $MW(j)$, is defined recursively:

$$MW(j) = \bigcup_{T_i \text{ that immediately precede } T_j} MW'(i) \cup \{x_a | T_j \text{ writes } x \text{ but not } x_a\}$$

$$MW'(j) = MW(j) - \{x_a | T_j \text{ writes } x_a\}.$$

That is, $x_a$ is in $MW(j)$ if (i) $T_j$ writes $x$ but does not write $x_a$; or (ii) $x_a$ is in $MW(i)$ for some $T_i$ that immediately precedes $T_j$, and $T_i$ does not write $x_a$.

If $MW(j)$ is nonempty, we say that $T_j$ is aware of missing writes and must run in failure mode. If $MW(j)$ is empty, $T_j$ is not aware of missing writes and may run in either mode.

For this definition to be effective, the dbs must store $MW$ for each transaction, and propagate this information from one transaction to the next.

If concurrency control is by two phase locking, there us a simple, brute force way of doing this. The dbs maintains two “missing writes sets,” $R\cdot MW(x_b)$ and $W\cdot MW(x_b)$, for each data item copy $x_b$. When transaction $T_j$ begins executing, the dbs initializes $MW(g) = \emptyset$. When $T_j$ gets a read-lock on $x_b$, the dbs adds $R\cdot MW(x_b)$ to $MW(j)$. When $T_j$ gets a write-lock on $x_b$, the dbs adds $W\cdot MW(x_b)$ to $MW(j)$. When $T_j$ reaches its locked point—i.e., when it has obtained all of its locks—the dbs calculates $MW'(j) = MW(j) - \{x_b | T_j \text{ has a write-lock on } x_b\}$. Before releasing its read-lock on $x_b$, the dbs sets $W\cdot MW(x_b) = W\cdot MW(x_b) \cup MW'(j)$. Before releasing its write-lock on $x_b$, the dbs sets $R\cdot MW(x_b) = MW'(j)$ and $W\cdot MW(x_b) = MW'(j)$. It is easy to verify that this mechanism computes the correct value of $MW(j)$.

It is possible to devise similar mechanisms for other concurrency control algorithms. See [Ea, ES] for details.

5.2. Missing Writes Logs

To analyze the algorithm, we formalize its behavior in terms of logs.

We partition the set of transactions into two classes, called normal and failure. If $T_i$ is aware of any missing writes, i.e., if $MW(i) \neq \emptyset$, $T_i$ belongs to the failure class; else $T_i$ may belong to either class. Hereafter, we use the phrase “runs in normal (resp. failure) mode” instead of “belongs to the normal (resp., failure) class.”
A missing writes (mw) log is an rd log $L$ such that

(i) Let $T_i$ write $x$.

(i.1) If $T_i$ runs in normal mode, $L$ contains $w_i[x_{a_1}],..., w_i[x_{a_l}]$, where $W = \{x_{a_1},..., x_{a_l}\}$ contains all copies of $x$.

(i.2) If $T_i$ runs in failure mode, $L$ contains $w_i[x_{a_1}],..., w_i[x_{a_l}]$, where $W = \{x_{a_1},..., x_{a_l}\}$ is a write quorum of $x$;

(i.3) In either case, let $\text{last}_i(x) = \{T_h | \text{for some } x_a \in W, w_h[x_a] \text{ is the last write on } x_a \text{ before } w_i[x_a]\}$. Define $\text{VN}_i(x) = 1 + \text{max}(\text{VN}_a(x) | T_a \in \text{last}_i(x))$, where $\text{max}(\emptyset) = 0$.

(ii) Let $T_j$ read $x$.

(ii.1) If $T_j$ runs in normal mode, $L$ contains $r_j[x_a]$ for some copy $x_a$ of $x$.

