CMU-CS-85-119

Using Type Information to Enhance the Availability of Partitioned Data

Maurice Herlihy
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213
8 April 1985

Abstract

A partition occurs when functioning sites in a distributed system are unable to communicate. This paper introduces a new method for managing replicated data in the presence of partitions. A novel aspect of this method is that it systematically exploits type-specific properties of the data to support better availability and concurrency than comparable methods in which operations are classified only as reads or writes. Each activity has an associated level, which governs how it is serialized with respect to other activities. Activities at the same level are serialized dynamically, but higher-level activities are serialized after lower-level activities. A replicated data item is a typed object that provides a set of operations to its clients. A quorum for an operation is any set of sites whose co-operation suffices to execute that operation, and a quorum assignment associates a set of quorums with each operation. Higher-level activities executing "in the future" may use different quorum assignments than lower-level activities executing "in the past." Following a failure, an activity that is unable to make progress using one quorum assignment may switch to another by restarting at a different level.

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.

Table of Contents

List of Figures

1. Introduction

A *distributed system* consists of multiple computers (called sites) that communicate through a network. Distributed systems are typically subject to two kinds of faults: site crashes and communication link failures. A crash renders a site's data temporarily or permanently inaccessible, while a communication link failure causes messages to be lost. 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.

A *distributed program* is one whose modules reside and execute at multiple sites in a distributed system. The data managed by a distributed program may 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 may be of little use if activities erroneously observe obsolete or inconsistent data. Consequently, replication is assumed to be *transparent*: its only observable effect is to make the data more available.

This paper introduces *layered consensus locking*, a new method for managing replicated data in the presence of crashes and communication link failures. A novel aspect of layered consensus locking is that it systematically exploits type-specific properties of the data to achieve better availability and concurrency than comparable methods based on the conventional read/write classification of operations. Each activity is an atomic action (or transaction). Each action has an associated *level*, which governs how it is serialized with respect to other actions. Actions at the same level are serialized dynamically, but higher-level actions are serialized after lower-level actions. A replicated data item is a typed object whose state can be altered and observed through a set of operations it provides to its clients. A *quorum* for an operation is a set of sites whose co-operation suffices to execute that operation, and a *quorum assignment* associates a set of quorums with each operation. Layered consensus locking exploits the observation that higher-level actions executing "in the future" may use different quorum assignments than lower-level actions executing "in the past."

Following a failure, an action that is unable to make progress using one quorum assignment may switch to another by restarting at a different level. Each action may choose the quorum assignment best suited to the partition in which it is executing. A complete characterization of realizable levels of availability and concurrency is derived directly from the data type specification.

Section 2 presents a brief survey of related work, and Section 3 describes our assumptions and terminology. Section 4 describes our technique for responding to failures, and Section 5 describes our technique for restoring normal operation when failures are repaired. Section 6 concludes with a discussion. A proof of correctness is given in the appendix.

2. Related Work

A useful survey of techniques for preserving consistency in partitioned networks appears in [7].

Early file replication methods were not transparent: the value read from a file is not necessarily the value most recently written [2, 19, 28]. Non-transparent replication methods for directories have also been proposed [23, 5, 11]. Replication methods transparent in the presence of site crashes but not partitions include [14, 15, 4].

Optimistic replication methods permit inconsistencies to develop during partitions, but these inconsistencies are detected and reconciled when communication is restored. Reconciliation methods may be *ad hoc*, as in Locus [25] or Data-patch [12], or systematic, as in proposals by Davidson [8] and Wright [31].

Pessimistic replication methods prevent inconsistencies from developing. Wright [31] has proposed a pessimistic scheme in which actions taken from a predefined set are classified by their read and write sets. A dependency analysis determines which classes can execute concurrently in distinct partitions. Unlike the scheme proposed here, Wright's scheme requires that each action's data dependencies be known in advance.

Replication methods based on quorum consensus [13, 6] are pessimistic; an operation invocation will succeed in any partition containing an appropriate quorum. The method proposed here is an extension of quorum consensus methods for arbitrary data types proposed by the author [16, 17, 18].

Replication methods proposed by Eager and Sevcik [9] and by Abbadi, Skeen, and Cristian [1] permit a file to be written in a majority partition while being read in other partitions. The principal limitation of these proposals is the lack of support for operations other than *Read* and *Write*. For example, the

Credit and *Debit* operations provided by a replicated bank account are both classified as writes, and thus both require a majority partition. By contrast, the method proposed here permits an account to be credited and debited concurrently in disjoint partitions. As discussed in Section 6, Eager and Sevcik's technique for responding to partitions can be viewed as a special case of our technique, but we propose an alternative method for restoring normal operation when partitions are rejoined.

3. Assumptions and Terminology

The basic units of computation are sequential processes called *actions*, or transactions. Actions are *atomic*, that is, serializable and recoverable. Serializability means that actions appear to execute in a serial order [24], and recoverability means that an action either succeeds completely, or has no effect. An action that completes all its changes successfully *commits*; otherwise it *aborts*, and any changes it has made are undone.

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 *Balance*, *Credit*, and *Debit* operations. *Balance* returns the current account balance:

Balance = Operation() Returns (Dollar)

Credit increments the account balance:

Credit = Operation(sum: Dollar).

Debit attempts to decrement the balance:

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

If the amount to be debited exceeds the account balance, the invocation signals an exception [21], leaving the account balance unchanged. For brevity, a *Debit* event that terminates normally is simply called a *debit*, otherwise it is called an *overdraft*.

3.1. Atomic Objects

With minor modifications, our model of atomic objects is that of Weihl [29, 30]. An object has 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 events ordered by a system of logical clocks [20].

