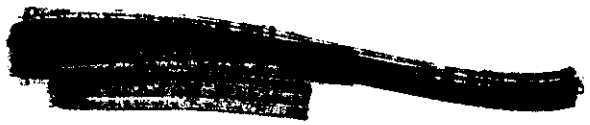
CMU-CS-85-108

Atomicity vs. Availability: Concurrency Control for Replicated Data

Maurice Herlihy
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213
5 February 1985

Abstract

Data managed by a distributed program may be subject to consistency and availability requirements that must be satisfied in the presence of concurrency, site crashes, and network partitions. This paper proposes two integrated methods for implementing concurrency control and replication for data of abstract type. Both methods use quorum consensus. The Consensus Locking method minimizes constraints on availability, and the Consensus Scheduling method minimizes constraints on concurrency. These methods systematically exploit type-specific properties of the data to provide better availability and concurrency than methods based on the conventional read/write classification of operations. Necessary and sufficient constraints on correct implementations are derived directly from the data type specification. These constraints reveal that an object cannot be replicated in a way that simultaneously minimizes constraints on both availability and concurrency.



This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

1. Introduction

A *distributed system* consists of multiple computers (called sites) that communicate through a network. A *distributed program* is one whose modules reside and execute at multiple sites in a distributed system. The components of a distributed system can fail independently: sites can crash, and communication links can be interrupted. Distributed programs should be designed to tolerate such failures. Atomicity and availability are two kinds of fault-tolerance useful to distributed programs. *Atomicity* ensures that the data managed by distributed programs are not rendered inconsistent by failures, while *Availability* ensures that the data remain accessible in the presence of a certain number of failures. In this paper, we propose new methods for implementing atomicity and availability. We show that availability and atomicity are not independent, and we provide a precise characterization of their interrelation.

The data managed by a distributed program may be subject to consistency constraints that must be preserved in the presence of failures and concurrency. These constraints apply not only to individual pieces of data, but also to distributed sets of data. For example, a distributed banking system might be subject to the constraint that the books balance: money is neither created nor destroyed, only transferred from one ledger to another. A widely-accepted approach to ensuring consistency is to make the activities that manage the data *atomic*. Atomicity encompasses two properties: indivisibility and recoverability. *Indivisibility* means that the execution of one activity never appears to overlap (or contain) the execution of another, while *recoverability* means that the overall effect of an activity is all-or-nothing: it either succeeds completely, or it has no effect. An unsatisfactory way to ensure indivisibility is to constrain activities to execute one at a time. Instead, activities are typically allowed to execute concurrently as long as they remain *serializable*: their overall effect is as if they had executed in a serial order.

The data managed by a distributed program may also be subject to *availability* requirements: the data should be accessible with high probability. Availability in the presence of failures can be enhanced by storing the data redundantly at multiple sites, a technique called *replication*. For example, the availability of a bank account might be enhanced by keeping additional copies of the records at multiple sites. If one set of records becomes temporarily or permanently inaccessible, activities might be able to progress using a different set. Care must be taken that the replicated records are managed properly; enhanced availability is of little use if activities erroneously observe obsolete or inconsistent data. Consequently, we require that replication be *transparent*: the only observable effect of replication is to make the data more available.

This paper proposes two new methods for implementing availability and atomicity. Both methods

systematically exploit type-specific properties of the data to impose fewer constraints on availability and concurrency than conventional methods that treat all operations as reads or writes. In both methods, availability is achieved by *quorum consensus*: associated with each operation of the data type is a set of quorums, which are collections of sites whose cooperation suffices to execute the operation. Constraints on quorum assignment (and hence on realizable availability properties) are derived directly from the data type specification.

We first propose *Consensus Locking*, a method in which scheduling decisions use predefined operation conflicts. This method favors availability; it minimizes constraints on quorum assignment, and it minimizes message traffic, but it supports a suboptimal level of concurrency. We then propose *Consensus Scheduling*, a more complex method in which scheduling decisions may use state information. This method favors concurrency; it minimizes constraints on interleaving of activities, but it requires additional message traffic, and it places additional constraints on quorum assignment. Each method is optimal for the information it uses; no quorum-consensus method can impose strictly fewer constraints on both quorum assignment and concurrency. The interdependence between availability and atomicity is fully characterized by the notion of an atomic dependency relation. Our analysis shows that an object cannot be replicated in a way that simultaneously optimizes constraints on concurrency and availability. A minimal set of constraints on availability requires a suboptimal level of concurrency, and vice-versa.

Section 2 presents a brief overview of related work, and Section 3 presents our model of computation. Consensus Locking is described in Section 4, and Consensus Scheduling in Section 5. We conclude with a discussion in Section 6. Correctness proofs are given in the appendix.

2. Related Work

Most mechanisms for implementing atomicity in distributed systems fall into three broad categories: two-phase locking schemes (e.g. [11, 21, 29]), timestamping schemes (e.g. [28, 27, 26]), and hybrid schemes employing both locking and timestamps (e.g. [8, 9, 2, 3]). Our model for atomic objects is essentially that of Weihl [32, 33].

Early file replication methods did not attempt to preserve serializability; the value read from a file is not necessarily the value most recently written [1, 19, 31]. Non-serializable replication methods for directories have also been proposed [25, 6, 12].

In the *available copies* replication method [14], failed sites are dynamically detected and configured out of the system, and recovered sites are detected and configured back in. Activities may read from

any available copy, and must write to all available copies. Systems based on variants of this method include SDD-1 [15] and ISIS [5]. Unlike the methods proposed in this paper, the available copies method does not preserve serializability in the presence of communication link failures such as partitions.

In the *true-copy token* scheme [24], a replicated file is represented by a collection of copies. Copies that reflect the file's current state are called *true copies*, and are marked by *true-copy tokens*. True copies can be moved to permit activities to operate on local data. This method preserves serializability in the presence of crashes and partitions, but the availability of a replicated file is limited by the availability of the sites containing its true copies.

A formal model for concurrency control in replicated databases proposed by Bernstein and Goodman can be used to show the correctness of several replication methods [4]. The model makes two assumptions that do not apply to the replication methods proposed in this paper: that a replicated object is implemented by multiple copies, and that all information about operations is captured by a simple read/write classification. We will see that these assumptions unnecessarily restrict availability and concurrency.

The earliest use of quorum consensus is a file replication method due to Gifford [13]. A quorum-consensus replication method for directories has been proposed by Bloch, Daniels, and Spector [7]. These methods can be viewed as specially optimized instances of *General Quorum Consensus*, a replication method for arbitrary data types [16, 17]. Like the methods proposed in this paper, General Quorum Consensus systematically exploits type-specific properties to enhance availability. Unlike the methods proposed here, it relies on a concurrency control mechanism provided by a lower level of the system. General Quorum Consensus includes a reconfiguration technique that can readily be extended to the replication methods proposed in this paper.

Extensions to quorum consensus that further enhance availability in the presence of partitions have been proposed for files by Eager and Sevcik [10] and for arbitrary data types by the author [16].