(ii.2) If $T_j$ runs in failure mode, $L$ contains $r_j[x_a],..., r_j[x_k]$ for some read quorum $R = \{x_{a_1},..., x_{a_l}\}$, and $r_j[x_{ak}]$ for some $x_{ak} \in R$. For each $x_{ak} \in R$, let $\text{VN}(x_{ak}) = \text{VN}(x)$, where $w_i[x_a]$ is the last write on $x_a$ before $a_j[x_{ak}]$. The copy, $x_{ak}$, read by $T_j$ must satisfy $\text{VN}(x_{ak}) = \text{max}(\text{VN}(x_{ak}) | x_{ak} \in R)$.

(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 $x_a$.

(v) $\text{PSG}(L)$ is acyclic.

For example, consider a database with 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

\[
T_0 = w_0[x] \quad T_1 = r_1[x] \quad T_2 = w_2[x] \quad T_3 = r_3[x] \\
\text{VN}_0(x) = 1 \quad \text{VN}_1(x) = \text{VN}_2(x) = \text{VN}_3(x) = 1
\]

A possible mw log is

\[
L_5 = \begin{array}{cccc}
\text{w}_0 \{x_a\} & \text{w}_2 \{x_a\} & \text{a}_3 \{x_a\} & \text{r}_3 \{x_a\} \\
\text{w}_0 \{x_b\} & \text{r}_1 \{x_a\} & \text{a}_3 \{x_b\} & \text{r}_3 \{x_c\} \\
\text{w}_0 \{x_c\} & \text{w}_2 \{x_c\} & \text{a}_3 \{x_c\} & \text{r}_3 \{x_c\}
\end{array}
\]

We can tell that $T_1$ runs in normal mode, because it does not access a read quorum of $x$. We can tell that $T_2$ runs in failure mode, because it does not write all copies of $x$. We can tell that $T_3$ runs in failure mode, because it accesses read quorums. Transactions $T_2$ and $T_3$ are aware of missing writes: $T_2$ is aware of its own missing write on $x$, and $T_3$ is aware of the same missing write. These transactions run in failure mode as required by the definitions.
5.3. Correctness Proof

We prove the algorithm correct by constructing an acyclic LSG for every mw log. The proof is more complex than for qc logs, because the PSG of an mw log does not necessarily include a read order. For example,

\[ \text{PSG}(L_5) = T_0 \]

\[ \text{T}_1 \text{ reads-x-from } \text{T}_0, \text{T}_2 \text{ writes } x, \text{ and } \text{T}_0 \ll \text{T}_2. \]

Hence, to induce a read order, we need \( T_1 \ll T_2 \), but this path does not exist in PSG(\( L_5 \)). It is important to note that \( T_1 \ll T_2 \) is the only path missing from PSG(\( L_5 \)). If we add the edge \( T_1 \rightarrow T_2 \) to PSG(\( L_5 \)), the result is an LSG.

\[ \text{T}_0 \]

\[ \text{T}_1 \]

\[ \text{T}_2 \]

\[ \text{T}_3 \]

We prove that this simple construction works in general.

Hereafter in this section, we use \( L \) to denote an arbitrary mw log.

Let \( \text{RB}(L) = \{ T_j \rightarrow T_k \mid \text{for some } x \text{ and } T_i, T_j \text{ reads-x-from } T_i, T_k \text{ writes } x, \text{ and } T_i \ll T_k \text{ in PSG}(L); i, j, k \text{ distinct} \} \). For example, \( \text{RB}(L_5) = \{ T_1 \rightarrow T_2 \} \). An edge in \( \text{RB}(L) \) is called a \emph{reads-before edge}; it signifies that \( T_j \) reads \( x \) "logically before" \( T_k \) writes \( x \). As noted in the previous paragraph, PSG(\( L \)) does not necessarily contain all reads-before edges. A reads-before edge, \( T_j \rightarrow T_k \), is called a \emph{new edge} if \( T_j \) does not precede \( T_k \) in PSG(\( L \)).

Define \( G(L) \) to be PSG(\( L \)) with all new edges added. We prove that \( G(L) \) is an acyclic LSG.

\textbf{Lemma 5.1.} \( G(L) \) is an LSG.