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()
Balance();Ok($5)
```

is a serial history in which an account, initially empty, is credited $20, debited $15, and the balance is found to be $5. 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*, which is a sequence of operation executions, *Commit* events, and *Abort* events. To keep track of interleaving, each event is associated with an action. For example,

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

is a behavioral history in which action A and B each credit $5 to the account, 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 the object returned the responses, not necessarily the order in which it received the invocations.

A *behavioral specification* for an object is a set of *legal* behavioral histories for that object. Behavioral specifications are assumed to be 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.

3.2. Replicated Objects

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

A replicated object's state is represented as a *log*, which is a sequence of *entries*, each consisting of a logical timestamp, an event, and an action identifier. The log entries are partially replicated among the 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
1:01 Credit($1);Ok() A	1:01 Credit($1);Ok() A	
	1:02 Credit($2);Ok() B	1:02 Credit($2);Ok() B
1:03 Credit($3);Ok() C		1:03 Credit($3);Ok() C
1:04 Commit A	1:04 Commit A	
	1:05 Abort B	1:05 Abort B

This account has been credited three times by three actions, of which A has committed, B has aborted, and C is still active. Note that no single repository has an entry for all events.

We emphasize that logs represent a conceptual model for the replicated data, not a literal design for an implementation. More compact and efficient representations can be achieved by simple optimizations (such as discarding entries for aborted actions) and by type-specific optimizations (such as replacing a prefix of an account's committed entries with a single timestamped balance). Bloch, Daniels, and Spector have proposed a compact representation for replicated directories [6], and the author has proposed compaction techniques for other data types [16, 17]. Nevertheless, to avoid further digression, our examples use logs.

An operation is executed in three steps: the front-end reads information from an *initial quorum* for the invocation, performs a local computation to choose a response, and writes to a *final quorum* for the event. (Either the initial or final quorum may be empty.) A *quorum* for an event is any set of repositories that includes both an initial and a final quorum.

An operation execution is successful if the client is able to locate an available front-end for the object, and if the front-end is able to locate a quorum of available repositories. An action must be aborted if it is unable to complete an operation execution. If the failed operation is executed as a nested action [26, 22], the enclosing action need not be aborted. Because front-ends can be replicated to an arbitrary extent, perhaps placing one at each client's site, the availability of a replicated object is dominated by the availability of its repositories.

4. Layered Consensus Locking

This section describes a technique by which an action whose progress is blocked by failures may switch to a quorum assignment better suited to the currently available set of repositories. (Section 5 addresses the problem of restoring the normal quorum assignments when partitions are rejoined.)

4.1. Overview

Each action has an associated *level*, which is denoted by a natural number. Actions at the same level are serialized dynamically through lock conflicts, but level n actions are serialized before level $n+1$ actions. More precisely:

> **Definition 1:** A behavioral history is *layered hybrid atomic* if it is serializable in the following order: actions at distinct levels are ordered by level, and actions at the same level are ordered by their *Commit* timestamps.

An object is layered hybrid atomic if every history in its behavioral specification is layered hybrid atomic. Layered hybrid atomicity is an extension of *hybrid atomicity* [29], in which actions are serialized in the order induced by their *Commit* timestamps. Like hybrid atomicity, layered hybrid atomicity is *local*: if all objects in a system are layered hybrid atomic, then the system as a whole is atomic.

Layered hybrid atomicity is guaranteed by a two-part mechanism:

1. An action may be delayed to prevent synchronization conflicts with other active actions. For example, if action A at level 1 credits $10 to an empty account, and action C at level 2 attempts to debit $10, then C's response depends on A's outcome, and C is delayed until A commits or aborts.

2. An action may be aborted to prevent synchronization conflicts with committed actions. In the example above, once A and C have committed, an action B at level 1 that attempts to debit the account is aborted, because neither response (*Ok* or *Overdrawn*) permits B to be serialized between A and C. (If, instead, B were to credit the account, then it could proceed without violating atomicity.)

Synchronization conflicts between active actions are resolved by a modified form of *consensus locking* [18]. Every repository maintains an *initial lock* for each invocation and a *final lock* for each

event. Let *e.inv* denote the invocation part of an event *e*. To execute *e*, an action must acquire an initial lock for *e.inv* at each repository in its initial quorum and a final lock for *e* at each repository in its final quorum. Locks are granted on the basis of a predefined *lock conflict* relation between certain initial and final locks. An action that attempts to acquire an initial lock will be delayed if an action at the same or lower level holds a conflicting final lock. Similarly, an action that attempts to acquire a final lock will be delayed if an action at the same or higher level holds a conflicting initial lock. Initial and final locks are *strict two-phase*: an action holds its locks until it commits or aborts.

Synchronization conflicts between active and committed actions are resolved by *level locks*. Each repository maintains a level lock for each invocation. The level lock for *inv* is a counter that records the highest level for an action that committed at that repository while holding the initial lock for *inv*. When a level *n* action commits holding an initial lock for *inv*, the level lock for *inv* is set to the greater of its current value and *n*. The conflict relation for initial and final locks also governs conflicts between level locks and final locks. Once a repository's level lock for an invocation has been set to *n*, it will refuse all requests for conflicting final locks from actions at levels less than *n*.

Every replicated object provides a *sequence* of quorum assignments, one for each level. Level *n* actions use level *n* quorum assignments. In our examples, level 1 is used for normal unpartitioned activity, and higher level quorum assignments are used when failures occur. If an action is unable to locate a particular quorum within its partition, it may still be able to make progress by restarting at a higher level.

