Concurrency and Availability as Dual Properties of Replicated Atomic Data

MAURICE HERLIHY

Carnegie Mellon University, Pittsburgh, Pennsylvania

Abstract. A replicated data object is a typed object that is stored redundantly at multiple locations in a distributed system. Each of the object's operations has a set of quorums, which are sets of sites whose cooperation is needed to execute that operation. A quorum assignment associates each operation with its set of quorums. An operation's quorums determine its availability, and the constraints governing an object's quorum assignments determine the range of availability properties realizable by replication.

In this paper, the restrictions on quorum assignment imposed by three kinds of atomicity mechanisms found in the literature are analyzed: (1) serial schemes, in which replication and atomicity are implemented independently at different levels in the system, (2) static schemes, in which the transaction serialization order is predetermined, and (3) hybrid schemes in which the serialization order emerges dynamically.

The following results are derived: (1) Although serial schemes place the strongest restrictions on concurrency, they place the weakest restrictions on availability. (2) Although hybrid and static mechanisms place incomparable restrictions on concurrency, hybrid mechanisms place weaker restrictions on availability. (3) Bounding the maximum depth of transaction nesting strengthens restrictions on concurrency for all classes, but weakens restrictions on availability for hybrid schemes only. Concurrency and availability are best considered as dual properties: A complete analysis of an atomicity mechanism should take both into account.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management—*distributed memories*; D.4.3 [**Operating Systems**]: File Systems Management—*distributed file systems*; D.4.5 [**Operating Systems**]: Reliability; H.2.4 [**Database Management**]: Systems—*concurrency, distributed systems*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Quorum consensus, replication

1. *Introduction*

Mechanisms for implementing atomicity in distributed systems fall into several broad categories, depending on how the serialization order for transactions is chosen. The serialization order may be predetermined, as in multiversion time-stamping schemes (e.g., [35, 36, 41]), or it may be chosen dynamically, as in

A preliminary version of this paper appeared in *Proceedings of the 4th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Minaki, Ont., Canada, Aug. 5–7). ACM, New York, 1985, pp. 102–110.

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.

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.

Author's present address: Digital Equipment Corporation, Cambridge Research Lab, One Kendall Square, Cambridge, MA 02139.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1990 ACM 0004-5411/90/0400-0257 $01.50

two-phase locking schemes (e.g., [15, 29, 37]) and in hybrid schemes employing both locking and timestamp-like mechanisms (e.g., [3, 6, 10, 12]).

This paper proposes a new criterion for evaluating these approaches: the constraints they impose on the availability of replicated data. Our analysis of availability is based on *quorum consensus* replication [24, 28]. A *replicated object* is a typed object that is stored redundantly at multiple locations in a distributed system. Each of the object's operations has a set of *quorums*, which are sets of sites whose cooperation is needed to execute that operation. A *quorum assignment* associates each operation with its set of quorums. An operation's quorums determine its availability, and the constraints governing an object's quorum assignments determine the range of availability properties realizable by replication. An analysis of the object's type specification yields a set of constraints on quorum assignment necessary and sufficient to ensure the correctness of the replicated implementation. Quorum consensus replication systematically exploits type-specific properties of data to support better availability and concurrency than conventional methods in which operations are classified only as reads or writes.

Our analysis of atomicity mechanisms is based on the notion of a *local atomicity property* [44]. Each class of mechanisms is identified with a local property of objects that suffices to ensure the atomicity of a system encompassing multiple objects. We consider three distinct local atomicity properties, as follows:

—*Serial atomicity* models replication methods in which concurrency control is handled by an independent underlying atomicity mechanism.
—*Static atomicity* encompasses the timestamping mechanisms cited above.
—*Hybrid atomicity* encompasses the locking and hybrid mechanisms.

If $\mathbf{P}_1$ and $\mathbf{P}_2$ are local atomicity properties, we say that $\mathbf{P}_1$ places weaker restrictions on *concurrency* than $\mathbf{P}_2$ if each interleaving permitted by $\mathbf{P}_2$ is also permitted by $\mathbf{P}_1$. The left-hand-side of Figure 1 compares the restrictions on concurrency imposed by these properties. Static and hybrid atomicity place incomparable restrictions on concurrency: Each permits interleavings the other does not. Serial atomicity places the strongest restrictions on concurrency: Each interleaving permitted by serial atomicity is permitted by the others, but not vice-versa.

As discussed in Section 3, each local atomicity property induces a set of restrictions on quorum assignment. We say that $\mathbf{P}_1$ places weaker restrictions on *availability* than $\mathbf{P}_2$ if each quorum assignment permitted by $\mathbf{P}_2$ is also permitted by $\mathbf{P}_1$. This paper compares the restrictions on availability imposed by the three local atomicity properties listed above, presenting the following results:

(1) Although serial atomicity places stronger restrictions on concurrency than static or hybrid atomicity, it places weaker restrictions on availability (Figure 1).
(2) Although static and hybrid atomicity place incomparable restrictions on concurrency, hybrid atomicity places weaker restrictions on availability (Figure 1).
(3) Transaction nesting can affect availability. Under hybrid atomicity, restrictions on availability can be relaxed by bounding the maximum depth of transaction nesting: the smaller the bound, the larger the set of permissible quorum assignments (Figure 2). By contrast, quorum assignments for static and serial atomicity are unaffected by transaction nesting.

These results illustrate the complex relation between the availability and concurrency supported by various local atomicity properties. Availability and concurrency are neither completely independent nor completely dependent: One cannot always

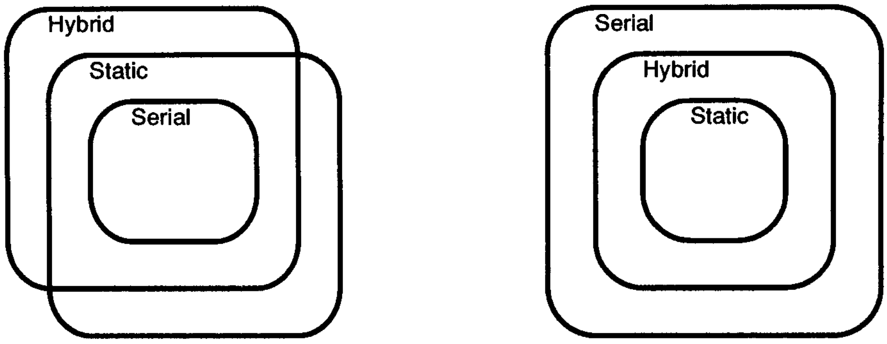

FIG. 1. Concurrency (L) vs. Availability (R).