\textbf{Proof.} Let \( T_i \) and \( T_k \) write \( x \). Since all write quorums for \( x \) intersect, and since the set of all copies subsumes all write quorums, there exists a copy \( x_o \) that \( T_i \) and
$T_k$ both write. This conflict forces $PSG(L)$ to have an edge connecting $T_i$ and $T_k$, and so $PSG(L)$ induces a write order. The write order carries over to $G(L)$ since $PSG(L) \subseteq G(L)$.

Let $T_j$ read-x-from $T_r$; $L$ contains $w_j[x_a] < r_j[x_a]$ for some copy $x_a$. This conflict places the edge $T_i \rightarrow T_j$ in $PSG(L)$, hence in $G(L)$. Let $T_k$ write $x$ with $T_i \ll T_k$. Either $T_j \ll T_k$ in $PSG(L)$, hence $G(L)$, or $T_j \rightarrow T_k$ is placed in $G(L)$ as a new edge. Thus $G(L)$ induces a read order.

The rest of the proof shows that $G(L)$ is acyclic. First, Lemma 5.2 analyzes how new edges arise. This lemma shows that every new edge connects a normal mode transaction to a failure mode transaction. For example, in $L_5$ above, the only new edge is $T_1 \rightarrow T_2$; $T_1$ runs in normal mode and $T_2$ runs in failure mode, as claimed.

**Lemma 5.2.** Let $T_j \rightarrow T_k$ be a new edge.

(i) Define $T_i$ and $x$ as in the definition of reads-before edge. Namely, $T_j$ reads-x-from $T_i$, $T_k$ writes $x$, and $T_i \ll T_k$ in $PSG(L)$. Let $x_a$ be the copy of $x$ that $T_j$ reads. Then, $T_k$ does not write $x_a$.

(ii) $T_k$ runs in failure mode.

(iii) $T_j$ runs in normal mode.

**Proof.** (i) Let $T_l$ be any transaction that writes $x_a$, other than $T_i$. By definition of reads-from, $w_l[x_a] < w_i[x_a]$ or $r_l[x_a] < w_i[x_a]$. In the first case, we have: $T_j \ll T_l$ in $PSG(L)$, because $w_i[x_a] < w_l[x_a]$; $T_l \ll T_k$, by hypothesis; and so $T_l \ll T_i \ll T_k$ in $PSG(L)$. It follows that $l \neq k$, since by point (v) of the definition of $mw$ log, $PSG(L)$ is acyclic.

In the second case, we have $T_j \ll T_l$ in $PSG(L)$, because $r_l[x_a] < w_i[x_a]$. It follows that $l \neq k$, since by definition of new edge, $T_j$ does not precede $T_l$ in $PSG(L)$.

Thus, $T_k$ does not write $x_a$, and part (i) is proved.

(ii) Immediate, since $T_k$ writes $x$ but does not write $x_a$.

(iii) Suppose $T_j$ runs in failure mode. Then $T_j$ reads $x$ using quorum consensus and the proof of Lemma 4.2 shows $T_j \ll T_k$ in $PSG(L)$. This contradicts the definition of a new edge. So, $T_j$ runs in normal mode, as claimed. 

An edge of $PSG(L)$ or $G(L)$, $T_j \rightarrow T_k$, is an **immediate edge** if $T_j$ immediately precedes $T_k$ in the log. (Recall the definition of "immediately precedes" from Section 5.1.) Immediate edges are important because missing write information flows along such edges. An **immediate path** is a path of immediate edges.

Two basic properties are easy to prove:

1. If $op_i < op_j$ and these operations conflict, then $PSG(L)$ contains an immediate path from $T_i$ to $T_j$.

2. $PSG(L)$ contains a path between two transactions iff it contains an immediate path between them.
The next lemma explains how missing write information flows along immediate paths.

**Lemma 5.3.** (i) Let $T_j \rightarrow T_k$ be an immediate edge. Then $MW'(k) \supseteq MW'(j) - \{x_a \mid T_k \text{ writes } x_a\}$.

(ii) Let $P$ be an immediate path from $T_k$ to $T_j$. Let $T_k$ write $x$, but not $x_a$, and let $T_j$ run in normal mode. Then, some transaction on $P$ other than $T_j$ writes $x_a$.