3. Assumptions and Terminology

We admit two kinds of faults: sites may crash and communication links may be interrupted. When a site crashes, its resident data becomes temporarily or permanently inaccessible. Communication link failures result in lost messages; garbled and out-of-order messages can be detected (with high probability) and discarded. Transient communication failures may be hidden by lower level protocols, but longer-lived failures can cause *partitions*, in which functioning sites are unable to communicate.

A failure is detected when a site that has sent a message fails to receive a response after a certain duration. The absence of a response may indicate that the original message was lost, that the reply was lost, that the recipient has crashed, or simply that the recipient is slow to respond.

The basic units of synchronization and recovery are atomic activities called *actions*, or transactions. An action is a sequential process. An action that completes all its changes successfully *commits*; otherwise it *aborts*, and any changes it has made are undone. We assume that failures that prevent an action from committing are turned into aborts using a distributed commit protocol [11, 30].

The basic containers for data are called *objects*. Each object has a *type*, which defines a set of possible *states* and a set of primitive *operations* that provide the (only) means to create and manipulate objects of that type. For example, a bank account might be represented by an object of type *Account* whose state is given by a non-negative dollar amount, initially zero. The *Account* data type provides two operations: *Credit* and *Debit*. The *Credit* operation increments the account balance:

Credit = Operation(sum: Dollar).

The *Debit* operation attempts to decrement the balance:

Debit = Operation(sum: Dollar) Signals (Overdrawn).

A debit is successful only if the balance of the account exceeds the amount to be debited. Otherwise the operation returns with an exception [23], leaving the account balance unchanged.

Following [32, 33], an object's behavior is characterized by two specifications: its *serial* specification characterizes its behavior in the absence of failures and concurrency, and its *behavioral* specification characterizes the level of concurrency it supports. For both serial and behavioral specifications, computations are modeled as sequences of operation executions. Although computations in a distributed system can be viewed as partially ordered sets of events, an unambiguous total order on events can be imposed by a system of logical clocks [22]. A disadvantage of logical clocks is that the logical ordering of events may differ markedly from their physical ordering, a problem that can be alleviated by techniques for approximate synchronization [22].

In the absence of failures and concurrency, an object's state is given by a *serial history*. A serial history is a sequence of *events*, where an event is a paired operation invocation and response. For example,

```
Credit($20);Ok()
Debit($15);Ok()
Debit($10);Overdrawn()
```

is a serial history in which an account, initially empty, is credited $20, debited $15, and an attempt to debit $10 returns with an exception. A *serial specification* for an object is a set of *legal* serial histories for that object. For example, the serial specification for the *Account* data type includes only serial histories in which the account balance never becomes negative. We assume that serial specifications are prefix-closed: any prefix of a legal serial history is legal.

In the presence of failure and concurrency, an object's state is given by a *behavioral history*. A behavioral history is a sequence of operation executions and *Commit* and *Abort* events. To keep track of interleaving, each event is associated with an action. For example the following is a behavioral history involving two actions, A and B:

Credit($5);Ok() A
Credit($5);Ok() B
Commit A
Debit($10);Ok() B
Commit B

Here, action A begins and credits $5 to the account, B begins and also credits $5, A commits, B debits $10 from the account, and commits. The ordering of operation executions in a behavioral history reflects the order in which the responses are returned, not necessarily the order in which the invocations occurred. A *behavioral specification* for an object is a set of *legal* behavioral histories for that object. We assume that all behavioral specifications are prefix-closed and *on-line*: the result of appending a *Commit* event for an active action to a legal behavioral history yields a legal behavioral history.

The serial and behavioral specifications for the objects considered in this paper are related by the notion of *atomicity*. Let $\gg$ denote a total order on committed and active actions, and let H be a behavioral history. The *serialization* of H *with respect to* $\gg$ is the serial history h constructed as follows:

- Discard all events associated with aborted actions.
- Reorder the events so that if B $\gg$ A then the subsequence of events associated with A precedes the subsequence of events associated with B, for all actions A and B.
- Discard all *Commit* events, and all action identifiers.

H is *serializable with respect to* $\gg$ if h is a legal serial history (i.e. is included in the object's serial specification). H is *serializable* if it is serializable with respect to some ordering $\gg$. H is *atomic* if the subhistory associated with committed events is serializable. An object is atomic if every history in its behavioral specification is atomic. All objects considered in this paper are atomic.

4. Consensus Locking

We are now ready to describe *Consensus Locking*, the first of the two novel concurrency control/replication methods proposed in this paper. Consensus Locking provides a systematic method for transforming a single-site serial implementation of a data type into a replicated atomic implementation. By taking advantage of type-specific properties, Consensus Locking provides better concurrency and availability than replication methods based on the conventional read/write classification of operations.

4.1. Overview

A *replicated object* is an object whose state is stored redundantly at multiple sites to enhance availability. Replicated objects are implemented by two kinds of modules: *repositories* and *front-ends*. Repositories provide long-term storage for the object's state, while front-ends carry out operations for clients. In database terminology, front-ends correspond roughly to transaction managers and repositories correspond roughly to data managers [3].

To apply an operation to a replicated object, a client sends an invocation to a front-end for the object. The front-end reads data from some collection of repositories, carries out a local computation, sends updates to some collection of repositories, and returns a response to the client. The client must first locate an available front-end for the object, and the front-end must in turn locate enough available repositories to carry out the operation. Front-ends can be replicated to an arbitrary extent, perhaps placing one at each client's site, implying that the availability of the replicated object is dominated by the availability of its repositories.

A *quorum* for an operation is any set of repositories whose cooperation suffices to execute that operation. It is convenient to divide a quorum into two parts: a front-end executing an operation reads from an *initial quorum* and writes to a *final quorum*. (Either the initial or final quorum may be empty.) A quorum for an operation is any set of repositories that includes both an initial and a final quorum.

A *log* is a data structure representing a behavioral history. A log consists of a sequence of *entries*, each consisting of a timestamp, an event, and an action identifier. For example, the following is a log for an account:

```
1:00 Credit($5);Ok() A
1:15 Credit($5);Ok() B
1:30 Commit A
1:45 Debit($10);Ok() B
2:00 Commit B
```

A replicated object is represented by a log whose entries are partially replicated among a set of repositories. For example, the following is a schematic representation of an *Account* replicated among three repositories. For readability, a "missing" entry at a repository is shown as a blank space.

R1	R2	R3
1:00 Credit($1);Ok() A	1:00 Credit($1);Ok() A	
	1:15 Credit($2);Ok() B	1:15 Credit($2);Ok() B
1:30 Credit($3);Ok() C		1:30 Credit($3);Ok() C
1:45 Commit A	1:45 Commit A	
	2:00 Abort B	2:00 Abort B

This account has been credited three times by three actions, but no single repository has an entry for all three credits.

It should be emphasized that logs are intended to serve as a conceptual model for the replicated data, not as a literal design for an implementation. More compact representations can be achieved by discarding unneeded entries, and by merging adjacent entries. Some techniques are straightforward, such as discarding entries for aborted actions. Other techniques are type-dependent, such as replacing a prefix of the committed entries in an account's log with a single timestamped balance. A compaction technique for replicated directories is given in [7], and techniques for other data types are given in [16, 17]. We do not address compaction techniques in this paper because we wish to focus directly on availability and concurrency issues.