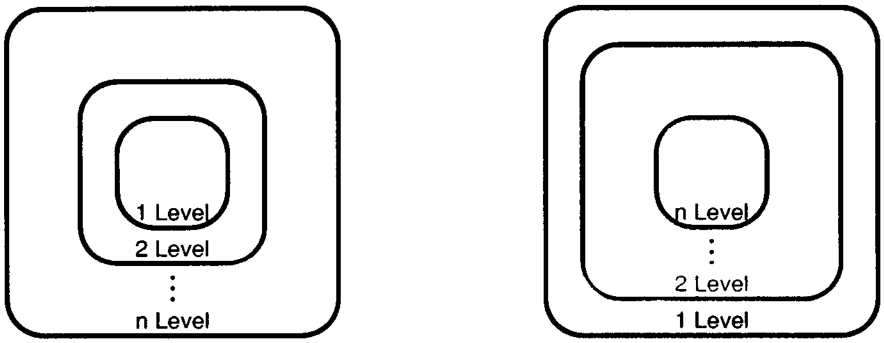

FIG. 2. Hybrid Atomicity: Concurrency (L) vs. Availability (R) for Nested Transaction.

minimize the restrictions on both, but strengthening the restrictions on one does not necessarily weaken the restrictions on the other. Instead, availability and concurrency are best considered as complementary properties, and a complete analysis of an atomicity property should take both into account.

The restrictions on availability analyzed in this paper are those necessary to realize the complete set of interleavings permitted by each local atomicity property. In practice, these restrictions are likely to be conservative, since practical schedulers typically admit only a subset of the schedules permitted by the system-wide local atomicity property (cf. [34, 35]). Nevertheless, a thorough understanding of this limiting case is a necessary step in understanding the entire range of trade-offs, which in turn is helpful for evaluating the alternative atomicity properties. Choosing the local atomicity property around which a distributed system will be organized is an important design decision; the property must be established in advance, perhaps before the demands of the application domain are fully understood, and once made, it is difficult to change.

Section 2 summarizes related work, Section 3 describes our terminology and assumptions, and Section 4 compares how the local atomicity properties support replication. Section 5 concludes with a discussion.

2. *Related Work*

The notion that replication should not affect an object's functional specification is known as *one-copy serializability* [4]. Early replication methods for files [1, 42] did not preserve one-copy serializability, nor do more recent replication methods for directories [8, 16]. Henceforth, we consider only techniques that preserve one-copy serializability.

Replication methods for files that tolerate site crashes include SDD-1 [20], the Available Copies algorithm [5], Circus [11], and ISIS [7]. Methods that also tolerate network partitions and timing anomalies include those proposed by Gifford [17], Eager and Sevcik [13], Skeen et al. [39], and by El-Abbadi and Toueg [14].

The replication method considered in this paper differs from the methods cited above in two important respects. First, rather than classifying operations only as reads and writes, our method systematically exploits type-specific properties of the data to provide more effective replication. Second, rather than using distinct mechanisms for replication and concurrency control, our method integrates both functions in a single mechanism. Although independent methods are simpler, integrated methods support better concurrency.

A quorum-consensus replication method for directories has been proposed by Bloch et al. [9]. The quorum-consensus file replication methods of Gifford and Eager and Sevcik can be generalized to exploit type information [25, 26]. The replication method considered here is a slight generalization of the author's *consensus scheduling* method [22], which is described in more detail below.

Goldman and Lynch [18] give a formal proof of Gifford's quorum consensus method, showing that its correctness is independent of the underlying atomicity mechanism. This paper adds a new perspective to that result: Although Gifford's original algorithm can be understood by considering replica management and concurrency control independently, a complete understanding of the range of availability and concurrency realizable by more general quorum consensus algorithms requires considering both properties together.

Formal models for nested transaction systems have been proposed by Lynch [31], Lynch and Merritt [32], and Beeri et al. [2]. The notion of a local atomicity property, as well as the particular properties investigated here, are due to Weihl [44]. Our treatment here borrows from each of these works.

3. *Model*

3.1 OBJECTS AND HISTORIES. 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 File might provide Read and Write operations, and a first-in-first-out (FIFO) Queue might provide Enq and Deq operations.

In the absence of failures and concurrency, a computation is modeled as a *history*, which is a sequence of object-operation pairs. Histories are denoted by lower-case letters (g, h). An *operation* is written as $op(args^*)/term(res^*)$, where x is an object name, op is an operation name, $args^*$ a sequence of argument values, $term$ a termination condition, and res^* a sequence of results. We use "Ok" for normal termination. For example, the following is a history of a queue object q:

$$\begin{array}{l} q\ \text{Enq}(x)/\text{Ok}(\,) \\ q\ \text{Enq}(y)/\text{Ok}(\,) \\ q\ \text{Deq}(\,)/\text{Ok}(x) \end{array}$$

An *object subhistory*, $h \mid x$ (h at x), is the subsequence of object-operation pairs whose object names are x. Each object has a *serial specification*, which defines a set of *legal* histories for that object. For example, the serial specification for a FIFO queue would permit all and only those histories in which items are enqueued and dequeued in FIFO order. A history h involving multiple objects is *legal* if each object subhistory $h \mid x$ lies within the serial specification for x. The object name is often omitted when it is clear from context.

3.2 TRANSACTIONS AND SCHEDULES. 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 network partitions. A crash renders a site's data temporarily or permanently inaccessible, while a network partition prevents functioning sites from communicating. 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 approach to ensuring consistency in the presence of crashes and network partitions is to make the activities that manage the data *atomic*. Atomicity encompasses two properties: serializability and recoverability. *Serializability* [34] means that the execution of one activity never appears to overlap (or contain) the execution of another, while *recoverability* means that the overall effect of an activity is all-or-nothing: It either succeeds completely, or it has no effect. Atomic activities are called *transactions*. A transaction's effects become permanent when it *commits*, its effects are discarded if it *aborts*, and a transaction that has not committed or aborted is *active*. Well-known atomic commitment protocols (e.g., [19, 38]) can be used to ensure the recoverability of distributed transactions.

Instead of treating transactions as monolithic entities, it is often useful to provide hierarchically structured nested transactions or *subtransactions* [33, 36]. A subtransaction's commit is dependent on that of its parent; aborting the parent will undo a committed child's effects. A transaction's effects become permanent only when it commits at the top level.

Our transaction model is essentially that of Lynch [31], extended to encompass typed objects. Let TRANS denote the domain of transactions. Transactions have a predefined tree structure, with a distinguished transaction U as the root. It is convenient to provide each transaction with a countably infinite set of children. For a transaction P distinct from U, let *parent*(P) denote P's unique parent, *anc*(P) and *desc*(P) P's ancestors and descendants (which include P), *proper-anc*(P) and *proper-desc*(P) P's proper ancestors and descendants (which do not include P). Let *lca*(P, Q) denote the least common ancestor of P and Q, and *laa*(P) the least active ancestor of P. *Siblings* is the set $\{(P, Q) \in \text{TRANS}^2 \mid \text{parent}(P) = \text{parent}(Q)\}$. Children of the root are called *top-level* transactions.