**Proof.** (i) By definition,

$$MW(K) = \bigcup_{\text{all other } T_i \text{ that immediately precede } T_k} MW'(i) \cup \{x_a \mid T_k \text{ writes } x_a \text{ but not } x_a\}$$

$$MW'(k) = MW(k) - \{x_a \mid T_k \text{ writes } x_a\}.$$

Plugging the definition of $MW(k)$ into $MW'(k)$, we get

$$MW'(k) = MW'(j) \cup \{x_a \mid T_k \text{ writes } x_a \}.$$

Thus,

$$MW'(k) \supseteq MW'(j) - \{x_a \mid T_j \text{ writes } x_a\}$$

as claimed.

(ii) Let us write $P$ as $T_k \rightarrow T_1 \rightarrow \cdots T_n \rightarrow T_j$. Applying part (i) iteratively, we find $MW'(ln) \supseteq MW'(k) - \{x_a \mid \text{ some } T_i \text{ writes } x_a\}$, while by definition $MW'(j) \supseteq MW'(ln)$. So, $MW'(j) \supseteq MW'(k) - \{x_a \mid \text{ some } T_i \text{ writes } x_a\}$.

Since $T_k$ writes $x$ but not $x_a$, $x_a \in MW'(k)$. Since $T_j$ runs in normal mode, $MW(j)$ is empty. Thus, some transaction on $P$ other than $T_j$ writes $x_a$, as claimed.

If $T_j \rightarrow T_k$ is a new edge, then, by definition $PSG(L)$ does not contain a path from $T_j$ to $T_k$. The next lemma proves that $PSG(L)$ does not have a path in the other direction either. In other words, every new edge connects a pair of transactions that are incomparable (not connected) in $PSG(L)$. Referring back to example $L_5$, the only new edge is $T_1 \rightarrow T_2$; as claimed $T_1$ and $T_2$ are incomparable in $PSG(L)$.

**Lemma 5.4.** Let $T_j \rightarrow T_k$ be a new edge. Then there is no immediate path from $T_k$ to $T_j$ in $PSG(L)$.

**Proof.** Define $T_i$, $x$ and $x_a$ as in Lemma 5.2. That is, $T_j$ reads $x_a$ from $T_i$, $T_k$ writes $x$ but not $x_a$, and $T_i \ll T_k$ in $PSG(L)$. Suppose, for the sake of deriving a contradiction, that $P$ is an immediate path from $T_k$ to $T_j$. Then, by Lemma 5.3, some $T_i$...
on $P$ ($l \neq j$) writes $x_a$; $l = i$ is possible. By definition of reads-from, $w_i[x_a] \leq w_j[x_a]$, or $r_j[x_a] < w_i[x_a]$.

In the first case, we have: $T_k \preceq T_l$ in $PSG(L)$ via $P$; $T_i \preceq T_l$ in $PSG(L)$, because $w_i[x_a] \leq w_i[x_a]$; hence $T_k \preceq T_l \preceq T_i$ in $PSG(L)$. But this is impossible since by assumption, $T_i \preceq T_k$ and $PSG(L)$ is acyclic.

In the second case, we have: $T_i \preceq T_j$ in $PSG(L)$ via $P$; and $T_j \preceq T_l$ in $PSG(L)$, because $r_j[x_a] < w_i[x_a]$. This, too, is impossible because $PSG(L)$ is acyclic. Since both cases lead to contradiction, we conclude that $P$ cannot exist, and the lemma is proved.  

**Lem 5.5.** $G(L)$ is acyclic.

**Proof.** If $G(L)$ contains a cycle, it must contain a cycle all of whose edges are immediate or new. Among all such cycles, let $C$ be one with the fewest new edges. We prove that $C$ cannot exist, hence $G(L)$ is acyclic. There are three cases.

**Case 1.** $C$ has 0 new edges. In this case, $C$ is also present in $PSG(L)$. This is impossible, because $PSG(L)$ is acyclic by definition of $mw$ log.