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 grants the action an initial lock for the invocation as soon as no other action holds a conflicting final lock. The repository sends its log back to the front-end.
- The front-end merges the logs from the initial quorum. The *view* is the serialization constructed by: (i) discarding all entries generated by aborted and higher-level actions, (ii) ordering actions by level, and (iii) ordering actions the same level by the timestamp order of their *Commit* entries, placing the client's own action last. A single-site serial implementation of the data type chooses a response from the view. The front-end generates a new timestamp, appends an entry for the event to the (unserialized) log, and sends the log to a final quorum of repositories.
- Each repository refuses the update if a level lock for a conflicting invocation has advanced beyond *n*. Otherwise, it grants the action a final lock for the new event as soon as no other action holds a conflicting initial lock. It 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.

Each action records its level at each repository before updating it for the first time. As part of its commit protocol [10, 27], an action reads the logical clocks at each repository visited, and sends back a *Commit* entry with a later timestamp. Each repository converts the action's initial locks to level locks and releases its initial and final locks. The *Commit* entry is recorded at each repository updated by the action. When an action aborts, initial and final locks are released, and an *Abort* entry is recorded at each repository updated by the action.

Layered consensus locking exploits the observation that higher-level actions executing "in the future" may use different quorum assignments than lower-level actions executing "in the past." Lock conflicts and quorum assignments must satisfy two essential requirements:

1. Each operation execution must encounter enough lock conflicts to preserve layered hybrid atomicity.

2. Each invocations's view must include enough entries to choose a correct response.

In the next section we give a precise definition of the constraints governing lock conflicts and quorum intersection.

4.2. Correctness Properties

We begin with some preliminary definitions. Let $\succ$ be a relation between invocations and events and let $h(i)$ denote the i-th event of the history h.

> **Definition 2:** A history g is a *closed subhistory* of h with respect to $\succ$ if there exists an injective order-preserving map s such that $g(i) = h(s(i))$ for all i in the domain of g, and if $e.inv \succ e'$, $j > j'$, $h(j) = e$, $h(j') = e'$, and $s(i) = j$, then there exists i' such that $s(i') = j'$.

Informally, if a closed subhistory contains an event e, it also contains every earlier event e' such that $e.inv \succ e'$.

Let $h \bullet$ [inv;res] denote the result of appending the event [inv;res] to the history h, and let S be a specification.

> **Definition 3:** A relation $\succ$ is a *serial dependency* relation for S if:
>
> $g \bullet$ [inv;res] $\in S \Rightarrow h \bullet$ [inv;res] $\in S$
>
> for all invocations *inv*, all responses *res*, all legal histories h, and all closed subhistories g that contain all events e of h such that $inv \succ e$.

Informally, a correct response to an invocation can be chosen by observing any closed legal subhistory that contains all the events on which the invocation serially depends. Of principal interest are *minimal* serial dependency relations, having the property that no smaller relation is a serial

dependency relation. A data type may have more than one minimal serial dependency relation [16, 17]. Some examples of serial dependency relations are given below.

In the appendix we show that every behavioral history generated by layered consensus locking is layered hybrid atomic if and only if the lock conflict and quorum intersection relations satisfy a common serial dependency relation. In other words, there must exist a serial dependency relation $\succ$ such that if $inv \succ e$, then:

- Initial and level locks for *inv* at level *n* conflict with final locks for *e* at levels less than or equal to *n*.
- Each initial quorum for *inv* at level *n* intersects each final quorum for *e* at levels less than or equal to *n*.

The key observation is that these constraints permit higher-level quorum assignments to have smaller final quorums and larger initial quorums than lower level quorum assignments.

	Read	Write
Level 1	(1,0)	(0,3)
Level 2	(2,0)	(0,2)
Level 3	(3,0)	(0,1)
.	.	.
.	.	.
.	.	.

Figure 4-1: Possible Quorum Assignments for a File

Perhaps the simplest example of a serial dependency relation is provided by the *File* type, for which *Read* invocations depend on *Write* events, but *Write* invocations do not depend on any prior events, because the response to *Write* is always *Ok*. A file replicated among three identical repositories might provide the range of quorum assignments shown in Figure 4-1, where (m,n) means that an event has initial quorums consisting of any *m* repositories and final quorums consisting of any *n* repositories. Levels three and higher are bound to the same quorum assignment. Actions in the majority partition can read and write the file (at level 2), while actions in a minority partition may either read the file but not write it (at level 1), or write the file but not read it (at levels 3 and higher).

For the *Account* data type, the correct response to a *Debit* invocation is *Ok* if the account balance covers the debit, and *Overdrawn* otherwise. Consequently, *Debit* invocations depend only on events that alter the account balance, i.e. credits and debits, but not overdrafts or balance inquiries. *Balance* invocations have the same dependencies as *Debit*. *Credit* invocations depend on no earlier events because the correct response to a *Credit* invocation is always *Ok*. An account replicated among

	Credit	Debit	Balance and Overdraft
Level 1	(0,3)	(1,3)	(1,0)
Level 2	(0,2)	(2,2)	(2,0)
Level 3	(0,1)	(3,1)	(3,0)
.	.	.	.
.	.	.	.
.	.	.	.

Figure 4-2: Possible Quorum Assignments for an Account

three identical repositories might provide the range of quorum assignments shown in Figure 4-2. An action executing in a majority partition may execute any operation at level 2, and an action executing in a minority partition may read the account balance at level 1, or credit the account at levels 3 and higher. Note that the quorum assignment at level 3 would not be permitted by a simple read/write classification of operations. The permissible quorum assignments for FIFO queues, priority queues, and stacks are similar; simply replace *Credit* by *Enq* or *Push*, *Debit* by *Deq* or *Pop*, and *Balance* by *Size*.

	Insert	Change	Lookup	Size
Level 1	(1,3)	(1,3)	(1,0)	(1,0)
Level 2	(1,3)	(1,2)	(2,0)	(1,0)
Level 3	(1,3)	(1,1)	(3,0)	(1,0)
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
Level 1	(1,3)	(1,3)	(1,0)	(1,0)
Level 2	(1,3)	(1,2)	(2,0)	(1,0)
Level 3	(2,2)	(2,2)	(2,0)	(2,0)
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

Figure 4-3: Possible Quorum Assignments for a Directory

A *Directory* stores pairs of values, where one value (the *key*) is used to retrieve the other (the *item*). The *Insert* operation inserts a new key/item pair in the directory, signaling an exception if the key is already present. The *Change* operation alters the item bound to the given key, signalling an exception if the key is absent. *Lookup* returns the item bound to the given key, signalling if the key is