A *schedule* is a sequence of *steps* of the form: $\langle x \text{ p } P\rangle$, $\langle x \text{ commit } P\rangle$, or $\langle x \text{ abort } P\rangle$, where x is an object, p an operation, and P a transaction. Schedules are denoted by upper-case letters (G, H). If H is a schedule and x an object name, $H \mid x$ is the subschedule of H consisting of steps whose object names are x. If P is a transaction and S a set of transactions, $H \mid P$ and $H \mid S$ are defined analogously. The object name is often omitted when it is clear from context. A transaction P *appears* in H if $H \mid P$ is nonempty.

When a transaction commits or aborts, news of the event propagates asynchronously through the system. A schedule's commit and abort steps represent the arrival of such news at an object (or a component of that object if it is replicated).

A schedule H is *well formed* if it satisfies the following properties:

—No transaction executes an operation after it commits.
—No transaction both commits and aborts.
—No transaction commits until all its children that appear in H have either committed or aborted.
—A transaction P is a *leaf* transaction in H if no proper descendant of P appears in H. Operations may be associated with leaf transactions only.

The first three conditions reflect the generally accepted semantics of transactions, and the last condition simplifies later definitions. Henceforth, all schedules are assumed to be well formed.

Let *committed*(H) be the set of transactions that have taken commit steps in H, and let *aborted*(H) be the set of transactions having an ancestor that has taken an abort step in H. Let *active*(H) be the set of transactions not in committed(H) or aborted(H). Transaction Q has *committed to* P in H if

$$\text{anc}(Q) \cap \text{proper-desc}(\text{lca}(P, Q)) \subseteq \text{committed}(H).$$

Informally, if Q is committed to P, then if the effects of P become permanent, so will the effects of Q.

A *commit set* S for a schedule H is any set of transactions that might eventually commit to the root transaction U:

—$U \in S$,
—$P \in S \Rightarrow P \notin \text{aborted}(H)$,
—$P \in S \wedge Q \in \text{anc}(P) \Rightarrow Q \in S$, and
—$P \in S \wedge P = \text{parent}(R) \wedge R \in \text{committed}(H) \Rightarrow R \in S$

Recall that operations may be executed only by leaf transactions. Let $\mathbf{R}$ be a relation on transactions appearing in H. If $\mathbf{R}$ partially (or totally) orders sets of active and committed siblings in H, then $\mathbf{R}$ induces a partial (total) order $\mathbf{R}'$ on active and committed leaf transactions in H: $(P, Q) \in \mathbf{R}'$ if there exist $P' \in \text{anc}(P) \wedge Q' \in \text{anc}(Q)$ such that $(P', Q') \in \mathbf{R}$.

Definition 1. A relation $\mathbf{P} \subseteq$ *siblings* is a *linearizing order* [31] if it totally orders each set of siblings.

To keep our notation from becoming cumbersome, we will not always distinguish between a linearizing order on transactions and the induced total order on leaf transactions.

Let $\mathbf{L}$ be a linearizing order for H. H is *serializable in the order* $\mathbf{L}$ if the history constructed by reordering H's object-operation pairs in the induced order $\mathbf{L}'$ is legal. A schedule is *serializable* if it is serializable in some linearizing order. Let *perm*(H) be the subschedule of H associated with the transactions that have committed to the root transaction U. H is *atomic* if perm(H) is serializable, and H is *on-line atomic* if $H \mid S$ is serializable for every commit set S of H. Informally, on-line atomicity captures the assumption that objects have no advance knowledge of which transactions will eventually commit.

A *concurrent specification* is a set of schedules for an object. A concurrent specification is *atomic* if all its schedules are on-line atomic. All concurrent specifications considered in this paper are assumed to be atomic and *prefix-closed*: if H is in a specification, so is every prefix of H.

H is not necessarily atomic just because $H \mid x$ is atomic for all objects x. A property $\mathbf{P}$ is a *local atomicity property* [43, 44] if H is atomic provided that each

$H \mid x$ is atomic and satisfies **P**. If a system-wide local atomicity property is agreed on in advance, then objects can be implemented independently subject only to the constraint that each implementation satisfies the system's local atomicity property. This paper compares and evaluates alternative local atomicity properties.

Our basic tool for showing that a property is a local atomicity property is the following lemma (cf. Lemma 3-2 in [44]).

LEMMA 2. *If* L *is a linearizing order such that each* $H \mid x$ *is serializable in the order* L, *then H itself is serializable in the order* L.

PROOF. Let *ser*(H, L) denote the history constructed by reordering the object-operation pairs of H in the order L. For all x, the history ser($H \mid x$, L) = ser(H, L)$\mid x$ is legal; hence, ser(H, L) is legal, thus H is serializable in the order L. □

3.3 QUORUM CONSENSUS REPLICATION. In this section we review quorum consensus replication. Informally, a replicated object is 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. Different objects may have different sets of repositories. 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. Internally, a repository's state is represented as a *log*, which is a sequence of *entries*, where an entry is the timestamped record of an operation. (In practice, logs can usually be replaced by more compact data structures [21, 24].) A client executes an operation by sending an invocation to a front-end. The front-end merges the logs from an *initial quorum* of repositories to construct a *view*. It chooses a response consistent with the view, appends the new entry to the view, and writes out the updated view to a *final quorum* of repositories. A quorum for an operation is any set of repositories that includes both an initial and a final quorum. As discussed elsewhere [24], front-ends executing concurrent operations synchronize through short-term locks at repositories.

Formally, we model a replicated object by a nondeterminstic *consensus scheduling automaton* that accepts certain schedules. We use the following primitive domains: REPOS is the set of repositories, TRANS the set of transaction identifiers, OP the set of operations, and TIMESTAMP a totally ordered set with the order type of the natural numbers. $X \rightarrow Y$ denotes the set of partial maps from X to Y. If f is a partial map from X to Y, $x \in X$, and $y \in Y$, let $f[x \rightarrow y]$ denote the function identical to f except at x, which is mapped to y. We use the following derived domains: STEP = (OP ∪ "Commit" ∪ "Abort") × TRANS is the set of steps, QUORUM = 2^{REPOS} the set of quorums, and a *log* is a partial map with a finite domain from timestamps to steps: LOG = TIMESTAMP → STEP. Two logs L and M are *coherent* if they agree for every timestamp where they are both defined. The *merge* operation ∪ 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 schedule in the obvious way. For brevity, we sometimes refer to a log L in place of its schedule, for example, "L is atomic" instead of "the schedule represented by L is atomic."

A consensus scheduling automaton has the following state components:

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

Clock models a system of logical clocks, *state*(R) is the log at R, and *visited*(P) is the set of repositories that participated in a quorum for an operation of P. (Because front-ends' individual accesses to repositories are serialized by short-term locks, it is not necessary to model front-ends explicitly.) If S is a set of repositories, define *State*(S) to be the result of merging logs from repositories in S. (Because each step is given a unique timestamp, logs at distinct repositories are always coherent, thus State(S) is well defined.)

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

—A concurrent specification *Concur*,
—Initial: $\text{OP} \rightarrow 2^{\text{QUORUM}}$, and
—Final: $\text{OP} \rightarrow 2^{\text{QUORUM}}$.

