Concurrency versus Availability: Atomicity Mechanisms for Replicated Data

MAURICE HERLIHY
Carnegie-Mellon University

A replicated object is a typed data object that is stored redundantly at multiple locations to enhance availability. Most techniques for managing replicated data have a two-level structure: At the higher level, a replica-control protocol reconstructs the object's state from its distributed components, and at the lower level, a standard concurrency-control protocol synchronizes accesses to the individual components. This paper explores an alternative approach to managing replicated data by presenting two replication methods in which concurrency control and replica management are handled by a single integrated protocol. These integrated protocols permit more concurrency than independent protocols, and they allow availability and concurrency to be traded off: Constraints on concurrency may be relaxed if constraints on availability are tightened, and vice versa. In general, constraints on concurrency and availability cannot be minimized simultaneously.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types*; *data types and structures*; D.4.3 [**Operating Systems**]: File Systems Management—*distributed file systems*; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance*; H.2.4 [**Database Management**]: Systems—*distributed systems*; *transaction processing*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Abstract data types, concurrency control, replication, synchronization

1. INTRODUCTION

A *replicated object* is a typed data object whose state is stored redundantly at multiple locations to enhance availability. A *replication method* is an algorithm for managing the object's distributed components so that its functional behavior is equivalent to that of a single-site object; this property is known as *one-copy serializability* [2]. A replication method must address two problems: concurrency control and replica management. A *concurrency-control* protocol ensures that incorrect behavior cannot occur as a result of concurrent access by multiple clients, and a *replica-management* protocol ensures that incorrect behavior cannot occur as a result of site crashes, network partitions, or timing anomalies. Many replication methods address these problems with independent mechanisms: At

This research was sponsored by the Defense Advanced Research Projects Agency (DoD), ARPA order no. 4976, monitored by the Air Force Avionics Laboratory under contract F33615-84-K-1520.
Author's address: Carnegie-Mellon University, Pittsburgh, Pa. 15213.

the lower level, a standard concurrency-control protocol synchronizes access to the individual components, and at the higher level, a replica-management protocol reconstructs the object's state from its distributed components without concern for concurrency.

In this paper, we explore an alternative way to manage replicated data. We propose two new replication methods in which concurrency control and replica management are handled by a single integrated protocol. These integrated protocols permit more concurrency than independent protocols, and they allow availability and concurrency to be traded off: Constraints on concurrency may be relaxed by tightening constraints on availability, and vice versa. We give a complete formal characterization of this interdependence, presenting examples where the constraints on concurrency and availability cannot be minimized simultaneously. These protocols also exploit type information to impose fewer constraints on concurrency and availability than techniques that rely on the conventional read/write classification of operations.

Section 2 presents a brief overview of related work, and Section 3 presents our model of computation. In Section 4, we introduce *consensus locking*, a replication method in which concurrency control is based on predefined lock conflicts. Consensus locking minimizes constraints on availability but not on concurrency. In Section 5, we introduce *consensus scheduling*, a more general protocol in which concurrency control may use arbitrary state information. Consensus scheduling can realize more concurrency than consensus locking, but this additional concurrency may incur a cost in reduced availability. We conclude with a discussion in Section 6. Formal models and correctness arguments are given in the Appendix.

2. RELATED WORK

Early file replication methods did not attempt to preserve one-copy serializability; the value read from a file is not necessarily the value most recently written [1, 22, 32]. Nonserializable replication methods for directories have also been proposed [6, 13, 29].

In the *available copies* replication method [3], 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 [16], ISIS [5], and Circus [8]. Unlike the methods proposed in this paper, these methods do not preserve one-copy serializability in the presence of network partitions.

In the *true-copy token* scheme [27], 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.

The earliest use of quorum consensus is a file replication method due to Gifford [14]. 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 [18]. 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 [9], El-Abbadi et al. [10], and for arbitrary data types by this author [20].

3. ASSUMPTIONS AND TERMINOLOGY

Distributed systems are subject to two kinds of faults: Sites may crash and communication links may be interrupted. When a site crashes, its resident data become 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.

A widely accepted technique for preserving consistency in the presence of failures and concurrency is to organize computations as sequential processes called *transactions*. Transactions are *atomic*, that is, serializable and recoverable. *Serializability* [30] means the execution of one transaction never appears to overlap (or contain) the execution of another, and *recoverability* means that a transaction either succeeds completely or has no effect. A transaction's effects become permanent when it *commits*, its effects are discarded if it *absorts*, and a transaction that has neither committed nor aborted is *active*. A standard commitment protocol (e.g., [15, 31]) aborts transactions interrupted by failures, and stable storage [26] ensures that the effects of committed transactions are not undone by later failures.

Our model for atomic objects borrows from that of Weihl [33]. 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 for creating and manipulating objects of that type. For example, a (FIFO) queue might be represented by an object of type *Queue* providing *Enq* and *Deq* operations.

```
Enq = Operation(Item)
Deq = Operation( ) Returns(Item)
```

Enq places an item at the end of the queue, and Deq returns the item at the head of the queue. Deq is *partial*: It is undefined when the queue is empty. A transaction invoking a partial operation is blocked until the operation can return a response.

In the absence of failures and concurrency, a computation is modeled as a *history*, which is a finite sequence of operations. Histories are denoted by

lowercase letters (g, h). An operation is written as $\langle x$ op(args*)/term(res*)$\rangle$, where x is an object name, *op* is an operation name, *args** denotes a sequence of argument values, *term* is a termination condition, and *res** is a sequence of results. The operation name and argument values constitute the *invocation*, and the termination condition and result values constitute the *response*. We use "Ok" for normal termination. The object name is omitted when it is clear from the context. For example,

Enq(a)/Ok( )
Enq(b)/Ok( )
Deq( )/Ok(a)

is a Queue history in which items a and b are enqueued and a is dequeued.

An *object subhistory*, $h \mid x$ (h at x), is the subsequence of operations in h whose object names are x. Each object has a *serial specification*, which defines a set of *legal* histories for that object. For example, the specification of a Queue includes all and only histories in which items are enqueued and dequeued in FIFO order. Specifications are prefix closed: Any prefix of a legal history is legal. A history h involving multiple objects is *legal* if each object subhistory $h \mid x$ belongs to the serial specification for x.

In the presence of concurrency and failures, an object's behavior is modeled by a *schedule*, which is a sequence of *steps* of the form $\langle x\ q\ Q\rangle$, $\langle x$ Commit $Q\rangle$, or $\langle x$ Abort $Q\rangle$, where x is an object, q an operation, and Q a transaction name. The object name is omitted when it is clear from the context. Schedules are denoted by uppercase letters (G, H). If H is a schedule, x an object, Q a transaction, and S a set of transactions, define $H \mid x$, $H \mid Q$, and $H \mid S$ by analogy with object subhistories.

A schedule is *well formed* if transaction names are unique, no transaction executes an operation after it commits, and no transaction both commits and aborts. All schedules are assumed to be well formed. For example, the following is a well-formed schedule for a Queue:

Enq(a)/Ok( ) P
Enq(b)/Ok( ) Q
Commit P
Deq( )/Ok(a) Q
Commit Q

Here, transaction P enqueues a, Q enqueues b, P commits, and Q dequeues a and commits. The ordering of operation executions in a schedule reflects the order in which the responses are returned, not necessarily the order in which the invocations occurred.

Each object has a *concurrent specification*, which defines a set of *legal* schedules for that object. All concurrent specifications are assumed to be prefix closed and *on-line*: An active transaction can choose to commit or abort at any time.

Serial and concurrent specifications are related by the notion of *atomicity*. Let $\ll$ denote a total order on committed and active transactions, and let H be a schedule. The *serialization* of H in the order $\ll$ is the history h constructed by reordering the operations in H, so that for all transactions P and Q, if $P \ll Q$,

the subsequence of operations associated with P precedes the subsequence associated with Q. H is *serializable in the order* $\ll$ if h is a legal history; H is *serializable* if it is serializable in some order; and H is *atomic* if the subschedule associated with committed transactions is serializable. An object is atomic if every schedule in its concurrent specification is atomic. All objects considered in this paper are atomic.