absent, and *Size* returns the number of key-item pairs currently in the directory. *Insert* invocations depend on prior *Insert* events, and *Change* invocations depend on prior *Insert* events but not on prior *Change* events. *Lookup* depends on prior *Insert* and *Change* events, and *Size* depends only on prior *Insert* events. Two alternative ranges of quorum assignments for a *Directory* replicated among three identical repositories are shown in Figure 4-3.

4.3. A Replicated Account

To illustrate how this replication method enhances the availability of partitioned data, we trace a brief history for an *Account* replicated among three repositories using the quorum assignments shown above. The level locks for each repository are shown at the top of each column. (Level locks for *Credit* are omitted because initial *Credit* quorums are empty.) For brevity, initial and final locks are not shown. The account balance is initially zero. Action A records that it is executing at level 1, credits ten dollars, and commits.

R1	R2	R3
Debit: 1	Debit: 1	Debit: 1
Balance: 1	Balance: 1	Balance: 1
0:00 Level(1) A	0:00 Level(1) A	0:00 Level(1)
0:01 Credit($10);Ok() A	0:01 Credit($10);Ok() A	0:01 Credit($10);Ok() A
0:02 Commit A	0:02 Commit A	0:02 Commit A

The repositories are partitioned into a group containing R1 and a group containing R2 and R3. In the minority partition, action B discovers it cannot locate a level 1 final *Credit* quorum, restarts at level 3, and credits five dollars at R1.

R1
Debit: 1
Balance: 1

0:00 Level(1) A
0:01 Credit($10);Ok() A
0:02 Commit A
0:03 Level(3) B
0:04 Credit($5);Ok() B
0:05 Commit B

Meanwhile, in the other partition, action C discovers it cannot locate a level 1 initial *Debit* quorum, restarts at level 2, and debits ten dollars using R2 and R3 as its initial and final quorums.

R2	R3
Debit: 1	Debit: 1
Balance: 1	Balance: 1
0:00 Level(1) A	0:00 Level (1) A
0:01 Credit($10);Ok() A	0:01 Credit($10);Ok() A
0:02 Commit A	0:02 Commit A
0:06 Level(2) C	0:06 Level(2) C
0:07 Debit($10);Ok() C	0:07 Debit($10);Ok() C

After the partitions are rejoined, C executes a *Balance* with an initial quorum of R1 and R2, observing that the account is empty. C does not observe the five dollars credited by B because B is serialized "in the future." When C commits, the level locks are set to 2 for *Debit* at R2 and R3, and for *Balance* at R1 and R2.

R1	R2	R3
Debit: 1	Debit: 2	Debit: 2
Balance: 2	Balance: 2	Balance: 1
0:00 Level(1) A	0:00 Level (1) A	0:00 Level(1) A
0:01 Credit($10);Ok() A	0:01 Credit($10);Ok() A	0:01 Credit($10);Ok() A
0:02 Commit A	0:02 Commit A	0:02 Commit A
0:03 Level(3) B		
0:04 Credit($5);Ok() B		
0:05 Commit B		
	0:06 Level(2) C	0:06 Level(2) C
	0:07 Debit($10);Ok() C	0:07 Debit($10);Ok() C
	0:09 Commit C	0:09 Commit C

Finally, action D at level 3 chooses all three repositories as the initial quorum for a *Balance* invocation, observing that the account contains $5. When D commits, the level locks for *Balance* are set to 3.

R1	R2	R3
Debit: 1	Debit: 2	Debit: 2
Balance: 3	Balance: 3	Balance: 3

Henceforth, the level locks for *Balance* will prevent any action at levels 1 or 2 from debiting or crediting the account.

5. Restoring Normal Quorums

So far, layered consensus locking suffers from an important limitation: level locks set by actions restarting at higher levels will eventually block the use of lower-level quorum assignments. In the example above, once the account balance has been observed by a level 3 action, level lock conflicts will prevent later credits and debits by any action using lower-level quorums. Nevertheless, it may be desirable to continue using normal quorum assignments for objects unaffected by failures, and to

restore normal quorum assignments to other objects when failures are repaired. This section describes a protocol for restoring an object's normal quorum assignment when enough repositories are available.

Each object maintains a binding between level numbers and quorum assignments. Initially, actions execute at level 1, using each object's level 1 quorum assignment. Suppose a partition forces some actions to restart at level 2, setting level locks that block the use of the level 1 quorum assignment for a particular object. Eager and Sevcik have proposed a restoration technique in which that object's normal quorum assignment would be restored by retroactively converting the level 2 actions to level 1. Here, we propose an alternative technique. An object's normal quorum assignment is restored not by retroactively lowering the levels of the actions that set its level locks, but by altering the object's bindings between levels and quorum assignments. For example, by rebinding level 2 to the level 1 quorum assignment, later actions executing at level 2 will use the normal quorum assignment for that object and for all objects that have carried out a similar rebinding. The advantage of this technique is that each object's normal quorum assignment can be restored independently of the others. A comparison of our technique with that of Eager and Sevcik is given in Section 6.

Following a partition, actions that choose to restart at a higher level will set level locks that in turn force other actions to restart, eventually propagating the higher level through the system. When an action executing at the higher level encounters an object whose repositories are unaffected by the partition (or perhaps no longer affected), it may rebind the object's original quorum assignment to its own level. If the partition is repaired, and no additional failures occur, the system will stabilize in a state where all actions execute at the higher level, but using the original unpartitioned quorum assignments. In short, actions' levels are monotonically increasing, but quorum assignments are adjusted dynamically on a per-object basis.

5.1. Rebinding Quorum Assignments

Each object maintains a *quorum assignment table* binding each level to a quorum assignment. An object's quorum assignment table is replicated among its repositories and cached at its front-ends. An *initial coquorum* for an invocation is any set of repositories that intersects each of the invocation's initial quorums. *Final coquorums* and *coquorums* are defined similarly for events. The quorum assignment for level *n* is rebound in the following steps, which must be executed atomically.

- Merge the level *n* entries from an old final coquorum for each event.
- Write out the entries to a new initial coquorum for each invocation.
- Update the binding for level *n* at an old coquorum for each event.

The first step ensures that each level *n* entry appears in the merged log. The second step ensures that each level *n* entry appears at a new final quorum of repositories. The third step ensures that the caches maintained by the front-ends remain consistent, as described below. Note that restoration must preserve the constraints on quorum intersection; if levels 1, 2, and 3 are bound to distinct quorum assignments, then one cannot rebind level 3 to the quorum assignment for level 1 without also rebinding level 2. A single round of messages might be used to rebind multiple levels.