Each invocation permitted by the data type has an *initial lock* at each repository, and each event has a *final lock*. An action is granted an initial lock when the repository agrees to co-operate with an invocation, and a final lock when the repository agrees to accept an entry for a new event. Scheduling decisions are based on predefined conflicts between certain initial and final locks. All locks are *strict two-phase*: an action holds its locks until it commits or aborts.

An operation is executed in the following steps:

- The client sends the invocation and action identifier to a front-end, which forwards them to an initial quorum of repositories.
- Each repository in the initial quorum grants the action an initial lock for the invocation if no other action holds a conflicting final lock. Otherwise, the action is delayed until the conflicting locks are released. Once the initial lock has been granted, the repository sends its log to the front-end.
- The front-end constructs a log by merging the responses from an initial quorum for the invocation. The *view* is the serialization of the front-end's log in which committed actions

are ordered by the timestamp order of their *Commit* entries, and the client's action is ordered last. Entries for all other actions are discarded. A single-site serial implementation of the data type chooses a response using the view. The front-end generates a new timestamp, creates an entry to record the new event, appends the new entry to its log, and sends the log to a final quorum of repositories.

- Each repository in the final quorum grants the action a final lock for the new event if no other uncommitted action holds a conflicting initial lock. Otherwise the action is delayed until the conflicting locks are released. Once the final lock has been granted, the repository merges the front-end's updated log with its resident log and returns an acknowledgment to the front-end.

- As soon as a final quorum of repositories has acknowledged the update, the front-end returns the response to the client.

As part of an action's commit protocol [11, 30], it reads the clocks at each repository whose contents it has read or written, generates a later timestamp, and inserts a *Commit* entry in the log at each visited repository. If an action aborts, *Abort* entries will eventually appear at all repositories visited by that action.

To illustrate this method, let us trace how action C might debit four dollars from the the replicated *Account* shown above. The front-end forwards the request to R1 and R2. Both repositories grant an initial *Debit* lock, and respond with their logs. The result of merging these logs yields:

1:00 Credit($1);Ok() A
1:15 Credit($2);Ok() B
1:30 Credit($3);Ok() C
1:45 Commit A
2:00 Abort B

which the front-end serializes as:

Credit($1);Ok()
Credit($3);Ok()

The view indicates that the account balance exactly covers the debit, so the front-end appends an entry for a successful debit to its log, which it sends to R2 and R3. These repositories grant final locks, and merge the updated log with their resident logs. The account's final state is:

R1	R2	R3
1:00 Credit($1);Ok() A	1:00 Credit($1);Ok() A	1:00 Credit($1);Ok() A
	1:15 Credit($2);Ok() B	1:15 Credit($2);Ok() B
1:30 Credit($3);Ok() C	1:30 Credit($3);Ok() C	1:30 Credit($3);Ok() C
1:45 Commit A	1:45 Commit A	1:45 Commit A
	2:00 Abort B	2:00 Abort B
	2:15 Debit($4);Ok() C	2:15 Debit($4);Ok() C

C holds initial *Debit* locks at R1 and R2, and final locks at R2 and R3.

4.2. Serial Dependency

Consensus locking is correct only if quorum assignments and lock conflicts are chosen properly. There are two essential requirements: (i) an invocation's view must contain enough information to choose a "correct" response, and (ii) the lock conflicts arising in the course of an operation execution must ensure serializability. We now describe a systematic method for deriving a correct and optimal set of constraints on quorum intersection and lock conflict directly from the data type's serial specification. By *correct*, we mean that any implementation consistent with the constraints is correct, and by *optimal*, we mean that no weaker set of constraints yields only correct implementations.

Let $\succ$ be a relation between invocations and events. Informally, a subhistory g of h is *closed* with respect to $\succ$ if whenever g contains an event e of h it also contains every earlier event e' of h such that $e.inv \succ e'$. (Here, $e.inv$ denotes the invocation part of the event e.) More precisely, let INV be the set of invocations for an object, RES the set of responses, and EVENT = INV × RES the set of events. A serial history of length n can be modeled as a map h: {1,...,n} → EVENT.

> **Definition 1:** A history g of length m is a closed subhistory of h with respect to $\succ$ if there exists an injective order-preserving map s: {1,...,m} → {1,...,n} such that $g(i) = h(s(i))$, and if $e.inv \succ e'$, $i > i'$, $h(i) = e$, $h(i') = e'$, and $s(j) = i$, then there exists j' such that $s(j') = i'$.

We omit mention of $\succ$ when it is clear from context.

The correctness condition for Consensus Scheduling is based on the notion of a *serial dependency* relation between invocations and events. Let h • [inv;res] denote the result of appending the event [inv;res] to the serial history h.

> **Definition 2:** Given a serial specification T, a relation $\succ$ between invocations and events is a serial dependency relation for T if for all invocations *inv*, all responses *res*, all legal serial histories h, and all closed subhistories g containing all events e of h such that $inv \succ e$:
>
> g • [inv;res] is legal ⇒ h • [inv;res] is legal.

Informally, this definition states that a correct response to an invocation can be chosen by observing any closed legal subhistory that contains all the events on which the invocation depends.

A replicated object's *quorum intersection relation* consists of all pairs of invocations and events such that each initial quorum for the invocation has a non-empty intersection with each final quorum for the event. An object's *lock conflict relation* consists of all pairs of invocations and events such that initial locks for the invocation conflict with final locks for the event. A replicated atomic object implemented by Consensus Locking satisfies the serial specification T if and only if:

> The quorum intersection relation and the lock conflict relation satisfy a common serial dependency relation for T.

This claim is proved in the appendix. A serial dependency relation is *minimal* if no smaller relation is a serial dependency relation. Minimal relations correspond to minimal sets of constraints. As shown in [16, 17], a data type may have several distinct minimal serial dependency relations.

The *Account* data type has a unique minimal serial dependency relation. Because the response to a *Debit* invocation depends on the account balance, *Debit* invocations depend on *Credit* events and on successful *Debit* events, but not on *Debit* events that signaled *Overdrawn*. Because the response to a *Credit* is always *Ok*, *Credit* invocations need not depend on any events. This relation implies that initial locks for *Debit* must conflict with final locks for both *Credit* and successful *Debit*, but initial and final *Credit* locks need not conflict, permitting *Credit* operations to execute concurrently.

Similarly, each initial quorum for *Debit* must intersect each final quorum for both *Credit* and successful *Debit*, but initial and final *Credit* quorums need not intersect. An *Account* replicated among five identical sites permits the following quorum assignments for *Credit* and successful *Debit* events:

Credit	(0,1)	(0,2)	(0,3)
Debit	(5,1)	(4,2)	(3,3)

Here, an entry of the form (i,f) indicates that any i repositories constitute an initial quorum, any f repositories constitute a final quorum, and hence any $max(i,j)$ repositories constitute a quorum for the event. Each column represents a distinct quorum assignment. For example, the column on the left represents a quorum assignment where all five sites constitute a quorum for a successful *Debit* event, and any site constitutes a quorum for a *Credit* event. Each of these assignments is *minimal*: no other correct quorum assignment permits a strictly larger set of quorums. (For example, although the quorum assignment Credit = (0,4) and Debit = (2,4) is correct, it is not minimal, because each of its quorums is also permitted by the assignment Credit = (0,2) and Debit = (4,2), but not vice-versa.) Given n sites, there are $\lceil n/2 \rceil$ distinct minimal quorum choices for *Credit* and *Debit*.