4. CONSENSUS LOCKING

4.1 Introduction

Consensus locking is the first of two integrated replication methods presented in this paper. Consensus locking ensures atomicity through predefined *conflicts* between pairs of operations. Conflict-based synchronization uses only some of the information available for scheduling: It does not consider interactions among more than two operations, and it does not take the object's state into account. Nevertheless, it is likely to be adequate for many applications, and it can be implemented efficiently by locking mechanisms.

4.2 The Protocol

Replicated objects are implemented by two kinds of sites: *repositories* and *front ends*. Repositories provide long-term storage for the object's state, and front ends carry out operations for clients. A client applies an operation to a replicated object by sending an invocation to one of the object's front ends. The front end reads data from some collection of repositories, carries out a local computation, sends updates to some collection of repositories, and returns the response to the client. The client must 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 assignment* associates each invocation with a set of valid initial quorums and each operation with a set of valid final quorums. An object's quorum assignment determines the availability of its operations; thus the constraints governing quorum assignment are the basic constraints governing the availability realizable by a replication method.

Quorum assignments are constrained by a *quorum intersection relation*: Certain initial and final quorums are required to have nonempty intersections. For example, any quorum assignment for a replicated file must ensure that each initial Read quorum intersects each final Write quorum; otherwise it would be impossible to guarantee that each value read is the value most recently written. If two operations are related by the quorum intersection relation, then their levels of availability can be traded off: If one operation's quorums are made

R1	R2	R3
1:00 Enq(*a*)/Ok( )	1:00 Enq(*a*)/Ok( )	
	1:15 Enq(*b*)/Ok( )	1:15 Enq(*b*)/Ok( )
1:30 Enq(*c*)/Ok( )		1:30 Enq(*c*)/Ok( )

Fig. 1. Three items have been enqueued, although no repository has an entry for all three.

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

At each repository, the operations of committed transactions are recorded in a *log*, which is a sequence of *entries*, where an entry is the timestamped record of an operation. For example, Figure 1 is a schematic representation of a Queue replicated among three repositories. For readability, a missing entry is marked by a blank.

Each log entry has a two-part timestamp:

(1) The low-order bits are occupied by an *operation field* that defines how this operation is serialized relative to other operations executed by the same transaction. This field is filled in when the operation is executed.

(2) The high-order bits are occupied by a *transaction field* that defines how the transaction is serialized relative to other transactions. This field is filled in when the transaction commits.

In our examples, a timestamp is represented as a single-digit transaction field separated by a colon from a two-digit operation field, and new timestamp values are generated sequentially. In practice, each field would be much longer, and successive values would be separated by arbitrary gaps.

Each active transaction's operations are recorded in an *intentions log*, one per transaction, also partially replicated among the repositories. When a transaction appends an entry to its intentions log, the operation field is filled in, but the transaction field is left empty. When the transaction commits, the coordinator for the commitment protocol [15, 31] generates a logical timestamp, which it distributes to the participating repositories [25]. When the protocol is complete, each repository inserts that timestamp in the transaction fields of the committing transaction's entries and merges its intentions log into the committed log.

To model the timestamp assignment protocol, we extend our notation as follows. Each commit step includes a timestamp argument: $\langle \text{Commit}(t)\ Q \rangle$, where t is the timestamp assigned to transaction Q. Timestamps are governed by an additional well-formedness constraint: If Q executes an operation after P commits, then Q's commit timestamp must be later than P's.

For concurrency control, consensus locking employs a form of strict two-phase locking [4, 12, 24]. Each repository maintains an *initial lock* for each invocation and a *final lock* for each operation. A repository grants a transaction an initial lock when the repository agrees to participate in an initial quorum for that invocation, and it grants a final lock when it agrees to participate in a final quorum for that operation. Certain initial and final locks *conflict*: A repository will not grant an initial (final) lock while a conflicting final (initial) lock is held

by a different transaction. Informally, the initial locks ensure that the operation cannot be invalidated by a concurrent operation, and the final locks ensure that the operation cannot invalidate any concurrent operations. Each transaction holds its locks until it commits or aborts. Like most locking protocols, consensus locking is subject to deadlock, which can be handled by standard techniques such as timeouts or deadlock detection [23].

An operation is executed in the following steps:

(1) The client sends the invocation and transaction identifier to a front end, which consults an internal table of initial and final quorums and forwards the client's request to an initial quorum for the invocation.
(2) Each repository in the initial quorum checks whether another transaction holds a conflicting final lock. If so, the call is delayed until the conflicting locks are released. When the initial lock is granted, the repository sends its committed log and the client transaction's intentions log to the front end.
(3) The front end merges the logs from the initial quorum by using the timestamps to discard duplicate entries. It constructs a history called the *view* by appending the client transaction's intentions to the history of committed operations, and it chooses a response consistent with the view. The front end generates an entry for the new operation, filling in the operation field by reading its local logical clock, and sends the view and the new entry to a final quorum of repositories.
(4) Each repository in the final quorum checks whether another transaction holds a conflicting initial lock. If so, the call is delayed until the conflicting locks are released. When the final lock is granted, the repository merges the logs and returns an acknowledgment to the front end.
(5) The front end returns the response to the client when a final quorum has acknowledged the update.

To illustrate this protocol, we trace a brief history for a Queue replicated among three repositories. In the example shown in Figures 2–4 any two out of three repositories constitute an initial quorum for Deq and a final quorum for both Enq and Deq. Initial Deq locks conflict with final Enq and Deq locks, but initial and final Enq locks do not conflict. Initially, the queue is empty. Transaction B enqueues item c at R1 and R3, and transaction A enqueues item a at R1 and R2 and item b at R2 and R3. Both transactions proceed without interference. Locks are shown at the top of each column.

Transaction C now attempts a Deq operation by choosing R1 and R2 as its initial quorum. Since initial Deq locks conflict with final Enq locks, C is delayed. Eventually, A commits with timestamp 1, B commits with timestamp 2, and each repository fills in its transaction fields and merges the intentions logs with the log of committed operations. C acquires initial Deq locks at R2 and R3 and constructs a view by merging their logs, thereby revealing that a is the first item in the queue. C appends a Deq entry to its view and sends its log to R2 and R3 (shown in Figure 4). Eventually, C commits with timestamp 3.

The principal benefit of a replication model based on partially replicated logs is generality: The same model can be used to analyze the availability and

R1	R2	R3
Final(Enq) = {*A*, *B*}	Final(Enq) = {*A*}	Final(Enq) = {*A*, *B*}
B:00 Enq(*c*)/Ok( )		*B*:00 Enq(*c*)/Ok( )
A:00 Enq(*a*)/Ok( )	*A*:00 Enq(*a*)/Ok( )	
	A:01 Enq(*b*)/Ok( )	*A*:01 Enq(*b*)/Ok( )

Fig. 2. "*A*:00" denotes a time stamp with an empty transaction field (to be filled when *A* commits) and an operation field of 0.

R1	R2	R3
1:00 Enq(*a*)/Ok( )	1:00 Enq(*a*)/Ok( )	
	1:01 Enq(*b*)/Ok( )	1:01 Enq(*b*)/Ok( )
2:00 Enq(*c*)/Ok( )		2:00 Enq(*c*)/Ok( )

Figure 3

R1	R2	R3
	Initial(Deq) = {C}	Initial(Deq) = {*C*}
	Final(Deq) = {*C*}	Final(Deq) = {*C*}
1:00 Enq(*a*)/Ok( )	1:00 Enq(*a*)/Ok( )	1:00 Enq(*a*)/Ok( )
	1:01 Enq(*b*)/Ok( )	1:01 Enq(*b*)/Ok( )
2:00 Enq(*c*)/Ok( )	2:00 Enq(*c*)/Ok( )	2:00 Enq(*c*)/Ok( )
	C:00 Deq( )/Ok(*a*)	*C*:00 Deq( )/Ok(*a*)

Figure 4

concurrency of queues, stacks, directories, or any other data type of interest. We can reason abstractly about the fundamental constraints governing availability and concurrency without having to consider type-specific techniques for data representation. Nevertheless, it is worth emphasizing that this protocol is an idealized model for replication, not a literal design for an implementation. In practice, logs are too large and inefficient to be practical. Instead, an implementation must address the problem of *log compaction*, that is, replacing a log with a more compact and efficient data structure.