Each binding in an object's quorum assignment table has a timestamp which is updated each time the binding is changed. Whenever a front-end sends a message to a repository on behalf of a level *n* action, it includes its cached timestamp for level *n*'s binding. Whenever a repository receives a message with an out-of-date timestamp, it notifies the front-end of the new binding. Because the newer binding appears at a coquorum for every event, any level *n* quorum chosen by a front-end whose cache is out of date will include at least one repository whose binding is more current. When a front-end receives notification of the new binding, it updates its cache and retries the operation.

5.2. An Example

For brevity, we use the following abbreviations for the account's quorum assignments.

Abbreviation	Credit	Debit	Balance
Q1	(0,3)	(1,3)	(1,0)
Q2	(0,2)	(2,2)	(2,0)
Q3	(0,1)	(3,1)	(3,0)

In the following diagram, the quorum assignment table binds levels 1, 2, and 3, to Q1, Q2, and Q3, respectively, and each binding has timestamp *0:00*. Levels higher than 3 are implicitly bound to Q3.

R1	R2	R3
0:00 Q1	0:00 Q1	0:00 Q1
0:00 Q2	0:00 Q2	0:00 Q2
0:00 Q3	0:00 Q3	0:00 Q3
Debit: 1	Debit: 2	Debit: 2
Balance: 3	Balance: 3	Balance: 3
0:00 Level(1) A	0:00 Level (1) A	0:00 Level(1) A
0:01 Credit($10);Ok() A	0:01 Credit($10);Ok() A	0:01 Credit($10);Ok() A
0:02 Commit A	0:02 Commit A	0:02 Commit A
0:03 Level(3) B		
0:04 Credit($5);Ok() B		
0:05 Commit B		
	0:06 Level(2) C	0:06 Level(2) C
	0:07 Debit($10);Ok() C	0:07 Debit($10);Ok() C
	0:09 Commit C	0:09 Commit C

To rebind level 2 to Q1, the first step is to collect the level 2 entries from a final coquorum in Q2 for

both *Credit* and *Debit*, say R1 and R2. These entries are then written out to an initial coquorum in Q1 for both *Credit* and *Debit*, here consisting of all three repositories. The final step is to update the quorum assignment table at a coquorum in Q2 for each event, say R1 and R2.

R1	R2	R3
0:00 Q1	0:00 Q1	0:00 Q1
0:10 Q1	0:10 Q1	0:00 Q2
0:00 Q3	0:00 Q3	0:00 Q3
Debit: 1	Debit: 2	Debit: 2
Balance: 3	Balance: 3	Balance: 3
0:00 Level(1) A	0:00 Level(1) A	0:00 Level(1) A
0:01 Credit($10);Ok() A	0:01 Credit($10);Ok() A	0:01 Credit($10);Ok() A
0:02 Commit A	0:02 Commit A	0:02 Commit A
0:03 Level(3) B		
0:04 Credit($5);Ok() B		
0:05 Commit B		
0:06 Level(2) C	0:06 Level(2) C	0:06 Level(2) C
0:07 Debit($10);Ok() C	0:07 Debit($10);Ok() C	0:07 Debit($10);Ok() C
0:09 Commit C	0:09 Commit C	0:09 Commit C

Now suppose a level 2 action attempts to credit the account from a front-end whose cache is out of date. The front-end sends the *Credit* entry to two repositories along with the timestamp of its cached binding for level 2. At least one repository will detect that the cached timestamp is out of date, and respond with the up-to-date binding. The front-end will update its cache and retry the *Credit* using the newer quorum assignment.

Although restoration requires the co-operation of all three repositories in this example, fewer repositories are needed for other quorum assignments. For example, if an account is replicated among five identical repositories, a level can be rebound from Credit = (0,2), Debit = (4,2), Balance = (4,0) to Credit = (0,3), Debit = (3,3), Balance = (3,0) with the co-operation of only four out of five repositories.

6. Remarks

Layered consensus locking is general, systematic, and effective. It is general because it is applicable to objects of arbitrary type; it is systematic because constraints on correct implementations are derived directly from the specification of the data type in question; it is effective because it provides better availability, better concurrency, and fewer restarts than comparable methods based on the conventional read/write classification of operations.

6.1. Availability

In Eager and Sevcik's replication method [9], actions execute in one of two modes: normal or partitioned. In *normal* mode, actions read from any copy of a file, and write to all copies. In *partitioned* mode, actions use Gifford's quorum consensus method [13] to read and write a majority of copies. Each partitioned-mode action keeps track of the *missing writes* it was unable to apply to each copy of each file. Missing write information is propagated by having each action post its own missing write information at each site visited, and by merging the site's posted information with its own. When a normal-mode action encounters missing write information at a site, the action must either restart in partitioned mode, or it must carry out all the missing writes before proceeding. A complex protocol ensures that the missing writes are executed in the correct order, and that any other missing writes discovered during the protocol are themselves executed correctly. Once a missing write has been executed, it need no longer be posted.

The quorum assignments permitted by Eager and Sevcik's scheme satisfy the constraints imposed by layered consensus locking. In Figure 4-1, the level 1 quorum assignment corresponds to normal mode, and the level 2 assignment corresponds to partitioned mode. (By replacing version numbers with logical timestamps, Eager and Sevcik's method could be extended to permit the minority write quorums at level 3.) For objects other than files, however, layered consensus locking provides enhanced availability by systematically exploiting type-specific properties. For example, if *Credit* and *Debit* are simply classified as writes, then they cannot execute concurrently in distinct partitions. Similar remarks apply to the operations of the *Directory* data type.

6.2. Restoration

Although Eager and Sevcik's restoration method has the advantage that a normal-mode action that becomes aware of missing writes may still execute to completion if it is able to carry out the missing writes, it has the disadvantage that it imposes a high level of interdependence among distinct objects. A repository for one object may have to keep track of an arbitrary amount of missing write information for other objects. Propagation of missing write information may force an action to use partitioned-mode quorums even when all repositories for all objects it uses are available. Restoring normal quorums for one object may require visiting an arbitrary number of other sites, making it difficult or impossible to predict the amount of work required, or the likelihood that the necessary sites will be available. Finally, restoration may become progressively more difficult as missing write information propagates.

The principal advantage of the alternative restoration method proposed here is that each object manages its quorum assignments independently of the others. The normal quorum assignment can

be restored to one object without affecting the quorum assignments for other objects, making it possible to predict both the amount of work required, and the likelihood that the necessary repositories will be available. Repositories for one object do not keep track of information about other objects, and restoration does not become progressively more difficult.

6.3. Concurrency