Initial and *Final* define an *initial/final quorum intersection relation*:

$$\textbf{IFQ} = \{(q, p) \mid \text{IQ} \in \text{Initial}(q) \wedge \text{FQ} \in \text{Final}(p) \Rightarrow \text{IQ} \cap \text{FQ} \neq \varnothing\}.$$

The automaton's transitions are characterized by pre- and postconditions. In postconditions, primed component names denote new values, and unprimed names denote old values. Any state component not mentioned in a postcondition is left unchanged.

For an operation step $\langle q Q \rangle$, there must exist an initial quorum IQ $\in$ Initial(q) and a final quorum FQ $\in$ Final(Q) such that

Precondition:

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

Postcondition:

$$\begin{aligned}&\text{For all } R \text{ in FQ},\\ &\quad \text{State}'(R) = (\text{State}(R) \cup \text{State(IQ)})[\text{Clock} \rightarrow \langle q Q \rangle]\\ &\text{Clock}' > \text{Clock}\\ &\text{Visited}' = \text{Visited}[Q \rightarrow \text{Visited}(Q) \cup \text{IQ} \cup \text{FQ}]\end{aligned}$$

To accept $\langle q Q \rangle$, the schedule constructed by merging the logs from an initial quorum and appending the new step must be permitted by the object's concurrent specification. When the step is complete, the extended log is merged with the logs at a final quorum, the clock is advanced, and the transaction's set of visited repositories is updated.

Commit and abort steps have initial and final quorums just like operations. All repositories visited by Q form a quorum for $\langle \text{commit } Q \rangle$, and any visited repository is a quorum for $\langle \text{abort } Q \rangle$. This convention mirrors the assumptions underlying the standard commitment protocols ([19, 38]), which permit any site visited by a transaction to abort unilaterally, as long as the commitment protocol is not in progress. For Q to commit

Precondition:

$$\text{For all } R \text{ in Visited}(Q), \langle \text{abort } Q \rangle \notin \text{State}(R).$$

Postcondition:

$$\begin{aligned}&\text{Visited}' = \text{Visited}[\text{parent}(Q) \rightarrow \text{Visited}(\text{parent}(Q)) \cup \text{Visited}(Q)]\\ &\text{For all } R \text{ in Visited}(Q), \text{State}'(R) = \text{State}(R)[\text{Clock} \rightarrow \langle \text{commit } Q \rangle]\\ &\text{Clock}' > \text{Clock}\end{aligned}$$

This postcondition states that if Q commits, its set of visited repositories is merged with its parent's, and a "commit record" is appended to the log of each repository visited.

For Q to abort:

Precondition:

$$\text{For some } R \text{ in Visited}(Q),\ \langle \text{commit } Q\rangle \notin \text{State}(R).$$

Postcondition:

$$\begin{aligned}&\text{For some } R \text{ in Visited}(Q),\ \text{State}'(R) = \text{State}(R)[\text{Clock} \rightarrow \langle \text{abort } Q\rangle]\\ &\text{Clock}' > \text{Clock}\end{aligned}$$

Let H be a schedule, G a subschedule (i.e., subsequence) of H, and $\mathbf{D}$ a binary relation between operations.

Definition 3. *G is a* **D**-*closed* subschedule of H if whenever G contains a step $\langle qQ\rangle$ of H it also contains every earlier step $\langle pP\rangle$ such that $(q, p) \in \mathbf{D}$.

Definition 4. A subschedule G of H is a **D**-*view* of H for q if G is **D**-closed, and if it includes every $\langle pP\rangle$ in H such that $(q, p) \in \mathbf{D}$.

Definition 5. A relation **D** between operations is a *dependency relation* for the concurrent specification *Concur* if for all operations q, all transactions Q, and all schedules G and H in *Concur*,

$$G \cdot \langle qQ\rangle \in \mathit{Concur} \Rightarrow H \cdot \langle qQ\rangle \in \mathit{Concur},$$

whenever G is a **D**-view of H for q.

A relation **D** is a *minimal* dependency relation if no $\mathbf{D}' \subset \mathbf{D}$ is a dependency relation.

The notion of dependency provides a necessary and sufficient condition for a quorum assignment to be correct.

THEOREM 6. *Every schedule accepted by a quorum consensus automaton is in* Concur *if the initial/final quorum intersection relation* **IFQ** *is a dependency relation for* Concur. *Moreover, given a relation* **IFQ** *that is not a dependency relation, there exists a consensus scheduling automaton with initial/final quorum intersection relation* **IFQ** *that accepts a schedule not in* Concur.

This theorem is proved elsewhere [22]. That proof makes two assumptions that differ from those made here: (1) dependency is defined between invocations and operations rather than between operations, and (2) transactions are single-level rather than nested. Neither difference affects the proof in any substantial way. If dependency is defined between invocations and operations, then fewer messages are needed to implement consensus scheduling, but some flexibility in quorum assignment is lost. Here we have chosen generality over efficiency. The structure of the automaton and the notion of atomic dependency are also essentially independent of the depth of transaction nesting. The automaton treats all steps as uninterpreted symbols, merging logs from an initial quorum, checking the specification, and writing the updated view to a final quorum. The constraints on quorum assignment are derived from the specification itself, independently of whether the specification is interpreted as encompassing serial histories [24], schedules with single-level transactions [22], or schedules with nested transactions, as here. It is also worth noting that the definition of dependency and the correctness proof make no assumptions about the nature of the well-formedness conditions.

The requirement that the initial/final quorum intersection relation be a dependency relation characterizes the range of legal quorum assignments for an object, and indirectly characterizes the range of availability properties realizable by quorum consensus replication. Availability is more directly characterized by the following relation:

Definition 7. An object's *quorum intersection relation* is the symmetric closure of its initial/final quorum intersection relation.

Informally, $(p, q) \in Q$ if every quorum for p *must* intersect every quorum for q in every correct quorum assignment. If the quorum for p is made smaller (making p more available), then the quorum for q must be made larger (making q less available). Two distinct initial/final quorum intersection relations may induce the same quorum intersection relation; their respective sets of quorum assignments will have different patterns of message traffic, but identical levels of availability.

4. *Serial, Static, and Hybrid Atomicity*

4.1 LOCAL ATOMICITY PROPERTIES. Most replication methods in the literature treat replication and concurrency control as distinct problems. At the higher level, the replication method reconstructs the object's state from its distributed components without concern for concurrency and failures. At the lower level, a standard concurrency control mechanism serializes uninterpreted accesses to repositories. Examples of concurrency control mechanisms that would generate such behavior include Moss' hybrid atomic two-phase locking scheme [33] (if each access acquires an exclusive lock), and Reed's static atomic multiversion timestamp scheme [36] (if each access is treated as a combination read and write).

Definition 8. Let H be a schedule, and let P and Q be sibling transactions. Define the relation precedes$(H) \subseteq$ *siblings*: $(P, Q) \in$ precedes(H) if a transaction is desc(Q) executes an operation after P commits.