For example, Bloch et al. [7] give a type-specific compaction technique for replicated directories. A complementary approach is the following: Whenever a repository's committed log includes a prefix of the object's history, that prefix can be replaced by a timestamped *version*, which is a compact single-site representation for the object. For example, a Queue version could be a list or array of items present in the queue. A repository might take active measures to assemble such a prefix, such as running periodic "garbage-collection" transactions, or it might respond to prefixes that arise naturally in the course of executing certain operations.

In the example shown in Figure 5, when *C* commits with timestamp 3, R2 and R3 may replace their committed logs with a single timestamped version, since the view assembled for the Deq included all earlier Enq and Deq entries, and

R1	R2	R3
1:00 Enq(*a*)/Ok( )		
2:00 Enq(*c*)/Ok( )		
	3:00 [*b*, *c*]	3:00 [*b*, *c*]

Figure 5

the lock conflicts ensure that no active transaction will be serialized before *C*. R1 cannot compact its entries, however, because it may be missing Enq entries whose timestamps lie between 1:00 and 2:00.

4.3 Correctness

As mentioned above, each transaction is issued a logical timestamp when it commits. A consensus-locking implementation is correct if transactions are serializable in the order of their commit timestamps, a property known as *hybrid atomicity* [33]. Weihl has shown that hybrid atomicity is a *local* property: If H is a schedule such that $H \mid x$ is hybrid atomic for all objects x, then H is atomic. Consensus locking is thus compatible with other protocols that guarantee hybrid atomicity, including the two-phase locking protocols cited above.

In this section we characterize the constraints on concurrency and availability necessary to ensure correctness. Let *inv*(p) denote the invocation part of operation p. Constraints on quorum assignment and lock conflict for consensus locking are expressed in terms of the following relations between operations:

—The *quorum intersection* relation $\succ_Q$: $\text{inv}(q) \succ_Q p \equiv$ each initial quorum for inv(q) intersects each final quorum for p.

—The *lock conflict* relation $\succ_L$: $\text{inv}(q) \succ_L p \equiv$ initial locks for inv(q) conflict with final locks for p.

It is easy to see that $\succ_L$ and $\succ_Q$ cannot be chosen independently. For example, to ensure that two operations never execute concurrently, it is not enough for their locks to conflict; their quorums must also intersect to ensure that the conflict is detected at some repository. We define the *effective conflict* relation to be the intersection of the lock conflict and quorum intersection relations:

$$\text{inv}(q) \succ_{LQ} p \equiv \text{inv}(q) \succ_L p \wedge \text{inv}(q) \succ_Q p$$

In this section, we characterize the constraints on $\succ_Q$, $\succ_L$, and $\succ_{LQ}$ that ensure the correctness of consensus locking.

Let $\succ$ be an arbitrary relation between invocations and operations.

Definition 1. A subhistory g of h is $\succ$-*closed* if, whenever g contains an operation q of h it also contains every earlier operation p such that $\text{inv}(q) \succ p$.

Definition 2. A subhistory g of h is a $\succ$-*view* of h for inv(q) if g is $\succ$-closed and if it includes every p in h such that $\text{inv}(q) \succ p$.

Fig. 6. First serial dependency relation for Queue.

	Enq/Ok	Deq/Ok
Enq		
Deq	X	X

	Enq/Ok	Deq/Ok
Enq	X	
Deq		X

Fig. 7. Second serial dependency relation for Queue.

Informally, $\succ$ is a *serial dependency* relation if, whenever an operation is legal for a $\succ$-view, it is legal for the complete history. More precisely, let "•" denote concatenation:

Definition 3. A relation $\succ$ is a *serial dependency* relation if, for all operations p and all legal histories g and h such that g is a $\succ$-view of h for $\text{inv}(p)$, $g \bullet p$ is legal implies that $h \bullet p$ is legal.

Of primary interest are *minimal* relations having the property that no smaller relation is also a serial dependency relation. In Appendix A, we show that consensus locking ensures hybrid atomicity if the effective confliction relation $\succ_{LQ}$ is a serial dependency relation, and also that no weaker constraint ensures correctness.

4.4 Examples

The Queue data type has two distinct minimal serial dependency relations, shown in Figures 6 and 7. Each relation permits interleavings and quorum assignments not permitted by the other. In Figure 6, Deq invocations depend on both Enq and Deq operations, thereby implying that initial Deq quorums must intersect final Enq and Deq quorums, and that initial Deq locks must conflict with final Enq and Deq locks. Initial and final Enq quorums need not intersect, and their locks need not conflict. As a practical matter, it may sometimes be useful to introduce additional lock conflicts not strictly required for correctness. For example, if initial Deq locks conflict, then it would not be possible for concurrent Deq operations to deadlock by attempting to dequeue the same item.

In Figure 7, Enq invocations depend on earlier Enq operations, and Deq invocations depend on earlier Deq operations, but Deq invocations do not depend on Enq operations, and vice versa. To see why this relation is a serial dependency relation, observe that if the view assembled for a Deq includes an Enq entry for an undequeued item, then because the view is closed it also includes the entry for the earliest undequeued item. A Deq invocation may block unnecessarily if it fails to observe any enqueued items, but it will never return an incorrect item.[1] Here, Enq quorums must intersect, and Deq quorums must intersect, but Enq quorums need not intersect Deq quorums. An enqueuing transaction can execute concurrently with a dequeuing transaction as long as the queue contains items enqueued by committed transactions.

[1] As discussed elsewhere [18], this relation would not be a serial dependency relation if a Deq applied to an empty queue were to signal an exception instead of blocking.

	Ins/Ok	Rem/Ok
Ins		
Rem		X

Fig. 8. Serial dependency relation for SemiQueue.

The queue example illustrates why constraints on quorum assignment and lock conflict cannot be specified independently: A replicated queue would not be one-copy serializable if, for example, quorum intersection were governed by the relation in Figure 6 and lock conflicts by the relation in Figure 7, because the effective conflict relation would not be a serial dependency relation.

Constraints on concurrency and availability can often be relaxed by introducing nondeterminism into data type specifications. A *Semiqueue* provides the following *Ins* and *Rem* operations:

```
Ins = Operation(Item)
Rem = Operation( ) Returns(Item)
```

Ins inserts an item in the Semiqueue, and Rem nondeterministically removes and returns an item from the Semiqueue. Like Deq, Rem returns only when there is an item to remove. There may be an additional probabilistic guarantee (not captured by our functional specifications) that the item removed is likely to be the oldest one. The Semiqueue data type has a unique minimal serial dependency relation shown in Figure 8. To ensure that no item is removed twice, initial and final Rem quorums must intersect and Rem operations cannot execute concurrently. Ins operations may execute concurrently with Rem operations and with one another. If a Semiqueue is intended to approximate a FIFO Queue, each front end could remove the oldest item in its view. To increase the likelihood that a Rem operation will choose older items, a background process could propagate Ins entries among the repositories, effectively causing their final quorums to grow asynchronously.

The notion of serial dependency can be extended to hold between pairs of operations instead of between operations and invocations. Such an extension can provide a slight increase in concurrency and availability at the cost of a less efficient implementation. For example, an *Account* provides the following *Credit* and *Debit* operations:

```
Credit = Operation(Dollar)
Debit = Operation(Dollar) Signals (No)
```

Credit increments the account balance by a specified amount, and Debit attempts to decrement the balance. If the balance cannot cover the debit, the operation returns with an exception, leaving the balance unchanged. If we allow serial dependency to take the invocation's results into account, then successful debits may be treated differently from attempted overdrafts, as illustrated by the relation shown in Figure 9. Here, transactions executing successful debits cannot execute concurrently, but a successful debit can execute concurrently with any number of overdrafts.