Several recent proposals for replication methods treat concurrency control and replication independently [13, 9, 17, 1]. Layered consensus locking goes against this trend by integrating replication and concurrency control in a single mechanism. Although independent methods are simpler, integrated methods support more concurrency. In layered consensus locking, both the lock conflict and quorum intersection relations are governed by a common constraint: they must satisfy the same serial dependency relation. If two-phase read/write locks were used in place of initial and final locks, then the availability of replicated objects would be unaffected, but actions would encounter more delays and restarts because the concurrency control mechanism would be unable to exploit the same type-specific information used to enhance availability. For example, two-phase read/write locks unnecessarily prohibit concurrent credits, and missing write awareness unnecessarily requires a lower-level action to restart if it becomes "aware" of a credit by a higher level action (see actions B and C in the example above).

Although every behavioral history generated by layered consensus locking is layered hybrid atomic, the converse is false: there exist layered hybrid atomic behavioral histories that cannot be realized by layered consensus locking. For example, if action A credits $10 and commits, concurrent actions B and C could each debit $5 without violating layered hybrid atomicity. If, however, A had only credited $5, then B and C could not be allowed to debit the account concurrently. Because layered consensus locking (like most locking mechanisms) makes scheduling decisions exclusively on the basis of predefined conflicts between invocations and events, it cannot distinguish between these scenarios.

More concurrency can be achieved by taking more information into account. Unfortunately, no concurrency control mechanism residing entirely at the repositories can take full advantage of state information, because a replicated object's state can be ascertained only by merging the entries from an appropriate quorum of repositories. For example, a repository for a replicated account may be missing an arbitrary number of *Credit* and *Debit* entries, and hence no strictly local concurrency control mechanism can tell whether an account's committed balance permits concurrent debits.

Consensus scheduling [18] is an alternative concurrency control/replication method in which

scheduling decisions are made at front-ends using information collected from multiple repositories. Consensus scheduling can implement any behavioral specification, including the layered hybrid atomic specifications not realizable by layered consensus locking. This additional power comes at a cost: as discussed in [16, 18], consensus scheduling may require more message traffic, it may place additional constraints on quorum assignment, and it may require more complex local computations. Additional experience is needed to evaluate these trade-offs.

The notion of layering can be applied to properties other than hybrid atomicity. For example, *Static atomicity* [29, 30] is an alternative local atomicity property in which each action executes a *Begin* event when it starts executing. Actions must be serializable in the timestamp order of their *Begin* entries. A consensus locking mechanism satisfying static atomicity proposed in [16] can readily be modified to encompass multiple levels.

More generally, hybrid atomicity can be replaced by an arbitrary sequence of atomicity properties. Let $\{\mathcal{P}_i\}$ be a sequence of atomicity properties, not necessarily distinct. A behavioral history is *layered* $\mathcal{P}_i$*-atomic* if it is serializable in the following order: actions at distinct levels are ordered by level, and actions at level *i* are ordered by $\mathcal{P}_i$. Layered $\mathcal{P}_i$-atomicity is a local atomicity property if each $\mathcal{P}_i$ is local. Although consensus scheduling can realize any layered $\mathcal{P}_i$-atomic behavioral specification, more work is needed to evaluate the extent to which such generalizations are useful.

I. Formal Definitions and Proofs

This appendix presents a formal definition and proof of layered consensus locking. The first section gives a formal definition of layered consensus locking in terms of a non-deterministic automaton that accepts certain behavioral histories. The second section defines and proves correctness properties for such automata.

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:

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

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

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

I.1. The Layered Consensus Locking Automaton

We use the following primitive domains:

- ACTION is the set of actions,
- INV is the set of invocations,
- NAT is the set of natural numbers,
- REPOS is the set of repositories,
- RES is the set of responses,
- TIMESTAMP is the set of timestamps.

We use the following derived domains:

- EVENT = INV × RES,
- QUORUM = 2^{REPOS}.

If X and Y are domains, (X → Y) denotes the set of partial maps from X to Y. A *log* *L* is a map from a finite set of timestamps to event/action pairs.

L: TIMESTAMP → EVENT × ACTION

Two logs *L* and *M* are *coherent* if they agree at every timestamp for which they are both defined. The *merge* operation ∪ is defined on pairs of coherent logs by:

(*L* ∪ *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 sometimes refer to a log *L* in place of its behavioral history, e.g. "*L* is legal" instead of "the behavioral history represented by *L* is legal." The exact meaning should be clear from context.

A *layered consensus locking automaton* accepts behavioral histories. Its set of states is a subset of the Cartesian product of the following component sets:

- Log: REPOS → (TIMESTAMP → EVENT × ACTION)
- Clock: TIMESTAMP
- I-Lock: REPOS → (INV → 2^{ACTION})
- F-Lock: REPOS → (EVENT → 2^{ACTION})

- L-Lock: REPOS → (INV → NAT)
- Committed: 2^{ACTION}
- Aborted: 2^{ACTION}

The *Log* component associates a log (initially empty) with each repository. The *Clock* component models a system of logical clocks, establishing an unambiguous ordering for events. *I-Lock*, *F-Lock*, and *L-Lock* keep track of the initial locks, final locks, and level locks maintained at each repository. For example, *I-Lock(R)(inv)* is the set of actions that hold initial locks for *inv* at *R*. Initially, no initial or final locks have been granted, and all level locks are set to 1. The sets *Committed* and *Aborted* keep track of the actions that have committed and aborted; each is initially empty.

The automaton's transition relation is defined using the following sets.

- A serial specification *Spec* ∈ EVENT*.
- *Initial*: INV → 2^{QUORUM} assigns initial quorums to invocations.
- *Final*: EVENT → 2^{QUORUM} assigns final quorums to events.
- *Level*: ACTION → NAT assigns levels to actions.
- A lock conflict relation $\succ_L \subseteq$ INV × EVENT.
- A quorum intersection relation $\succ_Q \subseteq$ INV × EVENT (derived from *Initial* and *Final*).

The transition relation for each event is described in two parts: (i) the states in which the event is accepted, and (ii) the automaton's possible states after accepting the event. We now describe the transition relation for operation executions, commits, and aborts.

An operation execution [e A] is accepted only in states satisfying the following properties.

- A has not committed. Entries for aborted actions are accepted and ignored. Henceforth, we assume A is active.