**Case 2.** $C$ has 1 new edge. Let $T_j \rightarrow T_k$ be the new edge. The part of $C$ that starts at $T_k$ and ends at $T_j$ is an immediate path from $T_k$ to $T_j$. By Lemma 5.4, this part cannot exist.

**Case 3.** $C$ has 2 or more edges. We can write $C$ in the form

$$C = \begin{array}{c|c|c|c|c|c|c|c|c|c|c} \text{new} & \text{immediate} & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \text{new} \\ T_j \rightarrow T_k \rightarrow \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & T_m \rightarrow T_n \rightarrow \ldots T_j \end{array}$$

where $T_j \rightarrow T_k$ is a new edge and $T_m \rightarrow T_n$ is the next edge along the cycle. Observe the following:

1. The path from $T_k$ to $T_m$ is an immediate path.
2. There exists an $x_a$ such that $T_j$ reads $x_a$ and $T_k$ writes $x$ but not $x_a$.
3. $T_m$ runs in normal mode (by Lemma 5.2).
4. Some $T_i$ along the path between $T_k$ and $T_m$ writes $x_a$ (by Lemma 5.3).

We now use $T_j$ to find a cycle with fewer new edges than $C$. Let $P_{jl}$ be the part of $C$ from $T_j$ to $T_i$. Let $P_{ji}$ be the rest of $C$ from $T_i$ to $T_j$.

$$C = \begin{array}{c|c|c|c|c|c|c|c|c|c|c} \text{new} & \text{immediate} & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \text{new} \\ T_j \rightarrow T_k \rightarrow \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & T_m \rightarrow T_n \rightarrow \ldots T_j \end{array}$$

There are two subcases.
3.1. \( r_j[x_a] < w_j[x_a] \).
Then there is an immediate path \( P_{ji} \) from \( T_j \) to \( T_i \). Let \( C' \) be the cycle consisting of \( P_{ji} \) followed by \( P'_{ij} \).

\[
\begin{array}{c}
\text{immediate} \\
\hline
C' = T_j \rightarrow \cdots \rightarrow T_i \rightarrow \cdots \rightarrow T_m \rightarrow T_n \rightarrow \cdots \rightarrow T_j \\
\hline
P_{ij} \quad P'_{ji}
\end{array}
\]

\( C' \) is a cycle containing new and immediate edges, with fewer new edges than \( C \). This contradicts our choice of \( C \) as a cycle with the fewest new edges.

3.2. \( w_j[x_a] < r_j[x_a] \).
Then there is an immediate path \( P'_{ij} \) from \( T_i \) to \( T_j \). Let \( C' \) be the cycle consisting of \( P_{ij} \) followed by \( P'_{ji} \).

\[
\begin{array}{c}
\text{new} \quad \text{immediate} \\
\hline
C' = T_j \rightarrow T_k \rightarrow \cdots \rightarrow T_i \rightarrow \cdots \rightarrow T_m \rightarrow T_n \rightarrow \cdots \rightarrow T_j \\
\hline
P_{ji} \quad P'_{ij}
\end{array}
\]

Again, \( C' \) is a cycle containing new and immediate edges, with fewer new edges than \( C \). (Indeed, in this case, \( C' \) only contains one new edge.) And, again, this contradicts our choice of \( C \).

We conclude that cycle \( C \) cannot exist, and so \( G(L) \) is acyclic as desired. \( \square \)

The main result follows.

**Theorem 5.** Every mw log is 1-SR. Thus missing writes is a correct replicated data algorithm.

**Proof.** By Lemmas 5.1 and 5.5, \( G(L) \) is an acyclic LSG. The result follows by Theorem 2. \( \square \)

6. **Summary**

We have extended serializability to analyze the correctness of replicated data algorithms. 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 proved that an execution is 1-SR iff it has an acyclic logical serialization graph. We used this result to prove the correctness of quorum consensus and missing writes algorithms.

ACKNOWLEDGMENTS

We thank Umesh Dayal, Vassos Hadzilacos, and Dale Skeen for many helpful discussions. We thank an anonymous referee for his insightful comments.

REFERENCES


571/31/3-6