Nevertheless, it may be difficult to exploit this additional concurrency in practice. How can a front end executing a Debit predict whether it should acquire

	Credit/Ok	Debit/Ok	Debit/No
Credit/Ok			
Debit/Ok	*X*	*X*	
Debit/No	*X*		

Fig. 9. Serial dependency relation for Account.

locks for a successful debit or an overdraft? An optimistic strategy is to "guess" that the debit will succeed by requesting locks for a successful debit. If the balance fails to cover the debit, the front end releases its locks and restarts the operation (but not the entire transaction), "guessing" this time that the debit will fail. A pessimistic strategy is to acquire locks for both operations, which is equivalent to an initial lock on the invocation, but to release the superfluous lock when the response becomes known. (Similar considerations arise if an operation's choice of initial quorums may depend on its anticipated result.) Either approach may require additional message traffic, and it remains unclear whether such refinements are cost-effective for replicated data.

4.5 Remarks

By exploiting type information, consensus locking places fewer constraints on quorum assignment than replication methods based on the conventional read/write classification of operations [14]. Each operation of the example data types reviewed above would be classified as a read followed by a write, since each modifies the object state in a way that depends on the operations' arguments and the object's current state. Since read and write quorums must intersect, all quorums for all operations must intersect, which rules out many of the quorum assignments permitted by consensus locking.

Consensus locking is a generalization of *general quorum consensus* [18], a type-specific replication method that relies on an underlying concurrency control protocol to ensure atomicity. Consensus locking permits more concurrency because it exploits information about the data type specification and about the quorum assignment. For example, if transactions synchronize through two-phase read/write locks at repositories [12], it would be impossible to distinguish between real synchronization conflicts (e.g., Enq/Deq conflicts in Figure 6 or Enq/Enq conflicts in Figure 7) and spurious conflicts (e.g., Enq/Enq conflicts in Figure 6 or Enq/Deq conflicts in Figure 7).

The two methods place identical constraints on quorum assignment: The quorum intersection relation must be a serial dependency relation. Clearly, no integrated replication method can permit more quorum assignments than general quorum consensus, simply because any quorum assignment that ensures one-copy serializability in the presence of concurrency must also work if transactions appear to execute serially. Nevertheless, because consensus locking is no worse than the general quorum consensus, it places the weakest possible constraints on quorum assignments. By contrast, the next section describes an integrated mechanism that permits more concurrency than consensus locking, but fewer quorum assignments.

Consensus locking compares favorably to several locking protocols in the literature, even in the absence of replication. Korth [24] and Bernstein et al. [4] have proposed type-specific two-phase locking protocols for single-site objects

whose operations are total and deterministic. In both protocols operations that do not commute have conflicting locks. (Informally, two invocations commute if they can occur in either order and both orders yield the same final state.) It can be shown that failure to commute is a serial dependency relation, but not necessarily a minimal relation. Consequently, consensus locking can realize any level of concurrency permitted by commutativity-based locking schemes, but not vice versa. For example, Enq operations do not commute, but Enq locks need not conflict (Figure 6).

The notion of serial dependency also arises in a variety of conflict-based synchronization mechanisms, including optimistic techniques [19], multiversion timestamping techniques [21], and techniques in which quorum assignments change dynamically in response to failures [20]. Although we do not address the issue here, consensus locking can be extended to nested transaction systems by introducing lock inheritance rules similar to those proposed by Moss [28].

5. CONSENSUS SCHEDULING

5.1 Introduction

Consensus locking has two limitations: It makes scheduling decisions exclusively on the basis of pair-wise lock conflicts, and it cannot be used in systems based on local atomicity properties other than hybrid atomicity. This section introduces *consensus scheduling*, a generalization of consensus locking that can be used to construct replicated objects satisfying arbitrary concurrent specifications. This additional power comes at a cost: Relaxing constraints on concurrency may require strengthening constraints on quorum assignment. In general, constraints on availability and concurrency cannot be minimized simultaneously.

The limitations of conflict-based scheduling can be illustrated by the following example in Figure 10. Committed transaction A has credited \$10 to a replicated account in two separate \$5 credits at different final quorums. Transaction B has tentatively credited another \$5. If transaction C attempts to debit \$10, it will be delayed by conflicts with B's Credit locks. This delay is unnecessary, however, because the account balance covers the debit, regardless of the order in which B and C commit. If, instead, C had attempted to debit \$15, it would indeed have to wait until B commits or aborts, because B's outcome will determine whether the balance covers the debit. Consensus locking cannot distinguish between these scenarios, and therefore C must be delayed in both cases.

An inability to take full advantage of state information for scheduling is a characteristic of any concurrency control scheme implemented at the individual repositories. In the example above, no single repository has enough information to recognize that the committed balance 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 unobserved Credit and Debit entries recorded elsewhere.

5.2 The Protocol

The basic idea behind consensus scheduling is the following: Instead of using partially replicated logs to represent serialized histories, we use the logs to represent concurrent schedules, explicitly representing information about

R1	R2	R3
Final(Credit) = {*B*}	Final(Credit) = {*B*}	
	1:00 Credit($5)/Ok( )	1:00 Credit($5)/Ok( )
1:01 Credit($5)/Ok( )		1:01 Credit($5)/Ok( )
B:00 Credit($5)/Ok( )	*B*:00 Credit($5)/Ok( )	

Figure 10

R1	R2	R3
	0:00 Credit($5)/Ok( ) *A*	0:00 Credit($5)/Ok( ) *A*
0:01 Credit($5)/Ok( ) *A*		0:01 Credit($5)/Ok( ) *A*
	0:02 Commit(2) *A*	0:02 Commit(2) *A*
0:03 Credit($5)/Ok( ) *B*	0:03 Credit($5)/Ok( ) *B*	

Figure 11

interleavings, commit and abort steps, and steps of active transactions. Scheduling decisions are made at front ends, rather than repositories, using information merged from an initial quorum. Timestamps need no longer be split into transaction fields and operation fields, since commit orderings can be reconstructed by inspecting the commit steps directly.

For example, the replicated Account in Figure 10 might be represented in Figure 11. The absence of a transaction field is indicated by a leading zero in each timestamp. All the information used by the consensus-locking protocol (such as commit timestamps and locks) can be reconstructed from this schedule.

An operation is executed in the following steps:

(1) The client sends the invocation and transaction identifier to a front end, which forwards them to an initial quorum of repositories.
(2) Each repository in the initial quorum responds by sending its entire log to the front end.
(3) The front end merges the logs from the initial quorum to construct a *view*. Note that the view is a schedule, not a history. It chooses a response consistent with the object's state as reconstructed from the view. The front end generates an entry for the new operation, appends it to the view, and sends the view to a final quorum of repositories. If the front end cannot choose a response, perhaps because it must await the outcome of another transaction, it waits for the object's state to change and restarts the protocol.
(4) Each repository in the final quorum merges the view with its log and returns an acknowledgment to the front end.
(5) As soon as the front end receives acknowledgments from a final quorum, the response is returned to the client.

When a transaction commits or aborts, an appropriate entry is inserted in the log of each repository visited by that transaction.

Some form of short-term synchronization is needed to ensure that the steps of the protocol are executed atomically. One solution is to place at each repository

R1	R2	R3
	0:00 Credit($5)/Ok( ) *A*	0:00 Credit($5)/Ok( ) *A*
0:01 Credit($5)/Ok( ) *A*	0:01 Credit($5)/Ok( ) *A*	0:01 Credit($5)/Ok( ) *A*
	0:02 Commit(2) *A*	0:02 Commit(2) *A*
0:03 Credit($5)/Ok( ) *B*	0:03 Credit($5)/Ok( ) *B*	0:03 Credit($5)/Ok( ) *B*
	0:04 Debit($10)/Ok( ) *C*	0:04 Debit($10)/Ok( ) *C*

Figure 12

a short-term mutual exclusion lock. The front end acquires a short-term lock at each repository in its initial quorum in step 1 and at its final quorum at step 3. The front end releases its short-term locks in step 4. Repositories may break short-term locks in response to suspected deadlocks, but the interrupted operation must be restarted. Returning to our example in Figure 10, let us trace how transaction *C* could debit $10 from the replicated account. 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. Merging these logs yields the following view:

0:00 Credit($5)/Ok( ) *A*
0:01 Credit($5)/Ok( ) *A*
0:02 Commit(2) *A*
0:03 Credit($5)/Ok( ) *B*

The view indicates that the $10 deposited by *A* will cover the debit, regardless of *B*'s outcome, so the front end appends the entries for the new Debit operation to its view and sends the view 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 shown in Figure 12.