- There exists an *initial quorum* IQ ∈ *Initial(e.inv)* such that no repository in IQ has granted a conflicting final lock to an action whose level is less than or equal to that of A.
- There exists a *final quorum* FQ ∈ *Final(e)* such that no repository in FQ has granted a conflicting initial lock to an action whose level is greater than or equal to that of A.
- No repository in the final quorum has a conflicting level lock whose value exceeds A's level.
- The event produce a legal serial history when appended to the serial history constructed by merging the logs from the initial quorum, serializing the committed actions first by

level, and then by the order of their commit events, discarding the events of actions whose levels exceed that of A, and appending the events of A.

Once the operation execution is accepted, the automaton undergoes a state transition.

- The clock is advanced.
- The action is granted an initial lock for the invocation at each repository in the initial quorum.
- The action is granted a final lock for the event at each repository in the final quorum.
- An entry for e is generated with the new clock value and appended to *Log(IQ)*. The updated log is merged with the log at each repository in the final quorum.

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 with the new timestamp is recorded at each repository R where A holds a final lock.
- The action is added to the set of committed actions.
- Level locks in the initial quorum are updated.
- All initial and final locks are released.

An *Abort* event for A is accepted only if the action has not already committed. When an action aborts, the clock is advanced, initial and final locks are released, abort entries are recorded at each repository where the action holds a final lock, and the action is added to the set of aborted actions.

I.2. Correctness Arguments

In this section we show that the behavioral histories accepted by a consensus locking automaton are layered hybrid atomic if and only if the lock conflict and quorum intersection relations satisfy a common serial dependency relation.

We start with a lemma about serial dependency.

Lemma 4: Let $\mathfrak{R}$ be a serial dependency relation, e an event, and h_1 and h_2 serial histories such that $h_1 \bullet h_2$ and $h_1 \bullet e$ are legal, h_1 contains no e' such that $e'.inv\ \mathfrak{R}\ e$, then $h_1 \bullet e \bullet h_2$ is legal.

Proof: By induction on the length of h_2. The result is immediate if h_2 is empty. Otherwise let $h_2 = h_2' \bullet e'$. The history $h_1 \bullet h_2'$ is a closed legal subhistory of $h_1 \bullet e \bullet h_2'$ containing all events e'' such that $e'.inv\ \mathfrak{R}\ e''$. Because $\mathfrak{R}$ is a serial dependency relation, the legality of $h_1 \bullet h_2' \bullet e'$ implies the legality of $h_1 \bullet e \bullet h_2' \bullet e'$. Thus $h_1 \bullet e \bullet h_2$.

Let $\gg$ denote the order used to define layered hybrid atomicity. If H is a behavioral history, and A an active action, *VIEW(H,A)* is the serial history constructed from H by serializing the committed actions in the order $\gg$, discarding the events of actions whose levels exceed that of A, and appending the events of A. An *outcome* of H is a serial history constructed by committing some set of active actions and serializing the result in the order $\gg$. If G is a subhistory of H, any outcome of H induces an outcome of G. Note that H is on-line layered hybrid atomic if and only if all its outcomes are legal serial histories.

> **Definition 5:** Let $\mathfrak{R}$ be a relation between invocations and events. A subhistory G of a behavioral history H is *view-closed* with respect to $\mathfrak{R}$ if *VIEW(G,A)* is a closed subhistory of every outcome of H.

We omit mention of $\mathfrak{R}$ and H when they are clear from context. The following lemma is an immediate consequence of our definitions:

> **Lemma 6:** The result of merging view-closed sublogs is a view-closed sublog.

Let $\succ$ denote $\succ_L \cap \succ_Q$, the intersection of the lock conflict and quorum intersection relations. Consider an automaton that has accepted a history $H \bullet$[e A], where e is an operation invocation, $h_1 \bullet e \bullet h_2$ is an outcome of $H \bullet$[e A], and $h_1 \bullet h_2$ is the induced outcome of H. As before, let *Log(IQ)* denote the merger of the logs from an initial quorum for *e.inv*.

> **Lemma 7:** If e' is an event in h_1 such that $e.inv \succ e$, then e' appears in *VIEW(Log(IQ),A)*.
>
> **Proof:** The intersection of the initial quorum for *e.inv* and the final quorum for e' is non-empty. The action that executed e' must have committed relative to A, otherwise some repository has granted conflicting locks.
>
> **Lemma 8:** There is no event e' in h_2 such that $e'.inv \succ e$.
>
> **Proof:** If e' exists, the intersection of the initial quorum for $e'.inv$ and the final quorum for e is non-empty. If B is the action that executed e', then A and B hold conflicting locks.

We now show some invariant properties of automata 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. We 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 view-closed subhistory of the accepted history.

> **Lemma 9:** Merging the logs from any set of repositories yields a view-closed subhistory of the accepted history.
>
> **Proof:** It suffices to show that the log at any single repository is view-closed; the more

general result follows from Lemma 6. If a repository R is outside the final quorum for e, then its log is unchanged. Otherwise, the new log is:

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

Log(IQ) is the merger of view-closed logs (induction hypothesis), and is therefore view-closed (Lemma 6). *Log(IQ)* • [e A] is view-closed (Lemma 7), *Log(R)* is view-closed (induction hypothesis), and therefore *Log'(R)* is view-closed (Lemma 6).

Corollary 10: *VIEW(Log(IQ),A)* is closed in any outcome of the accepted history.

The next step is to show that each invocation's view is a legal serial history.

Lemma 11: If $\succ$ is a serial dependency relation, then the result of merging logs from any collection of repositories is layered hybrid atomic.

Proof: Let S be a set of repositories, and let *Log(S)* and *Log'(S)* be the results of merging the logs from the repositories in S respectively before and after a new event is accepted. We show that if *Log(S)* is layered hybrid atomic, so is *Log'(S)*. If S does not intersect the final quorum for the new event e, then *Log(S)* = *Log'(S)*, and the result is immediate. Otherwise,

$$Log'(S) = (Log(S) \cup (Log(IQ) \bullet [e\ A])$$

Let $h_1 \bullet e \bullet h_2$ be any outcome of *Log'(S)*, and $h_1 \bullet h_2$ the induced outcome of *Log(S)*. *VIEW(Log(IQ),A)* is a closed subhistory of h_1 (Corollary 10), legal (induction hypothesis), and it contains each event e' of h_1 such that $e.inv \succ e'$ (Lemma 7). Because $\succ$ is a serial dependency relation and *VIEW(Log(IQ),A)* • e is legal, so is $h_1 \bullet e$. There is nc e' in h_2 such that $e'.inv \succ e$ (Lemma 8), hence $h_1 \bullet e \bullet h_2$ is legal (Lemma 4).

Corollary 12: *VIEW(Log(IQ),A)* is a legal serial history.