4.3. Discussion

A data type's serial dependency relations govern both the concurrency and the availability realizable by Consensus Locking. Concurrency is determined by lock conflicts. Availability is determined by quorum assignments; the likelihood an invocation will succeed is the likelihood an appropriate quorum will be available (neglecting issues such as synchronization conflicts or resource exhaustion). If each quorum for one operation is required to intersect each quorum for another, then their levels of availability are inversely related, because if one operation's quorums are made smaller

(rendering it more available) then the other's quorums must be made correspondingly larger (rendering it less available).

The range of availability properties realizable by Consensus Locking compares favorably to that of replication methods based on the conventional read/write classification of operations. If both *Credit* and *Debit* were classified as writes, then under quorum consensus, initial and final *Credit* quorums would be required to intersect, reducing the range of quorum assignments from $\lceil n/2 \rceil$ to one. Although the Consensus Locking and Available Copies schemes cannot be compared directly because of their different fault models, it is worth mentioning that the latter would require *Credit* and *Debit* to update all available copies, producing more message traffic in the absence of failures.

Elsewhere [16, 17], we have shown that if actions are physically serialized (or, equivalently, serialized by a lower-level mechanism such as two-phase read/write locks) then the quorum intersection relation must be a serial dependency relation. Clearly, enhancing concurrency cannot reduce the constraints on quorum intersection, simply because any quorum choice that works in the presence of concurrency must also work if actions happen to execute serially. Serial dependency is thus the weakest constraint on quorum intersection that can be imposed by any integrated concurrency control/replication method.

Like most two-phase locking protocols, Consensus Locking is subject to deadlocks, which can be handled by standard techniques, such as timeouts or deadlock detection [20]. Consensus Locking supports more concurrency than several comparable protocols, even in the absence of replication. Two-phase read/write locks [11] can be viewed as a degenerate form of Consensus Locking in which every initial (read) lock conflicts with every final (write) lock, and pairs of final (write) locks also conflict. Korth [21] and Bernstein *et al.* [2] have proposed type-specific two-phase locking protocols for single-site objects whose operations are total and deterministic. In both protocols, operations whose invocations do not commute have conflicting locks. Informally, two invocations commute if applying them in either order yields the same results and the same final state. It can be shown that the relation $inv \succ e$ if and only if *inv* and *e.inv* do not commute is a serial dependency relation, but not necessarily a minimal relation. Consequently, Consensus locking can realize any level of concurrency permitted by these commutativity-based locking schemes, but not vice-versa. For example, *Enq* invocations for the FIFO queue data type do not commute, but Consensus Locking permits concurrent *Enq* executions. The additional power of Consensus Locking is due to the use of timestamps to provide a globally known serialization ordering. Weihl [33] has proposed a type-specific two-phase locking protocol for single-site objects that supports a level of concurrency incomparable to that of Consensus Locking.

Consensus Locking is an example of *hybrid atomicity* [32, 33]; actions are serializable in the timestamp order of their *Commit* entries. Hybrid atomicity is a *local atomicity property*; if every object in a system is hybrid atomic, the system as a whole will be atomic. Hybrid atomicity encompasses two-phase locking schemes [11, 21, 29] and schemes combining locking and timestamps [9, 8, 2, 3]. The principal limitation of Consensus Locking is that it cannot realize all the concurrency permitted by hybrid atomicity. For example, although Consensus Locking always prevents *Debit* and *Credit* operations from executing concurrently, this restriction is not always necessary. Consider the following scenario, where committed action A has deposited $15 in three separate invocations, and uncommitted action B has deposited $5.

R1	R2	R3
	1:00 Credit($5);Ok() A	1:00 Credit($5);Ok() A
1:15 Credit($5);Ok() A	1:15 Credit($5);Ok() A	
1:30 Credit($5);Ok() A	1:30 Credit($5);Ok() A	
1:45 Commit A	1:45 Commit A	1:45 Commit A
2:00 Credit($5);Ok() B	2:00 Credit($5);Ok() B	

An attempt by C to debit $15 will be delayed by conflicts with B's final *Credit* locks. Note, however, that it is not really necessary to delay C, because the account balance covers the debit regardless of the order in which B and C commit. If C had attempted to debit $20, however, then it would indeed have to be delayed until B commits or aborts, because B's outcome will determine whether the account balance covers the debit. Consensus Locking cannot distinguish between these scenarios, and therefore C must be delayed in both instances.

An inability to take full advantage of state information for scheduling is a characteristic of any concurrency control scheme in which scheduling decisions are made at repositories. In the example above, no repository has enough information to recognize that the account balance established by the committed action covers the attempted debit. In general, no information about the account balance can be ascertained from the entries residing at any single repository, because there may be an arbitrary number of unobserved *Credit* and *Debit* entries recorded elsewhere.

A similar problem arises with *partial* operations, which are operations that cannot be executed in certain states. For example, a replicated FIFO queue might provide a partial *Deq* operation that is undefined when applied to an empty queue. It is desirable to delay a *Deq* applied to an empty queue until an item has been enqueued and committed. Unfortunately, such scheduling decisions cannot be made at the repositories, because no individual repository can determine whether a queue is empty.

Static atomicity [32, 33] is an alternative to hybrid atomicity. Each action executes a *Begin* event before accessing any data; actions must be serializable in the timestamp order of their *Begin* entries. Static atomicity encompasses multiversion timestamp schemes [28, 26]. A Consensus Locking method satisfying static atomicity is proposed in [16]. In many respects, the static atomic Consensus Locking method resembles the hybrid atomic method proposed in this paper. Both methods require that the lock conflict and quorum intersection relations satisfy a common serial dependency relation, both methods make scheduling decisions exclusively on the basis of predefined operation conflicts, and both methods take advantage of type-specific properties to provide more concurrency than comparable methods that classify operations simply as reads and writes. Unlike the hybrid atomic method, the static atomic method is not subject to deadlock, but actions may be forced to restart. The hybrid and static atomic Consensus Locking methods support incomparable levels of concurrency: each permits interleavings not permitted by the other.

5. Consensus Scheduling

This section introduces *Consensus Scheduling*, the second of the two concurrency control/replication methods proposed in this paper. Consensus Scheduling provides a systematic way of transforming a single-site atomic implementation of an object into a replicated implementation satisfying the same behavioral specification. Consensus Scheduling supports more concurrency than Consensus Locking because scheduling decisions are not restricted to the information residing at a single repository. Instead, scheduling decisions are made at front-ends using information collected from multiple repositories. Although Consensus Scheduling supports more concurrency than Consensus Locking, it may place additional constraints on quorum assignment, and it may require additional message traffic.