If *C* had attempted to debit $15 dollars, however, the front end, unable to choose a response, could have released its short-term locks and waited for *B* to commit or abort.

5.3 Constraints on Availability

Let $\succ$ be an arbitrary relation between invocations and operations. We extend the notion of $\succ$-closed and $\succ$-view to schedules.

Definition 4. *G* is a $\succ$-*closed* subschedule of *H* if, whenever *G* contains an operation step $\langle q\ Q\rangle$ of *H* it also contains every preceding operation step $\langle p\ P\rangle$, such that $\text{inv}(q) \succ p$ and neither *P* nor *Q* is aborted.

Definition 5. *G* is a $\succ$-*view* of *H* for inv(*q*) if *G* is $\succ$-closed and if it includes every $\langle p\ P\rangle$ in *H*, such that $\text{inv}(q) \succ p$, and *P* is not aborted.

These definitions are intended to avoid constraining the behavior of aborted transactions.

Let *Concur* be an arbitrary concurrent specification.

Definition 6. Let G be a $\succ$-view of H for inv(q). $\succ$ is an *atomic dependency* relation for *Concur* if $G \cdot \langle q\ Q \rangle$ is in *Concur* implies $H \cdot \langle q\ Q \rangle$ is in *Concur*.

In Appendix A, we show that a replicated atomic object implemented by consensus scheduling satisfies the concurrent specification Concur if and only if the quorum intersection relation is an atomic dependency relation for Concur.

The formal similarity between serial and atomic dependency belies some interesting differences. If T is a serial specification, let Hybrid(T) be the concurrent specification consisting of all hybrid atomic schedules for T. Let $\succ_H$ be a minimal atomic dependency relation for Hybrid(T). Clearly, $\succ_H$ must be a serial dependency relation, since it must ensure hybrid atomicity even when all operations are executed serially by a single transaction. It follows that every quorum assignment permitted by a consensus-scheduling implementation of Hybrid(T) is permitted by a consensus-locking implementation of T. The converse, however, is false. For example, let $\succ_S$ be the minimal serial dependency relation for the Account data type shown in Figure 9. Let H be the following hybrid atomic schedule:

$$\begin{array}{l}\text{Debit(\$20)/No( )}\ C \\ \text{Credit(\$10)/Ok( )}\ B,\end{array}$$

and let G be the subschedule consisting of the first step. Recall that Credit operations are unrelated by $\succ_S$; thus G is a $\succ_S$-view of H for Credit. Now suppose A attempts to credit \$10 to the account. $G \cdot \langle$Credit(\$10)/Ok( ) $A\rangle$ is hybrid atomic, but $H \cdot \langle$Credit(\$10)/Ok( ) $A\rangle$ is not, because an illegal serialization results if the transactions commit in the order A, B, and C. It follows that a consensus-scheduling implementation of Hybrid(Account) requires initial and final Credit quorums to intersect, whereas consensus-locking implementation of Account, which permits less concurrency, does not.

For an account replicated among n identical repositories, consensus locking permits $\lceil n/2 \rceil$ distinct minimal quorum assignments: Credit and Debit quorums must respectively encompass m and $n - m + 1$ repositories, where m ranges between 1 and $\lceil n/2 \rceil$. By contrast, a consensus-scheduling implementation of Hybrid(Account) permits exactly one minimal quorum assignment: Both Credit and Debit require a majority. A similar argument shows that if Static(Account) is the full set of static atomic schedules, a consensus-scheduling implementation of Static(Account) also permits only a single quorum assignment. Curiously, it can be shown that for any data type T, any quorum assignment permitted by Static(T) is also permitted by Hybrid(T), but not vice versa [17].

5.4 Remarks

Consensus scheduling may require more message traffic than consensus locking. Under consensus locking, any operation that is not partial can be executed in a fixed number of messages. Under consensus scheduling, however, additional messages may be needed to manage short-term locks and to retry operations blocked by synchronization conflicts. Consensus locking can 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, and by synchronizing operations in two phases instead of one.

Although we do not address the issue here, the consensus-scheduling algorithm and the notion of atomic dependency can be extended to a nested transaction model by introducing additional well-formedness conditions.

6. CONCLUSION

Replication methods in which concurrency control and replica management are handled by distinct protocols at different levels have an appealing simplicity. Each protocol can be designed and understood in isolation, and a variety of replication methods can be constructed by combining different techniques. Nevertheless, we have argued here that combining concurrency control and replica management in a single protocol permits greater concurrency, as well as more flexibility in trading concurrency against availability.

Consensus locking is a replication method that relies on predefined lock conflicts derived from the object's type specification and its quorum assignment. Consensus locking minimizes constraints on availability and concurrency: No replication method, integrated or not, can impose weaker constraints on quorum assignment, and no conflict-based concurrency control protocol permits more operations to execute concurrently. Consensus locking also supports more concurrency than a number of locking schemes in the literature, even in the absence of replication. Nevertheless, the concurrency realizable by consensus locking is limited by an inability to take the object's state into account.

Consensus scheduling is a generalization of consensus locking that can be used to construct replicated objects satisfying arbitrary concurrent specifications, including specifications that satisfy local atomicity properties other than hybrid atomicity and specifications in which scheduling decisions take the object's state into account. This additional power may come at the cost of additional message traffic and tighter constraints on quorum assignment.

Availability and concurrency are not independent properties: The availability and concurrency realizable by replication are governed by a common set of constraints, formally characterized by the notion of an atomic dependency relation. In particular, there exist data types for which the constraints on availability and concurrency cannot be minimized simultaneously: The more interleaving permitted by the concurrent specification, the more restrictive the constraints on quorum assignment.

APPENDIX A FORMAL DEFINITIONS AND PROOFS

A.1 Consensus-Locking Automata

For consensus locking, a replicated object is modeled by a nondeterministic *consensus-locking automaton* that accepts certain schedules. The automaton's state is defined using the following primitive domains: REPOS is the set of repositories, TRANS the set of transaction identifiers, INV the set of invocations, RES the set of responses, and TIMESTAMP a totally ordered set of timestamps. $X \rightarrow Y$ denotes the set of partial maps from X to Y. We use the following derived domains: OP = INV × RES is the set of operations, QUORUM = 2^{REPOS} the set of quorums, and a *log* is a map from a finite set of timestamps to operations: LOG = TIMESTAMP → OP.

Two logs L and M are *coherent* if they agree for every timestamp where they are both defined. The *merge* operation $\cup$ is defined on pairs of coherent logs by

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

Every log corresponds to a history in the obvious way. For brevity, we sometimes refer to a log L in place of its history, that is, "L is legal" instead of "the history represented by L is legal."

A consensus-locking automaton has the following state components:

Intent: REPOS $\rightarrow$ (TRANS $\rightarrow$ LOG)
I-Lock: REPOS $\rightarrow$ (INV $\rightarrow$ 2^{TRANS})
F-Lock: REPOS $\rightarrow$ (OP $\rightarrow$ 2^{TRANS})
Clock: TIMESTAMP
Committed: TRANS $\rightarrow$ TIMESTAMP

Let R be a repository and Q a transaction. *Intent*(R, Q) is the log of entries for Q recorded at R, initially none. *I-Lock*(R, inv) and *F-Lock*(R, q) are, respectively, the sets of transactions holding initial and final locks for inv and q at R, initially none. *Clock* models a system of logical clocks, and *Committed* maps committed transactions to their timestamps.

If S is a set of repositories, and Q a transaction, let Intent$(S, Q) = \cup_{R \in S}$ Intent(R, Q), and similarly for I-Lock(S, inv) and F-Lock(S, q). Let $\{P_1, \ldots, P_k\}$ be the sequence of committed transactions sorted in the order of their commit timestamps. Define

$$Perm(S) = \text{Intent}(S, P_1) \bullet \cdots \bullet \text{Intent}(S, P_k),$$

as the history constructed by appending the intentions in timestamp order. If Q is an active transaction, define

$$\text{View}(S, Q) = \text{Perm}(S) \bullet \text{Intent}(S, Q),$$

as the history constructed by serializing committed transactions in timestamp order, followed by Q.