Informally, if P precedes Q, then Q may have observed that P committed. A schedule H is *serial* if precedes(H) is a linearizing order on active and committed transactions.

Definition 9. A schedule H is *serial atomic* if, for each object x, perm$(H \mid x)$ is serial and serializable in the order precedes$(H \mid x)$. H is *on-line serial atomic* if, for all commit sets S, $H \mid x \mid S$ is serial and serializable in the same order.

Since each $H \mid x$ is serializable in the linearizing order precedes$(H \mid x)$, Lemma 2 implies that serial atomicity is a local atomicity property.

THEOREM 10. *If each $H \mid x$ is serial atomic, then H is atomic.*

Let Serial(T) denote the largest prefix-closed set of on-line serial atomic schedules for the data type T. We define a *serial consensus scheduling automaton* to be a consensus scheduling automaton with concurrent specification Serial(T) and one additional well-formedness constraint: all input schedules are serial. Informally, this definition is intended to capture the notion that serialization is enforced by an independent lower-level mechanism, but that serial atomicity is enforced by the replication method. A dependency relation for such an automaton is called a *serial dependency relation* [24]. Let Serial*(T) denote the associated set of quorum intersection relations.

Static atomicity [44] encompasses multiversion timestamping techniques [25, 35, 36, 41] in which each transaction chooses a timestamp when it begins execution, and transactions must remain serializable in timestamp order.

The domain of timestamps is a set with the order type of the natural numbers.

Definition 11. A *timestamp ordering* on transactions is a map from transactions to natural numbers whose restrictions to sets of siblings are bijective (i.e., one-to-one and onto).

We often use "*ts*" to denote both a map and the linearizing order it induces. Let *ts* be a fixed timestamp ordering.

Definition 12. A schedule H is *static atomic* if perm(H) is serializable in the order *ts*. H is *on-line static atomic* if $H \mid S$ is serializable in the order *ts* for every commit set S of H.

Since each $H \mid x$ is serializable in the linearizing order *ts*, Lemma 2 implies that static atomicity is a local atomicity property:

THEOREM 13. *If each $H \mid x$ is static atomic, then H is atomic.*

Let Static(T, *ts*) denote the largest prefix-closed set of on-line static atomic schedules for the data type T, where transactions are ordered by *ts*, and let Static*(T, *ts*) be the corresponding set of quorum intersection relations. We now show that the concurrency and availability permitted by static atomicity is independent of the particular choice of timestamp ordering.

LEMMA 14. *If ts and ts′ are distinct timestamp orders, then Static(T, ts) and Static(T, ts′) are identical up to renaming of transactions.*

PROOF. Let H be a schedule in Static(T, *ts*), and define ϕ: TRANS $\rightarrow$ TRANS so that $\phi(Q) = ts'^{-1}(ts(Q))$, which is a bijective map carrying each set of siblings to itself. Let H' be the schedule constructed by replacing each transaction Q in H with $\phi(Q)$. It is easy to check that if S' is a commit set of H', then $S = \{\phi^{-1}(Q) \mid Q \in S'\}$ is a commit set of H. The serialization of $H' \mid S'$ in the order *ts′* is identical to the serialization of $H \mid S$ in the order *ts*. Since H is in Static(T, *ts*), all such serializations are legal, implying that H' is in Static(T, *ts′*). □

A similar argument shows that Static*(T, *ts*) is identical to Static*(T, *ts′*). Henceforth, we omit explicit mention of *ts* except when necessary.

Hybrid atomicity [44] encompasses techniques in which transactions' serialization orders are determined dynamically as they commit.[1] Hybrid atomic techniques encompass two-phase locking protocols (e.g., [15, 29, 37]) and protocols that combine locking with timestamp-like mechanisms (e.g., [3, 6, 10, 12, 22, 27]).[2]

Definition 15. As before, let *ts* be a timestamp ordering. For a schedule H, let *ts*(H) be *ts* restricted to committed(H).

Definition 16. A schedule H is *hybrid atomic* if precedes(H) $\subseteq$ *ts*, and perm(H) is serializable in the order *ts*(H). Let *known*(H) be *ts*(H) $\cup$ precedes(H). H is on-line hybrid atomic if for every commit set S of H, $H \mid S$ is serializable in every linearizing order L such that known(H) $\subseteq L$.

[1] Weihl's original formulation of hybrid atomicity [43] included a distinct serialization mechanism for read-only transactions, a distinction we do not make here.

[2] Techniques that use locking without timestamps satisfy a restricted form of hybrid atomicity called *strong dynamic atomicity* [44] whose availability properties are analyzed elsewhere [23].

Since precedes(H) $\subseteq ts$, known(H) partially orders sets of siblings. The on-line condition captures the requirement that the scheduler cannot know in advance how active transactions will be ordered when and if they commit, thus it must ensure that any linearizing order compatible with known(H) yields a legal serialization.

Since each $H \mid x$ is serializable in the linearizing order ts, Lemma 2 implies that hybrid atomicity is a local atomicity property:

THEOREM 17. *If $H \mid x$ is hybrid atomic for all x, then H is atomic.*

Let Hybrid(T, ts), Hybrid*(T, ts), and *hybrid dependency relation* be defined by analogy to the static atomic case. An argument almost identical to that used for Lemma 14 yields:

LEMMA 18. *If ts and ts' are distinct timestamp orders, then Hybrid(T, ts) and Hybrid(T, ts') are identical up to renaming of transactions.*

We omit explicit mention of ts except when necessary. The *hybrid serialization* of a schedule H is a serialization of $H \mid S$ in an order consistent with known(H), for some commit set S of H. H is in Hybrid(T) if and only if all its hybrid serializations are legal.

4.2 PRELIMINARY LEMMAS. To quantify over the local atomicity properties used in this paper, we use *Local*(T) to stand for any one of Serial(T), Static(T), or Hybrid(T), and similarly for *Local**(T), *local dependency relation*, and *local serialization.*

Let G and H be schedules in Local(T), Q a transaction, and q an operation.

Definition 19. G is a *false* **D**-*view* of H for q in Local(T) if G is a **D**-view of H for q, $G \cdot \langle qQ \rangle$ is in Local(T), but $H \cdot \langle qQ \rangle$ is not. By Definition 5, **D** is a dependency relation for Local(T) if and only if it has no false views.

We often omit mention of Local(T) when it is clear from context.

LEMMA 20. *If* **D** *is not a dependency relation for Local(T), then Local(T) includes schedules G and H such that G is a false* **D**-*view of H for some operation q and G is missing exactly one operation step of H.*

PROOF. We first remark that if G is a false **D**-view of H for q missing a single step, then the missing step must be an operation step, not a commit or abort step, because otherwise every local serialization of $H \cdot \langle qQ \rangle$ would also be a local serialization of $G \cdot \langle qQ \rangle$, implying that every local serialization of $H \cdot \langle qQ \rangle$ is legal, a contradiction.