As before, repositories provide long-term storage for the object state, and front-ends execute operations for clients. As in the Consensus Locking method, each repository manages a log. Unlike Consensus Locking, each front-end encapsulates a single-site *atomic* implementation of the data type satisfying a behavioral specification S. We assume that S is on-line and prefix-closed.

An operation is executed in the following steps.

- The client sends the invocation and action identifier to a front-end, which forwards them to an initial quorum of repositories.
- Each repository in the initial quorum responds by with its log.
- The front-end merges the logs in timestamp order to construct a *view*. A single-site atomic implementation of the data type uses the view to choose a response. If the

front-end cannot choose a response, perhaps because it must await the outcome of another action, it waits for the object's state to change, and restarts the protocol.

- The front-end generates a new timestamp, appends the new entry to the view, and sends the updated view to a final quorum of repositories. Each repository in the final quorum merges the view with its resident log and returns an acknowledgment to the front-end.

- As soon as the front-end receives acknowledgments from a final quorum, the response is returned to the client.

Just as for Consensus Locking, an action's *Commit* or *Abort* entries are inserted in the log of each repository it has visited.

Some form of short-term synchronization is needed to ensure that front-ends do not interfere with one another. For example, two actions should not simultaneously withdraw the last $10 in an account. One solution is to place at each repository a short-term mutual exclusion lock that must be acquired by the front-end before reading or updating the log. Alternatively, more concurrency would be permitted by placing short-term initial and final locks at each repository, using the quorum intersection relation (characterized below) as the lock conflict relation. We emphasize that short-term locks serialize individual operation invocations, not actions. Unlike two-phase locks, short-term locks are released as soon as a response has been recorded at a final quorum, or as soon as the front-end has decided that no response is currently possible. Short-term locks may be subject to deadlock. If a repository breaks a short-term lock, the front-end must restart the interrupted operation, but not necessarily the entire action.

To illustrate this method, let us trace how action **C** might debit five dollars from the the replicated *Account* shown below.

R1	R2	R3
	1:00 Credit($10);Ok() A	1:00 Credit($10);Ok() A
	1:15 Commit A	1:15 Commit A
1:30 Credit($5);Ok() B	1:30 Credit($5);Ok() B	

Under Consensus Locking, **C** would be unable to acquire initial *Debit* locks until **B** releases its final *Credit* locks. Under Consensus Scheduling, however, when R1 and R2 receive the request, they grant short-term locks to the front-end, and respond with their logs. The result of merging these logs yields:

1:00 Credit($10);Ok() A
1:15 Commit A
1:30 Credit($5);Ok() B

The view indicates that the ten dollars deposited by **A** will cover the debit regardless of **B**'s outcome,

so the front-end appends the new *Debit* entry to its view, which it sends to R2 and R3. When the front-end confirms that the view has been merged with the repositories' logs, it releases all its short-term locks. The account's final state is:

R1	R2	R3
	1:00 Credit($10);Ok() A	1:00 Credit($10);Ok() A
	1:15 Commit A	1:15 Commit A
1:30 Credit($5);Ok() B	1:30 Credit($5);Ok() B	1:30 Credit($5);Ok() B
	1:45 Debit($5);Ok() C	1:45 Debit($5);Ok() C

If C had attempted to debit fifteen dollars, however, the front-end, unable to choose a response, would have released its short-term locks and waited for B to commit or abort.

5.1. Atomic Dependency

Consensus Scheduling is correct only if quorums are chosen properly. The essential requirement is that the view for an invocation must contain enough information not only to choose a "correct" response, but also to make scheduling decisions. This section describes a systematic method for deriving a correct and optimal set of constraints on quorum intersection directly from the data type's behavioral specification.

The definition of closure is slightly different for behavioral histories than for serial histories, because it is convenient to avoid constraining aborted actions. A subhistory G of H is a *closed* subhistory with respect to a relation $\succ$ if whenever G contains an event/action pair [e A] of H where A has not aborted, it also contains every earlier pair [e' A'] of H such that $e.inv \succ e'$, and A' has not aborted.

Constraints on quorum assignment are governed by an *atomic dependency* relation between invocations and events.

> **Definition 3:** Given a behavioral specification S, a relation $\succ$ between invocations and events is an atomic dependency relation for S if for all invocations *inv*, all responses *res*, all legal histories H, and all closed subhistories G containing all events e of H such that $inv \succ e$:
>
> $G \bullet$[inv;res A] is legal $\Rightarrow$ $H \bullet$[inv;res A] is legal.

Informally, this definition states that a correct response to an invocation can be chosen by observing any closed legal subhistory that includes the events on which the invocation depends.

A replicated atomic object implemented by Consensus Scheduling satisfies the behavioral specification S if and only if:

> The quorum intersection relation is an atomic dependency relation for S.

This claim is proved in the appendix. The similarity between serial and atomic dependency belies some interesting differences that are discussed in the next section.

5.2. Discussion

We show in the appendix that if T is a serial specification with a minimal serial dependency relation $\succ$, and if CL is the set of behavioral histories permitted by a Consensus Locking implementation of T using $\succ$ as the lock conflict/quorum intersection relation, then $\succ$ is a minimal atomic dependency relation for CL. Consensus Locking can thus be viewed as a specially optimized instance of Consensus Scheduling in which message traffic has been reduced by moving scheduling decisions from the front-ends to the repositories.

Let "hybrid dependency" denote a minimal atomic dependency relation for the full set of behavioral histories permitted by hybrid atomicity. Since any serial history is hybrid atomic, every hybrid dependency relation is a serial dependency relation. The converse, however, is false. For example, the minimal serial dependency relation for the *Account* data type is not a hybrid dependency relation. Consider the following hybrid atomic behavioral history, in which C has unsuccessfully attempted to withdraw $20, and B has deposited $10:

Debit($20);Overdrawn() C
Credit($10);Ok() B

Now suppose A attempts to credit $10 to the account. Recall that under the minimal serial dependency relation for the *Account* data type, *Credit* invocations do not depend on *Credit* events. The unsuccessful *Debit* event is thus a closed subhistory containing all events on which the *Credit* invocation depends. The result of appending the *Credit* event to the subhistory is hybrid atomic:

Debit($20);Overdrawn() C
Credit($10);Ok() A

But the result of appending the *Credit* to the entire history is not:

Debit($20);Overdrawn() C
Credit($10);Ok() B
Credit($10);Ok() A

because an illegal serialization results if the actions commit in the order A, B, and C.

This example illustrates that the enhanced concurrency provided by Consensus Scheduling may come at the cost of increased constraints on availability. For an *Account* replicated among *n* identical sites, Consensus Locking permits $\lceil n/2 \rceil$ distinct minimal quorum assignments, while a Consensus Scheduling implementation capable of realizing all hybrid atomic histories permits exactly one minimal quorum assignment: both *Credit* and *Debit* require a majority. A similar argument shows that

a Consensus Scheduling implementation capable of realizing all static atomic histories also permits only a single quorum assignment. Curiously, it can be shown that any quorum assignment that supports full static atomicity also supports full hybrid atomicity, but not vice-versa [18].