The automaton's transaction relation is defined using the following sets:

—A serial specification *Serial.*
—A *lock-conflict* relation $\succ_L \subseteq \text{INV} \times \text{OP}$.
—Initial: INV $\rightarrow$ 2^{QUORUM} assigns initial quorums to invocations.
—Final: OP $\rightarrow$ 2^{QUORUM} assigns final quorums to operations.

Initial, Final, and $\succ_L$ induce the relations $\succ_Q$ and $\succ_{LQ}$ defined above.

For brevity, we assume that all input schedules are well formed. (Well-formedness could be checked explicitly by adding more state components and preconditions.)

The precondition for accepting an operation step $\langle q\ Q \rangle$ is the following. There must exist IQ in Initial(inv(q)) and FQ in Final(q) such that

If inv$(q) \succ_L p$, then F-Lock$(IQ, p) - \{Q\} = \emptyset$,
If inv$' \succ_L q$, then I-Lock$(FQ, \text{inv}') - \{Q\} = \emptyset$, and
View$(IQ, Q) \bullet q$ is in Serial.

No repository in the initial quorum has granted a conflicting final lock, no repository in the final quorum has granted a conflicting initial lock, and appending the operation to the transaction's view yields a legal history.

Let View′ be the result of appending the new step to the view:

$$\text{View}'(S, Q)(t) = \text{if } (t = \text{Clock}) \text{ then } q \text{ else View}(S, Q)(t)$$

Following the operation execution, the automaton's new state satisfies the following postconditions, where x' denotes the new value of component x:

Clock′ > Clock
For all R in IQ, I-Lock′$(R, \text{inv}(q))$ = I-Lock$(R, \text{inv}(q)) \cup \{Q\}$.
For all R in FQ, F-Lock′(R, q) = F-Lock$(R, q) \cup \{Q\}$.
For all R in FQ and P in TRANS, Intent′(R, P)
= Intent$(R, P) \cup$ View′$(IQ, Q) \mid P$.

The clock is advanced, the transaction is granted locks, and the view is merged with the logs in the final quorum.

The precondition for accepting a commit step $\langle \text{Commit}(t)Q \rangle$ is *true*. After accepting the commit step, the following postcondition holds:

Clock′ > max(Clock, t),
Committed′(P) = if $(P = Q)$ then t else Committed(P),
For all inv in INV and R in REPOS,
I-Lock′(R, inv) = I-Lock$(R, \text{inv}) - \{Q\}$, and
For all p in OP and R in REPOS, F-Lock′(R, p) = F-Lock$(R, p) - \{Q\}$.

The commit advances the clock beyond the commit time, the transaction is marked as committed, and its locks are released. The precondition for an abort step is true, and the postcondition advances the clock and releases locks.

A.2 Correctness Arguments

A schedule is *hybrid atomic* [33] if transactions are serializable in commit timestamp order. A schedule is *on-line* hybrid atomic if the result of committing any set of active transactions yields a hybrid atomic schedule. Henceforth, we say simply "hybrid atomic" for "on-line hybrid atomic." A *hybrid serialization* of H is a history constructed by committing some set of active transactions and serializing them in commit timestamp order. A schedule is hybrid atomic if all its hybrid serializations are legal. We will show that the schedules accepted by a consensus-locking automaton are hybrid atomic if the effective conflict relation is a serial dependency relation, and that no weaker constraints on quorum intersection and lock conflict preserve hybrid atomicity.

We use the following technical lemma:

LEMMA 7. *If $\succ$ is an arbitrary relation between invocations and operations, the result of merging $\succ$-closed sublogs of a particular log is itself a $\succ$-closed sublog.*

The following lemma is proved elsewhere [19].

LEMMA 8. *If $\succ$ is a serial dependency relation, g and h histories, and q an operation such that $g \cdot q$ and $g \cdot h$ are legal, and there is no p in h such that $inv(p) \succ q$, then $g \cdot q \cdot h$ is legal.*

The locks ensure that conflicting operations cannot execute concurrently.

LEMMA 9. *If a consensus-locking automaton has accepted a schedule H containing operations p and q executed by distinct active transactions, then inv(q)* $\not\succ_{LQ}$ *p and inv(p)* $\not\succ_{LQ}$ *q.*

PROOF. If not, some repository in the intersection of the quorums for p and q has granted conflicting locks. □

LEMMA 10. *Let h be a hybrid serialization of the accepted schedule H, S a set of repositories, and Q an active transaction. The following property is invariant: View(S, Q) is a* $\succ_{LQ}$*-closed subhistory of h.*

PROOF. By Lemma 7, it is enough to show that View(R, Q) is $\succ_{LQ}$-closed for a single repository R. We argue by induction on the length of the accepted schedule. The base case is immediate. Assume the automaton has accepted the schedule H, and let h be a hybrid serialization of H. By induction, h is legal.

Suppose the automaton accepts the operation step $\langle q\ Q\rangle$. If h is a hybrid serialization of H in which Q aborts, then h is also a legal hybrid serialization of $H \bullet \langle q\ Q\rangle$. Otherwise, h can be written as $h_1 \bullet h_2$ and the corresponding hybrid serialization h' of $H \bullet \langle q\ Q\rangle$ as $h_1 \bullet q \bullet h_2$. Let P be an active transaction distinct from Q. By induction, View(S, P) is $\succ_{LQ}$-closed in h. If View$'(S, P)$ is not $\succ_{LQ}$-closed in h', then it includes an operation p in h_2 such that inv(p) $\succ_{LQ}$ q. If P executed p, then p and q contradict Lemma 9. If a committed transaction executed p, the well-formedness constraints on commit timestamps ensure it is serialized before Q in every hybrid serialization; thus p cannot appear in h_2. Therefore, View$'(S, P)$ must be $\succ_{LQ}$-closed in h'.

The commit step for Q shrinks the set of hybrid serializations. If R is not part of a final quorum for Q, View$'(R, P)$ = View(R, P), which remains $\succ_{LQ}$-closed. Otherwise,

$$\text{View}'(R, P) = \text{View}(R, P) \cup \text{View}(R, Q)$$

which is closed by Lemma 7.

The abort step $\langle$Abort $Q\rangle$ shrinks the set of hybrid serializations but leaves View(S, P) unchanged. □

LEMMA 11. *Let h be a hybrid serialization of the accepted schedule H in which Q commits. If Q has acquired initial locks for inv(q) at IQ in initial(inv(q)), then View(IQ, Q) is a* $\succ_{LQ}$*-view of h for inv(q).*

PROOF. View(IQ, Q) is a $\succ_{LQ}$-closed subhistory of h (Lemma 10). If inv(q) $\succ_{LQ}$ p and transaction P executed p, then either P is committed or P is the same transaction as Q (Lemma 9). In either case, p appears in View(IQ, Q) because the initial quorum for inv(q) intersects the final quorum for p. □

We are now ready for the basic correctness result.

THEOREM 12. *If* $\succ_{LQ}$ *is a serial dependency relation, then every schedule accepted by a consensus-locking automaton is hybrid atomic.*

PROOF. We argue by induction that every hybrid serialization of every schedule accepted by the automaton is legal. The property is immediate after accepting the empty schedule, and it is clearly preserved after accepting a commit or abort step.

Assume the automaton has accepted the schedule H, and let h be a hybrid serialization of H. By induction, h is legal. Suppose the automaton accepts the operation step $\langle q\ Q\rangle$ with initial quorum IQ. If h is a hybrid serialization of H in which Q aborts, then h is also a legal hybrid serialization of $H \bullet \langle q\ Q\rangle$. Otherwise, h can be written as $h_1 \bullet h_2$ and the corresponding hybrid serialization h' of $H \bullet \langle q\ Q\rangle$ as $h_1 \bullet q \bullet h_2$. View($IQ$, Q) is a $\succ_{LQ}$-view of $h_1 \bullet h_2$ for inv(q) (Lemma 11), but since every operation in View(IQ, Q) is serialized before q, it is also a $\succ_{LQ}$-view of h_1. The precondition for accepting an operation step ensures that View(IQ, Q) $\bullet$ q is legal, and because $\succ_{LQ}$ is a serial dependency relation, $h_1 \bullet q$ is also legal (Definition 3). There is no p in h_2 such that inv(p) $\succ_{LQ}$ q (Lemma 9); thus $h_1 \bullet q \bullet h_2$ is legal (Lemma 8). □.