Since **D** is not a dependency relation for Local(T), there exists a false **D**-view G of H for some operation q. Suppose G is missing k steps of H. Consider the sequence of schedules $\{H_i \mid i = 0, \ldots, k\}$, where $H_0 = G$, $H_k = H$, and H_{i+1} is constructed from H_i by restoring its earliest missing step.

$G \cdot \langle qQ \rangle$ is in Local(T) but $H \cdot \langle qQ \rangle$ is not, so there is some index i such that $H_i \cdot \langle qQ \rangle$ is in Local(T) but $H_{i+1} \cdot \langle qQ \rangle$ is not. Let $G_0 \cdot \langle pP \rangle \cdot G_1 \cdot \langle rR \rangle$ be the shortest prefix of $H_{i+1} \cdot \langle qQ \rangle$ not in Local(T), where $\langle pP \rangle$ is the operation step inserted in H_i to produce H_{i+1}. The schedule $G_0 \cdot G_1 \cdot \langle rR \rangle$ is in Local(T) as a prefix of H_i, $G_0 \cdot \langle pP \rangle \cdot G_1$ is in Local(T) by construction, but $G_0 \cdot \langle pP \rangle \cdot G_1 \cdot \langle rR \rangle$ is not. Because H_i is a **D**-closed subschedule of H_{i+1}, $G_0 \cdot G_1$ is a false **D**-view of $G_0 \cdot \langle pP \rangle \cdot G_1$ for r, proving the lemma.

4.3 COMPARISONS. Although serial atomicity places the strongest restrictions on concurrency of any local atomicity property considered here, we now show that it places the weakest restrictions on availability. This flexibility arises because any quorum assignment that does not guarantee serial atomicity also does not guarantee any other local atomicity property.

THEOREM 21. *Local*(T) $\subseteq$ Serial*(T).*

PROOF. We show that every local dependency relation is a serial dependency relation; thus, any quorum intersection relation that ensures local atomicity also ensures serial atomicity.

Suppose not. Pick a relation **D** that is a local dependency relation but not a serial dependency relation. Because **D** is not a serial dependency relation, there exists an operation q, a transaction Q, and serial schedules G, H, $G' = G \cdot \langle qQ \rangle$, and $H' = H \cdot \langle qQ \rangle$, such that G is a false **D**-view of H for q. Because H' is serial atomic, precedes(H') partially orders sets of siblings. If we choose a timestamp order ts compatible with the precedence order, so that precedes(H') $\subseteq ts(H')$, then G, H, and G' are in Local(T) (i.e., in Hybrid(T) or Static(T)), G is a **D**-view of H for q, but H' is not in Local(T), thus **D** fails to satisfy Definition 5 and cannot be a local dependency relation, a contradiction. □

Serial*(T) thus provides an "upper bound" on flexibility of quorum assignment for these local atomicity properties. We now show that static atomicity provides a corresponding "lower bound." We start with some definitions and lemmas.

Definition 22. The relation $\mathbf{D}_S$ between invocations and operations is defined as follows: $(q, p) \in \mathbf{D}_S$ if there exist histories h_1, h_2, and h_3 such that $h_1 \cdot h_2 \cdot h_3$ is legal, and either:

(1) $h_1 \cdot p \cdot h_2 \cdot h_3$ and $h_1 \cdot h_2 \cdot q \cdot h_3$ are legal, but $h_1 \cdot p \cdot h_2 \cdot q \cdot h_3$ is illegal, or

(2) $h_1 \cdot q \cdot h_2 \cdot h_3$ and $h_1 \cdot h_2 \cdot p \cdot h_3$ are legal, but $h_1 \cdot q \cdot h_2 \cdot p \cdot h_3$ is illegal.

We now show that $\mathbf{D}_S$ is a local dependency relation by deriving a contradiction from the assumption that a false $\mathbf{D}_S$-view exists for some operation.

LEMMA 23. *$\mathbf{D}_S$ is a local dependency relation.*

PROOF. Suppose not. Let G be a false $\mathbf{D}_S$-view of H for an operation q. By Lemma 20, we may assume G is missing a single operation step $\langle pP \rangle$. For brevity, we consider only the case where the illegal static serialization of $H \cdot \langle qQ \rangle$ has the form $h_1 \cdot p \cdot h_2 \cdot q \cdot h_3$. The histories $h_1 \cdot h_2 \cdot h_3$, $h_1 \cdot p \cdot h_2 \cdot h_3$, and $h_1 \cdot h_2 \cdot q \cdot h_3$ are legal as static serializations of G, H, and $G \cdot \langle qQ \rangle$, implying that $(q, p) \in \mathbf{D}_S$ (Definition 22; Property 1), which contradicts the assumption that G is a $\mathbf{D}_S$-view of H for q. □

We now show that $\mathbf{D}_S$ is the unique minimal static dependency relation by showing that any relation **D** that does not include $\mathbf{D}_S$ has a false **D**-view in Static(T).

LEMMA 24. *Every static dependency relation contains $\mathbf{D}_S$.*

PROOF. Let **D** be a relation that does not contain $\mathbf{D}_S$. **D** fails to satisfy either Property (1) or (2) of Definition 22. For brevity, we consider only Property (1). If h is a history, let $\langle hA \rangle$ denote the schedule in which each object-operation pair in h is associated with transaction A. Let h_1, h_2, h_3, p, and q be histories and operations satisfying Property (1) of Definition 22, let A, B, C, D, and E be transactions such that ts orders A before B before C before D before E, and let H be the following schedule:

$$\begin{gathered} h_1 A \\ \text{commit } A \\ h_2 C \\ \text{commit } C \\ h_3 E \\ \text{commit } E \\ pB \end{gathered}$$

Let G be the schedule that includes all but the last step. By construction, G and $G \cdot \langle qD \rangle$ are in Static(T), but $H \cdot \langle qD \rangle$ is not, thus G is a false **D**-view of H for q (Definition 19), and **D** cannot be a static dependency relation.

From Lemmas 23 and 24, and the observation that $\mathbf{D}_S$ is symmetric:

COROLLARY 25. *$\mathbf{D}_S$ is the unique minimal element of Static*(T).*

THEOREM 26. *Static*(T) $\subseteq$ Local*(T).*

Theorems 21 and 26 give us the following hierarchy of restrictions on availability:

$$\text{Static}^*(T) \subseteq \text{Hybrid}^*(T) \subseteq \text{Serial}^*(T). \tag{1}$$

Since Hybrid(T) and Static(T) are typically incomparable, this result illustrates our claim that analyzing restrictions on availability permits comparisons that analyzing restrictions on concurrency does not.

4.4 EXAMPLES. We now present examples of data types for which the inclusions in Hierarchy (1) are strict.

Example 27. There exists a T such that Serial*(T) $\not\subseteq$ Hybrid*(T).