Consensus Scheduling may require more message traffic than Consensus Locking. Additional messages are needed to manage short-term locks, and to retry operations that could not be executed. The front-end may unable to execute an operation because the operation is partial (e.g. applying a partial *Deq* to an empty queue), or because the response depends on the outcome of a concurrent action (e.g. attempting to read a file that has been written by an uncommitted action). In either case, the front-end must release its short-term locks, wait for some duration, and try again. Message traffic can be reduced if repositories notify waiting front-ends when actions commit or abort.

6. Conclusion

This paper has introduced two new methods for managing highly concurrent replicated data in the presence of crashes and partitions. These methods are effective, general, and systematic. They are effective because they exploit type-specific properties of the data to provide better concurrency and more flexible availability than conventional methods that classify operations as reads or writes. They are general because they are applicable to objects of arbitrary type, and they are systematic because necessary and sufficient constraints on correct implementations are derived directly from the data type specification.

Consensus Locking is a simple and efficient replication method in which scheduling is based on predefined lock conflicts. Consensus Locking minimizes constraints on availability: no method can impose weaker constraints on quorum assignment. Consensus Locking combines two-phase locking and timestamping to provide better concurrency than other two-phase locking methods, even in the absence of replication. Nevertheless, because scheduling decisions are based exclusively on predefined conflicts between pairs of operations, Consensus Locking supports a suboptimal level of concurrency, and provides poor support for partial operations.

Consensus Scheduling is a more general method in which scheduling decisions may take the object's state into account. Consensus Scheduling can realize any level of concurrency realizable by a single-site implementation. It supports more concurrency than Consensus Locking, but at the potential cost of increased message traffic and additional constraints on quorum assignment. Consensus Locking can be viewed as an optimized special case of Consensus Scheduling.

Our analysis of Consensus Scheduling reveals a basic interdependence between the constraints

governing the availability and concurrency realizable by quorum consensus replication methods. This interdependence is fully characterized by the notion of atomic dependency: a Consensus Scheduling implementation will realize a behavioral specification S if and only if the quorum intersection relation is an atomic dependency relation for S. It is an unfortunate consequence of this result that availability and concurrency typically cannot be optimized within a single implementation: the more interleaving permitted by S, the more restrictive the associated atomic dependency relation.

We have seen that the conventional read/write classification of operations places unnecessary restrictions on both concurrency and availability. The contribution of this paper has been to show that integrated, type-specific methods can enhance both availability and concurrency, and to illuminate the interdependencies between these important properties.

Acknowledgements

I would like to thank Joshua Bloch, Daniel Duchamp, Barbara Liskov, and William Weihl for their comments on an earlier draft of this paper.

I. Formal Definitions and Proofs

This appendix presents formal definitions and proofs of the replication methods proposed in this paper. In the first section, we model Consensus Scheduling as a non-deterministic automaton that accepts certain behavioral histories. In the second section, we show that a Consensus Scheduling automaton accepts histories in S if and only if the quorum intersection relation is an atomic dependency relation for S. In the last section, we show that Consensus Locking can be treated as an optimized special case of Consensus Scheduling.

I.1. Consensus Scheduling

An *automaton* is a tuple $\langle Q, q_0, S, \delta \rangle$, where Q is a set of *states*, q_0 is the *initial state*, S is a set of *input symbols*, and $\delta \subseteq Q \times S \times Q$ is a *transition relation*. The transition relation can be extended to sets of states:

$$\delta(\emptyset, s_0) = \emptyset$$

$$\delta(X, s_0) = \bigcup_{q \in X} \delta(q, s_0)$$

and to sequence of input symbols:

$$\delta(X, \Lambda) = X$$

$$\delta(X, s \bullet s_0) = \delta(\delta(X, s), s_0)$$

Here Λ denotes the empty string. A string s is *accepted* by an automaton if $\delta(q_0, s) \neq \emptyset$.

A *log* L is a map from a finite set of timestamps to event/action pairs.

L: TIMESTAMP $\rightarrow$ EVENT $\times$ ACTION

A log M is a *sublog* of L if $M(t) = L(t)$ for every timestamp for which M is defined. Two logs L and M are *coherent* if they agree at every timestamp for which they are both defined. The *merge* operation $\cup$ is defined on pairs of coherent logs by:

$(L \cup M)(t)$ = if $L(t)$ is defined then $L(t)$
else $M(t)$.

Because the merge operation is defined only for coherent logs, it is commutative and associative.

Every log corresponds to a behavioral history in the obvious way. For brevity, we will sometimes refer to a log L when we really intend to refer to its corresponding behavioral history, e.g. "L is an element of the behavioral specification S." The exact meaning should be clear from context.

If X and Y are domains, (X $\rightarrow$ Y) denotes the set of partial maps from X to Y. Let REPOS be the domain of repositories. A *consensus scheduling automaton* is an automaton whose set of states is the Cartesian product of the following component sets:

Log: REPOS $\rightarrow$(TIMESTAMP $\rightarrow$ EVENT $\times$ ACTION)

Visited: ACTION $\rightarrow 2^{\text{REPOS}}$

Clock: TIMESTAMP

Committed: 2^{ACTION}

Aborted: 2^{ACTION}

The *Log* component associates a log (initially empty) with each repository. The *Visited* component associates with each action the set of repositories whose logs have been observed or updated by that action (initially none). The *Clock* component models a system of logical clocks, establishing an unambiguous ordering for events. The clock may have an arbitrary initial value. The sets *Committed* and *Aborted* keep track of the actions that have committed and aborted; each is initially empty.

The automaton's input symbols are event/action pairs. The automaton's transitions are governed by a *quorum assignment*: $\langle Initial, Final \rangle$. Let QUORUM be the domain of sets of repositories.

QUORUM = 2^{REPOS}

Initial is a function that associates each invocation with its initial quorums:

Initial: INV $\rightarrow 2^{\text{QUORUM}}$

and *Final* associates each event with its final quorums:

Final: EVENT $\rightarrow 2^{\text{QUORUM}}$.

The transition relation is defined for three kinds of events: operation executions, commits, and aborts. An operation execution [e A] is accepted only in states satisfying the following properties. First, the action must not have committed:

A $\notin$ Committed

Entries for aborted actions are always accepted and ignored. Henceforth, we assume A has not aborted. Second, there must exist a *view* constructed by merging the logs from an *initial quorum* IQ $\in$ *Initial*(e.inv):

$V = \cup_{R \in IQ} \text{Log}(R)$

such that the result of appending the new entry to V is legal:

$V \bullet [e\ A] \in S$.

If the event is accepted, the clock is advanced:

Clock' > *Clock*.

The new entry is appended to the view, and the updated view is merged with the log at each repository in a *final quorum* FQ $\in$ *Final*(e).

Log'(R)(t) = If R $\notin$ FQ then Log(R)(t)
elseif t = Clock' then [e A]
else (Log(R)$\cup V$)(t)

Each repository in the quorum is added to the action's set of visited sites.

Visited'(A) = Visited(A) $\cup$ IQ $\cup$ FQ

A *Commit* event for A is accepted only if the action has not already committed or aborted. When an action commits, the clock is advanced, a *Commit* entry is appended to the log at each repository visited by the action, and the action is added to the set of committed actions.