We now show that no weaker set of constraints on lock conflicts and quorum intersection can guarantee correctness. Our argument relies on the following lemma, proved in [19]:

LEMMA 13. *Let $\succ$ be a relation between invocations and operations. If $\succ$ is not a serial dependency relation, then there exist histories g and h and an operation q such that* (1) *g is a $\succ$-view of h for inv(q),* (2) *$g \bullet q$ is legal,* (3) *$h \bullet q$ is illegal, and g is missing exactly one operation of h.*

No quorum-consensus method can place weaker constraints on quorum assignment.

THEOREM 14. *If $\succ$ is not a serial dependency relation, there exists a consensus-locking automaton with quorum-intersection relation $\succ$ that accepts a schedule that is not hybrid atomic.*

PROOF. Let h, g, and q satisfy the conditions of Lemma 13 and let $h = g_1 \bullet p \bullet g_2$, and $g = g_1 \bullet g_2$. Consider an automaton with two repositories, $R1$ and $R2$. Transaction Q executes the operations in g_1, choosing $\{R1, R2\}$ for initial and final quorums. It then executes p, choosing $\{R2\}$ for initial and final quorums, and executes g_2, choosing $\{R1\}$ for initial and final quorums. Q executes q and commits, choosing an initial quorum of $\{R1\}$, and assembling the view g. These quorum intersections satisfy $\succ$, but the automaton has accepted an illegal schedule. □

A natural way to compare the concurrency permitted by locking schemes is to compare which operations may execute concurrently. For consensus locking, this set is determined by the *symmetric conflict* relation:

$$p \sim q \equiv \text{inv}(p) \succ_{LQ} q \vee \text{inv}(q) \succ_{LQ} p.$$

If $p \sim q$, the two operations cannot execute concurrently because one operation's initial locks will conflict with the other's final locks. By a minor abuse of notation, we say that $\sim$ is a serial dependency relation, if the relation $\sim'$ is a serial

dependency relation, where $\text{inv}(q) \sim' p$ if $q \sim p$. If $\succ_{LQ}$ is a serial dependency relation, so is $\sim$. Moreover, any such $\sim$ can be generated by some serial dependency relation $\succ_{LQ}$; thus any scheme for choosing an effective conflict relation permitting more concurrency than our scheme must induce a symmetric conflict relation that is not a serial dependency relation. We now show that no such scheme is possible.

THEOREM 15. *Let $\sim$ be a symmetric relation between operations. If $\sim$ is not a serial dependency relation, there exists a consensus-locking automaton with effective conflict relation $\sim$ that accepts a schedule that is not hybrid atomic.*

PROOF. Since $\sim'$ is not a serial dependency relation, pick h, g, and q to satisfy the conditions of Lemma 13, and let $h = g_1 \cdot p \cdot g_2$, $g = g_1 \cdot g_2$. Consider an automaton with two repositories, $R1$ and $R2$. Transaction A executes each of the operations in g_1 and commits, choosing both $R1$ and $R2$ as initial and final quorums for each operation. Transaction B executes p, choosing $R2$ as its initial and final quorums. Let $g_2 \cdot q = q_1 \cdot \cdots \cdot q_k$. Transaction C executes $g_2 \cdot q$, choosing the following quorums. If $\text{inv}(q_i) \succ_Q p$, choose the initial quorum $\{R1, R2\}$, otherwise choose $\{R1\}$. If $\text{inv}(p) \succ_Q q_i$, choose the final quorum $\{R1, R2\}$, otherwise choose $\{R1\}$. These quorums satisfy $\succ_Q$, and no lock conflicts arise because if $\text{inv}(q_i) \succ_L p$, then the initial quorum for q_i does not intersect the final quorum for p, and similarly if $\text{inv}(p) \succ_L q_i$. The accepted schedule is not hybrid atomic because it has the illegal hybrid serialization $h \cdot q$. □

A.3 Consensus Scheduling

For consensus scheduling, a replicated object is modeled by a nondeterministic *consensus scheduling automaton*, having the following components. Let STEP = OP ∪ "Commit" ∪ "Abort." It is convenient to redefine LOG to be TIMESTAMP → STEP.

State: REPOS → LOG
Clock: TIMESTAMP
Visited: TRANS → 2^{REPOS}

Visited (Q) is the set of repositories that participated in a quorum for Q, and *State*(R) is the log at R. If S is a set of repositories, define State(S) in the usual way.

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

—The concurrent specification *Concur*,
—Initial: INV → 2^{QUORUM}, and
—Final: OP → 2^{QUORUM}.

As before, Initial and Final define a quorum intersection relation $\succ_Q$.

The precondition for accepting an operation step $\langle q\ Q \rangle$ is the following. There must exist an initial quorum IQ in Initial(inv(q)) such that

$$\text{State}(IQ) \cdot \langle q\ Q \rangle \text{ is in } \textit{Concur}.$$

The schedule constructed by merging the logs from an initial quorum and appending the new step must lie within the concurrent specification. The

postcondition is the following. There must exist a final quorum FQ in Final(q) such that

$$\text{Clock}' > \text{Clock}$$
$$\text{For all } R \text{ in } FQ,\ \text{State}(R)(t) = \text{if } (t = \text{Clock}) \text{ then } \langle q\ Q\rangle \text{ else } (\text{State}(R) \cup \text{State}(IQ))(t)$$

The front end's log is merged with the logs at a final quorum, and the clock is advanced.

Commits and aborts are treated as operations with dummy invocations: The clock is advanced, and a commit or abort entry is appended to the log at each repository visited by the transaction.

A.4 Correctness Arguments

We first show that the view assembled for each operation is a $\succ_Q$-closed subschedule of the schedule accepted by the automaton.

LEMMA 16. *The result of merging logs from any set of repositories is $\succ_Q$-closed.*

PROOF. It suffices to show that the property holds for any single repository R; the more general result follows from Lemma 7. The argument is by induction on the length of the accepted schedule. The base case is immediate, and the result is clearly preserved when a commit or abort step is accepted.

Let $\langle q\ Q\rangle$ be an operation step accepted in a state satisfying the lemma. If R is outside the final quorum for q, then $\text{State}(R) = \text{State}'(R)$, which remains $\succ_Q$-closed. Otherwise,

$$\text{State}'(R) = \text{State}(R) \cup \text{State}(IQ) \bullet \langle q\ Q\rangle,$$

where "•" is shorthand for appending the new entry to the log as defined above. State(IQ) is $\succ_Q$-closed because it is the merger of closed logs (induction hypothesis and Lemma 7). State(IQ) • $\langle q\ Q\rangle$ is $\succ_Q$-closed by construction, and State(R) is $\succ_Q$-closed by the induction hypothesis; therefore State'(R) is $\succ_Q$-closed by Lemma 7. □

Constraints on quorum intersection yield:

COROLLARY 17. *If the automaton has accepted the schedule H, and IQ is in Initial(inv(q)), then State(IQ) is a $\succ_Q$-view of H for inv(q).*

We are now ready to present the basic correctness result:

THEOREM 18. *If the quorum intersection relation $\succ_Q$ is an atomic dependency relation for Concur, every schedule accepted by a quorum-consensus automaton is in concur.*

PROOF. The proof is by induction on the length of the accepted schedule H. The base case is immediate. Because Concur is on-line, the result holds when the automaton accepts commit or abort steps. Suppose the automaton accepts the operation step $\langle q\ Q\rangle$. State (IQ) is a $\succ_Q$-view of H for inv(q) (Corollary 17), State(IQ) • $\langle q\ Q\rangle$ is in Concur, and $\succ_Q$ is an atomic dependency relation for Concur; hence $H \bullet \langle q\ Q\rangle$ is also in Concur (Definition 6). □

No set of constraints on quorum intersection weaker than atomic dependency guarantees that all schedules accepted by a consensus-scheduling automaton are in Concur.