A *DoubleBuffer* consists of a *producer* buffer and a *consumer* buffer, each capable of holding a single item. The object is initialized with a default item in each buffer. The data type provides

```
 Produce = Operation(Item)
Transfer = Operation( )
 Consume = Operation( ) Returns (Item)
```

Produce copies an item into the producer buffer, *Transfer* copies the item currently in the producer buffer to the consumer buffer, and *Consume* returns a copy of the item currently in the consumer buffer. DoubleBuffer has two distinct minimal quorum intersection relations, $\mathbf{Q}_1$ and $\mathbf{Q}_2$, shown in Figures 3 and 4.[3] As discussed in more detail elsewhere [24], Produce entries can appear in the view constructed for a Consume either because the quorums for Produce and Consume intersect directly, or because they intersect indirectly through Transfer.

Hybrid*(DoubleBuffer), however, has only one minimal element, shown in Figure 5. To see why $\mathbf{Q}_1$ is not in Hybrid*(DoubleBuffer), let H be the following

[3] $\mathbf{Q}_1$ and $\mathbf{Q}_2$ are not *minimal* serial dependency relations, although they are the symmetric closures of minimal relations.

	Produce	Transfer	Consume
Produce		X	
Transfer	X		X
Consume		X	

FIG. 3. First Minimal Element of Serial* (DoubleBuffer).

	Produce	Transfer	Consume
Produce			X
Transfer			X
Consume	X	X	

FIG. 4. Second Minimal Element of Serial* (DoubleBuffer).

	Produce	Transfer	Consume
Produce		X	X
Transfer	X		X
Consume	X	X	

FIG. 5. Minimal Element of Hybrid* (DoubleBuffer).

schedule,

$$\begin{array}{c}\text{Produce}(x)/\text{Ok}(\,)\ A\\ \text{Transfer}(\,)/\text{Ok}(\,)\ A\\ \text{Commit}\ A\\ \text{Transfer}(\,)/\text{Ok}(\,)\ C\\ \text{Consume}(\,)/\text{Ok}(x)\ D\end{array}$$

and let G be the subschedule consisting of all but the last step. G is a Q_1-view of H for Produce; G, H, and $G \cdot \langle \text{Produce}(y)/\text{Ok}(\,)\ B\rangle$ are in Hybrid(DoubleBuffer); $H \cdot \langle \text{Produce}(y)/\text{Ok}(\,)\ B\rangle$ is not in Hybrid(DoubleBuffer), however, because an illegal serialization results if transactions are serialized in the order A, B, C, and D, it follows that G is a false Q_1-view of H for Produce; hence, Q_1 is not a hybrid dependency relation for DoubleBuffer, nor is it in Hybrid*(DoubleBuffer). Reversing the order of the last two steps in H yields a false Q_2-view for Produce, illustrating that Q_2 is not in Hybrid*(DoubleBuffer).

Example 27 illustrates our claim that concurrency and availability are not independent properties. For a DoubleBuffer replicated among n identical repositories, Serial*(DoubleBuffer) permits n distinct quorum assignments. The range of quorum assignments for a DoubleBuffer replicated among five identical repositories is shown in Figure 6. Each column represents an alternative quorum assignment, and an entry of n indicates that any n out of five repositories constitutes a quorum for the operation. The quorum assignment in the first column favors the availability of Consume and Produce, while the last column favors Produce and Transfer.

FIG. 6. Quorum Assignments for 5-Site Serial Atomic DoubleBuffer.

Consume	1	2	3	4	5
Transfer	5	4	3	2	1
Produce	1	2	3	2	1

By contrast, Hybrid*(DoubleBuffer) permits only one quorum assignment: All operations require majority quorums. In this example, increasing concurrency reduces the range of permissible quorum assignments.

Example 28. There exists a T such that Hybrid*(T) $\not\subset$ Static*(T).

A *Prom* is a container for an item. When a Prom is created, it is initialized with a default value, and its contents can be overwritten, but not read. Once the Prom has been *sealed*, its contents can be read but not written. Prom provides three operations:

```
Write = Operation(item)
```

stores a new item in the Prom. A Write may occur only if the Prom has not been sealed.

```
Read = Operation( ) Returns(item)
```

returns the item in the Prom. A Read may occur only if the Prom has been sealed.

```
Seal = Operation( )
```

enables Reads and disables Writes. It has no effect if the Prom has already been sealed.

Corollary 25 implies that the unique minimal element of Static*(Prom) is the relation $\mathbf{D}_S$ shown in Figure 7. We now argue that the relation $\mathbf{D}_H$ in Figure 8 is a hybrid dependency relation for Prom. Since it is symmetric, it is also an element of Hybrid*(Prom). Because no Write operation can follow a Seal in any hybrid serialization:

PROPERTY 29. *If $H \cdot \langle Write(x)/Ok(\,)Q\rangle$ is in Hybrid(Prom), then H includes no Seal operations executed by active or committed transactions.*

Because every Read operation must follow a Seal in every hybrid serialization:

PROPERTY 30. *If $H \cdot \langle Read(\,)/Ok(x)Q\rangle$ is in Hybrid(Prom), then H includes a Seal operation executed by an active or committed transaction.*

If $\mathbf{D}_H$ is not a hybrid dependency relation, then by Lemma 20, some operation q has a false $\mathbf{D}_H$-view G of H missing a single operation step $\langle pP\rangle$. Moreover, since $\mathbf{D}_S$ is in Hybrid*(T) by Theorem 26, G cannot be a $\mathbf{D}_S$-view of H for q. The rest is a case analysis on possible values for q. For brevity, assume that H includes no abort steps.

—Since every $\mathbf{D}_H$-view of H for Seal is also a $\mathbf{D}_S$-view, q cannot be Seal.

—If q is Read, then p must be Write, and H has the form $G_1 \cdot \langle \text{Write}(x)/\text{Ok}(\,)P\rangle \cdot G_2$. By Property 29, G_1 includes no Seal operations. By Property 30, G must include a Seal; hence, G_2 must include a Seal. But Seal and Write are related by $\mathbf{D}_H$, contradicting the assumption that G is $\mathbf{D}_H$-closed.

—If q is Write, then p must be Read, and H has the form $G_1 \cdot \langle \text{Read}(\,)/\text{Ok}(x)P\rangle \cdot G_2$. Because G is in Hybrid(Prom), Property 29 implies that G_1, and hence G, includes a Seal operation. Because $G \cdot \langle qQ\rangle$ is in Hybrid(T), however, Property 30 implies that G does not include a Seal operation, a contradiction.

	Write	Seal	Read
Write		X	X
Seal	X		X
Read	X	X	

FIG. 7. $\mathbf{D}_S$ in Static*(Prom).

	Write	Seal	Read
Write		X	
Seal	X		X
Read		X	

FIG. 8. $\mathbf{D}_H$ in Hybrid*(Prom).

Write	1	2	3	4	5
Seal	5	4	3	2	1
Read	1	2	3	4	5

FIG. 9. Quorum Assignments for 5-Site Hybrid Atomic Prom.

Static*(Prom) requires Read and Write quorums to intersect, while Hybrid*(Prom) does not. These additional restrictions translate directly into restrictions on availability, as illustrated by Figure 9. Each column shows a quorum assignment for a hybrid atomic Prom replicated among five identical repositories. The quorum assignments shown in the first two columns, however, do not guarantee static atomicity. This result suggests that hybrid atomicity is more promising than static atomicity in distributed systems where availability is important.