Clock' > Clock,

Log'(R)(t) = if R $\in$ Visited(A) and t = Clock' then [Commit A]
else Log(R)(t)

Committed' = Committed $\cup$ {A}

Similarly, an *Abort* event for A is accepted only if the action has not already committed. When an action aborts, the clock is advanced, abort entries are appended to the log at each repository visited by A, and the action is added to the set of aborted actions.

I.2. Correctness Arguments

Let $\succ$ denote the quorum intersection relation. We use the following technical lemmas.

Lemma 4: If $inv \succ e$ then all earlier entries for e appear in any view constructed for inv.

Lemma 5: The result of merging logs representing closed subhistories is a log representing a closed subhistory.

Lemma 6: If G is a closed subhistory of H, then it is a closed subhistory of $H \bullet [e\ A]$.

Note that the history accepted by an automaton is not necessarily the history reconstructed by merging the logs at all repositories, because events with empty final quorums appear at no repositories.

We now identify some invariant properties of consensus scheduling automata. Invariance is shown by induction on the length of the accepted history. Each property clearly holds in the initial state, and each property is clearly preserved when *Commit* or *Abort* events are accepted. Our arguments focus on showing that each property is preserved when an operation execution [e A] is accepted.

The first step is to show that the view for each invocation is a closed subhistory of the accepted history.

Lemma 7: The result of merging logs from any set of repositories is closed.

Proof: It suffices to show the property holds for any single repository; the more general result follows from Lemma 5. If a repository R is outside the final quorum for e, then $Log'(R) = Log(R)$, which remains closed (Lemma 6). Otherwise,

$$Log'(R) = (Log(R) \cup V) \bullet [e\ A]$$

The view V is:

$$V = \cup_{I \in IQ} Log(I)$$

Each $Log(I)$ is closed (induction hypothesis), hence V is closed (Lemma 5). $V \bullet [e\ A]$ is closed (Lemma 4), and $Log(R)$ is closed (induction hypothesis), therefore $Log'(R)$ is closed (Lemma 5).

Because the view for an invocation is the result of merging the logs from the repositories in an initial quorum:

Corollary 8: Each invocation's view is a closed subhistory of the accepted history.

The next step is to show that the view for each invocation is legal.

Lemma 9: If the quorum intersection relation is an atomic dependency relation for S, then the result of merging logs from any collection of repositories is legal.

Proof: Let U be an arbitrary set of repositories. and let $Log(U)$ and $Log'(U)$ be the results of merging the logs from the repositories in U respectively before and after a new event is

accepted. We show that if $Log(U)$ is legal, so is $Log'(U)$. If U does not intersect the final quorum for the new event e, then $Log(U) = Log'(U)$, and the result is immediate. Otherwise,

$$Log'(U) = (Log(U) \cup V) \bullet [e\ A]$$

where V is the view for e. Both V and $(V \cup Log(U))$ are closed (Corollary 8) and legal (induction hypothesis). V is a closed subhistory of $(Log(U) \cup V)$ that contains all events on which $e.inv$ depends (Lemma 4). Because $\succ$ is an atomic dependency relation for S, and $V \bullet [e\ A]$ is legal by construction, $(Log(U) \cup V) \bullet [e\ A]$ is legal.

This theorem reveals a fail-safety property of Consensus Scheduling: even if a catastrophic failure makes it permanently impossible to assemble a quorum for certain operations, the result of merging the surviving logs yields a legal subhistory of the true (lost) history.

Corollary 10: If the quorum intersection relation is an atomic dependency relation for S, then each invocation's view is legal.

We are now ready to present the basic correctness result:

Theorem 11: If the quorum intersection relation is an atomic dependency relation for S, then every history accepted by a Consensus Scheduling automaton is legal.

Proof: Let V be the view for e, and let H be the accepted history. V corresponds to a closed subhistory of H (Theorem 7), V is legal (Theorem 10), and it contains every event e' such that $e.inv \succ e'$ (Lemma 4). Because $\succ$ is an atomic dependency relation for S, and $V \bullet [e\ A]$ is legal, $H \bullet [e\ A]$ is also legal.

We now show that no set of constraints on quorum intersection weaker than atomic dependency guarantees that all behavioral histories accepted by a Consensus Scheduling automaton satisfy S.

Theorem 12: If the quorum intersection relation is not an atomic dependency relation, then the automaton will accept an illegal history.

Proof: Given a relation $\succ$ that does not satisfy Definition 3, we construct a Consensus Scheduling automaton whose quorum intersection relation satisfies $\succ$, and display a scenario in which it accepts an illegal history. If $\succ$ is not an atomic dependency relation for S, there exists an invocation *inv*, a response *res*, a legal history H having a closed subhistory G containing all events on which *inv* depends, such that:

$G \bullet [e\ A]$ is legal but $H \bullet [e\ A]$ is not.

We construct a consensus scheduling automaton that accepts the illegal history $H \bullet [e\ A]$ by first accepting the legal history H, and then choosing G as the view for *inv*. The automaton uses two repositories: R1 and R2. The automaton accepts the history H, choosing the following quorums for each event. For events in G, it chooses an initial quorum of R1 and a final quorum of both R1 and R2. For events in H but not in G, it chooses an initial quorum of R1 and R2 and a final quorum of R2. The view for each event in G thus contains all and only the prior events in G, and the view for every other event contains all prior events.

The intersection relation for these quorums must satisfy $\succ$, because all initial and final quorums intersect except the initial quorums for events in G and the final quorums for events not in G. If any of these quorums were required to intersect, then G would not be closed, contradicting the assumption. R1 is a valid initial quorum for *inv* because it intersects the final quorums for every event in G. Once the automaton has accepted H, it will then accept [e A], choosing G as its view. By assumption, $G \bullet$[e A] is legal but $H \bullet$[e A] is not.

I.3. Consensus Locking

In this section we show that Consensus Locking can be treated as a special case of Consensus Scheduling. We first characterize the set of behavioral histories permitted by Consensus Locking. We then apply two "optimizations" to transform a Consensus Scheduling automaton that accepts those histories into an equivalent automaton whose structure models Consensus Locking. We first show that initial and final locks can be used for scheduling, and then that front-ends can use serial histories instead of behavioral histories.

Let T be a serial specification for a data type, and let $\succ$ be a minimal serial dependency relation for T. If H is a behavioral history for T, event/action pairs [e A] and [e' A'] are said to be *concurrent* if A and A' are active and distinct. Let CL be the largest prefix-closed behavioral specification containing only hybrid atomic histories in which concurrent events are unrelated by $\succ$. Informally, CL is the set of behavioral specifications realizable by a Consensus Locking implementation of T having $\succ$ as its quorum intersection/lock conflict relation. CL is clearly on-line.

Theorem 13: $\succ$ is a minimal atomic dependency relation for CL.