We remark that Theorem 11 does not quite prove the correctness of layered consensus locking because the history accepted by an automaton is not necessarily the history reconstructed by merging the logs at all repositories since events with empty final quorums appear at no repositories.

We are now ready to present the basic correctness result:

Theorem 13: The histories accepted by a layered consensus locking automaton are layered hybrid atomic if the intersection of the lock conflict and quorum intersection relations is a serial dependency relation.

Proof: Suppose the automaton accepts [e A] after accepting H. Let $h_1 \bullet e \bullet h_2$ be any outcome of H•[e A], and let $h_1 \bullet h_2$ be the induced outcome of H. H is layered hybrid atomic (induction hypothesis), thus h_1 is legal. *VIEW(Log(IQ),A)* is a closed subhistory of h_1 (Corollary 10), legal (Corollary 12), and contains every event e' such that $e.inv \succ e'$ (Lemma 7). Because $\succ$ is a serial dependency relation and *VIEW(Log(IQ),A)*•e is legal, so is $h_1 \bullet e$. There is no e' in h_2 such that $e'.inv \succ e$ (Lemma 8), hence $h_1 \bullet e \bullet h_2$ is legal (Lemma 4).

We now show the converse: if $\succ$ is not a serial dependency relation, there exists a quorum

assignment for which an automaton will accept a history that is not layered hybrid atomic.

We start with a lemma. Let $\mathfrak{R}$ be a relation between invocations and events that is *not* a serial dependency relation; *i.e.* there exist serial histories h and g and an event e such that g is a closed subhistory of h containing all events e' such that $e.inv\ \mathfrak{R}\ e$, $g \bullet e$ is legal, but $h \bullet e$ is illegal.

Lemma 14: There exist g, h, and e such that g is missing exactly one event of h.

Proof: Suppose g is missing k events of h. Consider the sequence of histories $\{h_i \mid i = 0,...,k\}$, where $h_0 = g$, $h_k = h$, and h_{i+1} is derived from h_i by restoring its earliest missing event.

If there exists an i such that h_i is legal but h_{i+1} is not, then h_i can be written as $g_0 \bullet g_1 \bullet e_2 \bullet g_2$, and h_{i+1} as $g_0 \bullet e_1 \bullet g_1 \bullet e_2 \bullet g_2$, where $g_0 \bullet e_1 \bullet g_1$ is legal, and $g_0 \bullet e_1 \bullet g_1 \bullet e_2$ is not. But $g_0 \bullet g_1$ is a closed legal subhistory of $g_0 \bullet e_1 \bullet g_1$ containing all events related to $e_2.inv$ by $\mathfrak{R}$, proving the lemma.

Otherwise, suppose h_i is legal for all i between 0 and k. Because $h_0 \bullet e$ is legal and $h_k \bullet e$ is not, there must exist an i such that $h_i \bullet e$ is legal but $h_{i+1} \bullet e$ is not. This h_i is a closed legal subhistory of h_{i+1} containing all events e' such that $e.inv\ \mathfrak{R}\ e'$, proving the lemma.

We now show that serial dependency is the weakest property that guarantees layered hybrid atomicity.

Theorem 15: Given a relation $\succ$ that is not a serial dependency relation, there exists an automaton that (i) has $\succ$ as the intersection of its lock conflict and quorum intersection relations, and (ii) accepts a history that is not layered hybrid atomic.

Proof: Given a quorum intersection relation $\succ_Q$ and a lock conflict relation $\succ_L$ such that $\succ = \succ_Q \cap \succ_L$ is not a serial dependency relation, choose h, g, and e to satisfy Lemma 14, where $h = h_1 \bullet e' \bullet h_2$ and $g = h_1 \bullet h_2$.

We construct an automaton with two repositories: R1 and R2. Action A at level 1 executes the events in h_1 and commits, choosing R1 and R2 as initial and final quorums for each event. B at level 2 executes e', choosing R1 as initial and final quorums. Action C at level 3 executes the events in h_2 followed by e, choosing R2 as the initial quorum for all invocations inv such that $inv \succ_L e'$, and both R1 and R2 for all other invocations. C chooses R1 and R2 as final quorums for all events.

These quorum choices must satisfy $\succ_Q$, because otherwise there would be an invocation inv in h_2 such that $inv \succ e'$, a contradiction. The entry for e' does not appear in any view for C, because B is active. C is not delayed, because the final lock held by B at R1 does not conflict with any initial locks acquired there by C. If B and C both commit, the accepted history has the illegal serialization $h_1 \bullet e' \bullet h_2 \bullet e$.

References

[1] Abbadi, A. E., Skeen, D., and Cristian, F.
An efficient, fault-tolerant protocol for replicated data management.
In *Proceedings, 4nd ACM SIGACT-SIGMOD Conf. on Principles of Database Systems*. 1985.

[2] 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.

[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] Birman, K. P., Joseph, 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.

[5] Birrel, A. D., Levin, R., Needham, R., and Schroeder, M.
Grapevine: an Exercise in Distributed Computing.
Communications of the ACM 25(14):260-274, Ap, 1982.

[6] 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.

[7] Davidson, S. B., Garcia-Molina, H., Skeen, D.
Consistency in a Partitioned Network: A Survey.
Technical Report TR 84-4, Dept. of Computer and Information Sciences, University of Pennsylvania, February, 1984.

[8] Davidson, S. B.
Optimism and consistency in partitioned distributed database systems.
ACM Transactions on Database Systems 9(3):456-482, September, 1984.

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

[10] 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.

[11] 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.

[12] Garcia-Molina, H., Allen, T, Blaustein, B., Chilenskas, R. M., Ries, D.
Data-Patch: Integrating Inconsistent Copies of a Database after a Partition.
In *Proceedings of the 3rd Symposium on Reliability in Distributed Software and Database Systems*. October, 1983.

[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.

[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.

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

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

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

[22] Moss, J. E. B.
Nested Transactions: An Approach to Reliable Distributed Computing.
Technical Report MIT/LCS/TR-260, Massachusetts Institute of Technology Laboratory for Computer Science, April, 1981.

[23] Oppen, D., Dalal, Y. K.
The clearinghouse: a decentralized agent for locating named objects in a distributed environment.
Technical Report OPD-T8103, Xerox Corporation, October, 1981.

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

[25] Popek, G. J., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G., and Thiel, G.
Locus: a network transparent high reliability distributed system.
In *Proceedings, Eighth Symposium on Operating Systems Principles*. December, 1981.

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

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

[28] Thomas, R. H.
A solution to the concurrency control problem for multiple copy databases.
In *Proc. 16th IEEE Comput. Soc. Int. Conf. (COMPCON)*. Spring, 1978.

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

[30] Weihl, W.
Specification and implementation of atomic data types.
Technical Report TR-314, Massachusetts Institute of Technology Laboratory for Computer Science, March, 1984.

[31] Wright, D., D.
Managing Distributed Databases in Partitioned Networks.
Technical Report 83-572, Cornell University, September, 1983.