THEOREM 19. *If $\succ$ is not an atomic dependency relation for Concur, there exists a consensus-scheduling automaton for which $\succ_Q = \succ$ that accepts a schedule that is not in Concur.*

PROOF. By Definition 6, if $\succ_Q$ is not a serial dependency relation, there exist schedules G and H in Concur and an operation q, such that G is a $\succ$-view of H for inv(q), $G \bullet \langle q\ Q \rangle$ is in Concur, but $H \bullet \langle q\ Q \rangle$ is not. We construct a consensus-scheduling automaton with quorum-intersection relation $\succ_Q$ that accepts $H \bullet \langle q\ Q \rangle$.

The automaton has two repositories: $R1$ and $R2$. It accepts H and chooses the following quorums for each operation. For each operation in G, choose an initial quorum of $\{R1\}$ and a final quorum of $\{R1, R2\}$. For operations in H but not in G, choose an initial quorum of $\{R1, R2\}$ and a final quorum of $\{R2\}$. The view assembled for each operation in G contains all and only the prior steps in G, and the view assembled for every other operation contains all prior steps.

These quorums satisfy $\succ_Q$ because all initial and final quorums intersect, except the initial quorums for operations in G and the final quorums for operations not in G. If any of these quorums were required to intersect, then G would not be $\succ_Q$-closed, a contradiction.

The automaton then accepts $\langle q\ Q \rangle$, choosing an initial quorum of $\{R1\}$ and a final quorum of $\{R1, R2\}$.This transition is legal because the automaton assembles the view G, and $G \bullet \langle q\ Q \rangle$ is in Concur. The accepted schedule $H \bullet \langle q\ Q \rangle$ is not in Concur; thus the automaton is incorrect.

REFERENCES

1. ALSBERG, P. A., AND DAY, J. D. A principle for resilient sharing of distributed resources. In *Proceedings of 2d Annual Conference on Software Engineering* (San Francisco, Calif., Oct. 13–15, 1976). IEEE, New York, 1976.
2. BERNSTEIN, P. A., AND GOODMAN, N. The failure and recovery problem for replicated databases. In *Proceedings of the 2d ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Montreal, Quebec, Aug. 17–19, 1983). ACM, New York, 1983, 114–122.
3. BERNSTEIN, P. A., AND GOODMAN, N. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. Database Syst. 9*, 4 (Dec. 1984), 596–615.
4. BERNSTEIN, P. A., GOODMAN, N., AND LAI, M. Y. Two-part proof schema for database concurrency control. In *Proceedings of 5th Berkeley Workshop on Distributed Data Management and Computer Networks* (Berkeley, Calif., Feb. 1981). Lawrence Berkeley Laboratory, 1981, 71–84.
5. BIRMAN, K. P. Replication and fault-tolerance in the ISIS system. In *Proceedings of 10th Symposium on Operating Systems Principles.* (Orcas Island, Wash., Dec. 1–4, 1985). ACM, New York, 1985, 79–86. Also published as Tech. Rep. 85-668, Cornell University, Computer Science Dept., Ithaca, N.Y.
6. BIRRELL, A. D., LEVIN, R., NEEDHAM, R., AND SCHROEDER, M. Grapevine: an exercise in distributed computing. *Commun. ACM 25*, 14 (Apr. 1982), 260–274.
7. BLOCH, J. J., DANIELS, D. S., AND SPECTOR, A. Z. A weighted voting algorithm for replicated directories. 1987. To appear, *J. ACM.*

8. COOPER, E. C. Circus: a replicated procedure call facility. In *Proceedings of the 4th Symposium on Reliability in Distributed Software and Database Systems* (Oct. 1984), 11–24.
9. EAGER, D. L., AND SEVCIK, K.C. Achieving robustness in distributed database systems. *ACM Trans. Database Syst. 8*, 3 (Sept. 1983), 354–381.
10. EL-ABBADI, A., AND TOUENG, S. Availability in partitioned replicated databases. Tech. Rep. 85-721, Dept. of Computer Science, Cornell University, Ithaca, N.Y., Dec. 1985.
11. EL-ABBADI, A., SKEEN, D., AND CRISTIAN, F. An efficient, fault-tolerant protocol for replicated data management. In *Proceedings of the 4th ACM SIGACT/SIGMOD Symposium on Principles of Database Systems* (Portland, Ore., Mar. 25–27, 1985). ACM, New York, 1985, 215–229.
12. ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The notion of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.
13. FISCHER, M. J., AND MICHAEL, A. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings of ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Los Angeles, Calif., Mar. 29–31, 1982). ACM, New York, 1982, 70–75.
14. GIFFORD, D. K. Weighted voting for replicated data. In *Proceedings of the 7th Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec. 10–12, 1979). ACM, New York, 1979, 150–162.
15. GRAY, J. N. *Notes on Database Operating Systems.* Lecture Notes in Computer Science Vol. 60. Springer-Verlag, Berlin, 1978, pp. 393–481.
16. HAMMER, M. M., AND SHIPMAN, D. W. Reliability mechanisms in SDD-1, a system for distributed databases. *ACM Trans. Database Syst. 5*, 4 (Dec. 1980), 431–466.
17. HERLIHY, M. P. Comparing how atomicity mechanisms support replication. In *Proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computing* (Minaki, Ontario, Canada, Aug. 5–7, 1985). ACM, New York, 1985, 102–110.
18. HERLIHY, M. P. A quorum-consensus replication method for abstract data types. *ACM Trans.Comput. Syst. 4*, 1 (Feb. 1986), 32–53.
19. HERLIHY, M. P. Optimistic concurrency control for abstract data types. In *Proceedings of the 5th Annual Symposium on Principles of Distributed Computing* (Calgary, Alberta, Canada, Aug. 11–13, 1986). ACM, New York, 1986, 206–217.
20. HERLIHY, M. P. Dynamic quorum adjustment for partitioned data. *ACM Trans. Database Syst. 12*, 2 (June 1987), 170–194.
21. HERLIHY, M. P. Extending multiversion timestamping protocols to exploit type information. *IEEE Trans. Comput. C-35*, 4 (Apr. 1987), 443–449. Special issue on parallel and distributed computing.
22. JOHNSON, P. R., AND THOMAS, R. H. The maintenance of duplicate databases. Tech. Rep. RFC 677 NIC 31507, Network Working Group (Jan. 1975).
23. KOHLER, W. H. A survey of techniques for synchronization and recovery in decentralized computer systems. *ACM Comput. Surv. 13*, 2 (June 1981), 149–185.
24. KORTH, H. F. Locking primitives in a database system. *J. ACM 30*, 1 (Jan. 1983), 55–79.
25. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.
26. LAMPSON, B. *Atomic Transactions.* Lecture Notes in Computer Science Vol. 105: Distributed Systems: Architecture and Implementation. Springer-Verlag, Berlin, 1981, 246–265.
27. MINOURA, T., AND WIEDERHOLD, G. Resilient extended true-copy token scheme for a distributed database system. *IEEE Trans. Softw. Eng. 8*, 3 (May 1982), 173–188.
28. MOSS, J. E. B. Nested transactions: an approach to reliable distributed computing. Tech. Rep. MIT/LCS/TR-260, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Mass. Apr. 1981.
29. OPPEN, D., AND DALAL, Y. K. The clearinghouse: a decentralized agent for locating named objects in a distributed environment. Tech. Rep. OPD-T8103, Xerox Corporation, Palo Alto, Calif. (Oct. 1981).
30. PAPADIMITRIOU, C. H. The serializability of concurrent database updates. *J. ACM 26*, 4 (Oct. 1979), 631–653.
31. SKEEN, M. D. Crash recovery in a distributed database system. Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, May 1982.

32. THOMAS, R. H. Consensus approach to concurrency control for multiple-copy databases. *ACM Trans. Database Syst. 4*, 2 (June 1979), 180–209.
33. WEIHL, W. E. Specification and implementation of atomic data types. Tech. Rep. 314, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Mass. (Mar. 1984).

Received February 1985; revised March 1987; accepted April 1987