Proof: Let e be an event, and G and H be histories in CL, such that G is a closed subhistory of H containing all events e' such that $e.inv \succ e'$. We first show that $\succ$ is an atomic dependency relation for CL by showing that if $G \bullet$[e A] is in CL, so is $H \bullet$[e A]. Let $h_1 \bullet e \bullet h_2$ be any serialization of H[e A] in *Commit* timestamp order, and let $g_1 \bullet e \bullet g_2$ be the corresponding serialization of $G \bullet$[e A]. Because $G \bullet$[e A] is in S, $g_1 \bullet e \bullet g_2$ and hence $g_1 \bullet e$ are legal. Because $\succ$ is a serial dependency relation, the legality of $g_1 \bullet e$ implies the legality of $h_1 \bullet e$. Because h_2 contains no events that depend on e, $h_1 \bullet e \bullet h_2$ is legal. If any smaller relation were an atomic dependency relation for CL, it would also be a serial dependency relation smaller than $\succ$, which we have assumed is minimal.

Initial and final locks are modeled by adding new components to the automaton's state.

I-Lock: REPOS $\rightarrow$(INV $\rightarrow 2^{\text{ACTION}}$)

F-Lock: REPOS $\rightarrow$(EVENT $\rightarrow 2^{\text{ACTION}}$)

For example, *I-Lock*(R)(inv) is the set of actions that hold initial locks for *inv* at R. Initially, no locks have been granted. When an operation execution [e A] is accepted, the action is granted an initial

lock for the invocation at each repository in the initial quorum, and a final lock for the event in each repository in the final quorum:

I-Lock'(R)(e.inv) = if R $\in$ IQ then I-Lock(R)(e.inv) $\cup$ {A}
else I-Lock(R)(e.inv)

F-Lock'(R)(e) = if R $\in$ FQ then F-Lock(R)(e) $\cup$ {A}
else F-Lock(R)(e)

An action's locks are released when it commits or aborts.

I-Lock'(R)(inv) = I-lock(R)(inv)-{A}

F-Lock'(R)(e) = F-lock(R)(e)-{A}

Note that the locks do not affect the histories accepted by the automaton; they simply track the automaton's state.

Theorem 14: If [e A] can be accepted, then no repository the initial quorum has granted a conflicting final lock, and no repository in the final quorum has granted a conflicting initial lock.

Proof: Otherwise there exists a concurrent event related to *e* by $\succ$.

This theorem implies that the set of histories accepted by an automaton is unaffected if initial and final locks are used for scheduling.

Theorem 15: An action B is *committed relative to* action A if B is committed, or if it is the same action as A. Let *H* be a behavioral history in CL, let *G* be the subhistory of events committed relative to A, and let *g* be the hybrid atomic serialization of *G*. *H*•[e A] is in CL if and only if (i) there is no concurrent event related to *e* by $\succ$, and (ii) *g*•*e* is in T.

Proof: The "only if" part is immediate from the definition of CI and because CI is on line. If *g*•*e* is in T, *G*•[e A] is in CL. But *G* is a closed subhistory of *H* containing the events on which *e.inv* depends, therefore if *G*•[e A] is in CL so is *H*•[e A].

This theorem implies that if initial and final locks are used for scheduling, then the set of histories accepted by the automaton is unaffected if front-ends choose responses using serial histories instead of behavioral histories.

References

[1] Alsberg, P. A., and Day, J. D.
A principle for resilient sharing of distributed resources.
In *Proceedings, 2nd Annual Conference on Software Engineering*. October, 1976.

[2] Bernstein, P., Goodman N., and Lai, M.-Y.
Two-part proof schema for database concurrency control.
In *Proc. Fifth Berkeley Workshop on Distributed Data Management and Computer networks*. February, 1981.

[3] Bernstein, P. A., and Goodman, N.
A survey of techniques for synchronization and recovery in decentralized computer systems.
ACM Computing Surveys 13(2):185-222, June, 1981.

[4] Bernstein, P. A., and Goodman, N.
The failure and recovery problem for replicated databases.
In *Proceedings, 2nd Annual Symposium on Principles of Distributed Computing*. August, 1983.

[5] Birman, K. P., Josepth, T. A., Raeuchle, T., and Abbadi A. E.
Implementing fault-tolerant distributed objects.
In *Proc. 4th Symposium on Reliability in Distributed Software and Database Systems*. October, 1984.

[6] Birrel, A. D., Levin, R., Needham, R., and Schroeder, M.
Grapevine: an exercise in distributed computing.
Communications of the ACM 25(14):260-274, April, 1982.

[7] Bloch, J. J., Daniels, D. S., and Spector, A. Z.
Weighted voting for directories: a comprehensive study.
Technical Report CMU-CS-84-114, Carnegie-Mellon University, April, 1984.

[8] Chan, A., Fox, S., Lin, W. T., Nori, A., and Ries, D.
The implementation of an integrated concurrency control and recovery scheme.
In *Proceedings of the 1982 SIGMOD Conference*. ACM SIGMOD, 1982.

[9] Dubourdieu D. J.
Implementation of distributed transactions.
In *Proceedings 1982 Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 81-94. 1982.

[10] Eager, D., L., and Sevcik, K. C.
Achieving robustness in distributed database systems.
ACM Transactions on Database Systems 8(3):354-381, September, 1983.

[11] Eswaran, K.P, Gray, J.N, Lorie, R.A., and Traiger, I.L.
The notion of consistency and predicate locks in a database system.
Communications ACM 19(11):624-633, November, 1976.

[12] Fischer, M., and Michael, A.
Sacrificing serializability to attain high availability of data in an unreliable network.
In *Proceedings, ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*. March, 1982.

[13] Gifford, D. K.
Weighted voting for replicated data.
In *Proceedings of the Seventh Symposium on Operating Systems Principles*. ACM SIGOPS, December, 1979.

[14] 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 Symp. on Principles of Database Systems*. March, 1983.

[15] Hammer, M. M., and Shipman D. W.
Reliability mechanisms in SDD-1, a system for distributed databases.
ACM Transactions on Database Systems 5(4):431-466, December, 1980.

[16] Herlihy, M. P.
Replication methods for abstract data types.
Technical Report MIT/LCS/TR-319, Massachusetts Institute of Technology Laboratory for Computer Science, May, 1984.
Ph.D. Thesis.

[17] Herlihy, M. P.
General quorum consensus: a replication method for abstract data types.
Technical Report CMU-CS-84-164, Carnegie-Mellon University, December, 1984.
Submitted for publication.

[18] Herlihy, M. P.
Comparing how atomicity mechanisms support replication.
1985.
Submitted for publication.

[19] Johnson, P. R., and Thomas, R. H.
The maintenance of duplicate databases.
Technical Report RFC 677 NIC 31507, Network Working Group, January, 1975.

[20] Kohler, W. H.
A survey of techniques for synchronization and recovery in decentralized computer systems.
ACM Computing Surveys 13(2):149-185, June, 1981.

[21] Korth, H. F.
Locking primitives in a database system.
Journal of the ACM 30(1), January, 1983.

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

[23] Liskov, B., and Snyder, A.
Exception handling in CLU.
IEEE Transactions on Software Engineering 5(6):546-558, November, 1979.

[24] Minoura, T., and Wiederhold, G.
Resilient extended true-copy token scheme for a distributed database system.
IEEE Transactions on Software Engineering 8(3):173-188, May, 1982.