Example 31. There exists a T such that Serial*(T) $\not\subset$ Hybrid*(T) $\not\subset$ Static*(T).

A *CrossProduct* is constructed from a DoubleBuffer and a Prom. The object's set of operations is the union of the DoubleBuffer and Prom operations, and a CrossProduct history is legal if the subsequence of DoubleBuffer operations is a legal DoubleBuffer history, and the subsequence of Prom operations is a legal Prom history. Since DoubleBuffer operations do not interact with Prom operations, Local*(CrossProduct) = Local*(DoubleBuffer) ∪ Local*(Prom).

4.5 BOUNDED TRANSACTION NESTING. Nested transactions enhance concurrency by permitting a transaction to be decomposed into parallel subtransactions. Nested transactions also facilitate fault-tolerance, since a subtransaction can be aborted without aborting its parent. Nevertheless, we show here that under hybrid atomicity, single-level transaction systems place fewer restrictions on availability than nested transaction systems. Moreover, in systems in which the depth of transaction nesting is bounded, restrictions on availability tighten as the maximum depth of transaction nesting increases. The restrictions on availability induced by serial and static atomicity are insensitive to transaction nesting.

A transaction's *depth of nesting* is defined as follows: the root transaction has depth 0, and every other transaction's depth is one greater than its parent's. Let Hybrid$_n$(T) be the subset of Hybrid(T) consisting of schedules in which no transaction's depth exceeds n. The bounded-depth hybrid atomic concurrent specifications for T form a strict infinite hierarchy with respect to concurrency:

$$\text{Hybrid}_1(T) \subset \text{Hybrid}_2(T) \subset \cdots \subset \text{Hybrid}(T).$$

The greater the maximum depth of transaction nesting, the greater the set of permissible interleavings.

LEMMA 32. *Any dependency relation for Hybrid$_{n+1}(T)$ is also a dependency relation for Hybrid$_n(T)$.*

PROOF. We show that if **D** is not a dependency relation for Hybrid$_n(T)$, then it is not a dependency relation for Hybrid$_n(T)$. Suppose not. Let G be a false **D**-view of H for q in Hybrid$_n(T)$, but not in Hybrid$_{n+1}(T)$. By Definition 19, there exists a transaction Q such that $G \cdot \langle qQ \rangle$ is in Hybrid$_n(T)$, while $H \cdot \langle qQ \rangle$ is in Hybrid$_{n+1}(T)$ but not in Hybrid$_n(T)$. Since $H \cdot \langle qQ \rangle$ is in Hybrid$_{n+1}(T)$ but not in Hybrid$_n(T)$, some transaction is nested to depth $n + 1$. Since H is in Hybrid$_n(T)$, that transaction must be Q, but since $G \cdot \langle qQ \rangle$ is in Hybrid$_n(T)$, it cannot be Q, a contradiction. □

It follows that the bounded-depth hybrid atomic specifications for T also form an infinite hierarchy with respect to availability.

$$\text{Hybrid}^*(T) \subseteq \cdots \subseteq \text{Hybrid}_2^*(T) \subseteq \text{Hybrid}_1^*(T). \tag{2}$$

Here, however, the ordering of the hierarchy is reversed: the greater the maximum depth of transaction nesting, the smaller the set of permissible quorum assignments. In this section, we give an example showing that this hierarchy is strict.

To show the inclusions in Eq. (2) are strict, we need the following lemma.

LEMMA 33. *If operation p precedes operation q in every history of T, and $\langle pP \rangle$ and $\langle qQ \rangle$ are operation steps of a schedule H in Hybrid(T), where P and Q are in active$(H) \cup$ committed(H), then laa$(P) \in$ anc(laa(Q)).*

PROOF. First note that $(\langle pP \rangle, \langle qQ \rangle) \in$ known(H), because otherwise known(H) could be extended to a linearizing order serializing q before p, contradicting our assumption about T. If $(\langle pP \rangle, \langle qQ \rangle) \in$ known(H), then there exists siblings $P' \in$ anc(P) and $Q' \in$ anc(Q) such $(P', Q') \in$ known(H). By Definition 16, P' must be committed; hence, laa$(P) \in$ proper-anc(P'). If lca$(P, Q) \in$ committed(H), then laa(P) = laa(Q); otherwise, laa(P) = lca$(P, Q) \in$ anc(laa(Q)). □

Let t_n be the tree shown in Figure 10 whose nodes are labeled with operations. (Here, left(i) and Right(i) are shorthand for Left( )/OK(i) and Right( )/Ok(i).) A *partial preorder traversal* of t_n is defined as follows:

(1) Visit the root.
(2) Perform a partial preorder traversal on the subtrees rooted at zero, one, or both of the root's children.

The history generated by a partial preorder traversal is the history constructed by concatenating the nodes of t_n in the order visited.

Definition 34. The serial specification Tree$_n$ consists of all histories generated by partial preorder traversals of t_n except for the "forbidden" history:

$$\text{Left}(n) \cdot \text{Left}(n-1) \cdot \cdots \cdot \text{Left}(0) \cdot \text{Right}(0) \cdot \cdots \cdot \text{Right}(n-1) \cdot \text{Right}(n),$$

generated by the complete left-to-right preorder traversal.

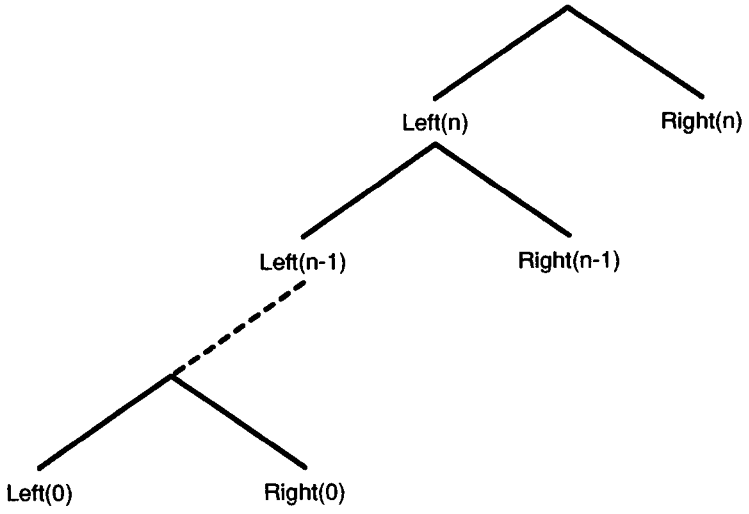

FIG. 10. The Tree T_n.

Note that these serial specifications are *partial*: responses are undefined for certain invocations.

Let **D** be the quorum intersection relation encompassing all pairs of operations except (Right(0), Left (0)) and (Left(0), Right(0)).

LEMMA 35. **D** *is not in* $\text{Hybrid}^*_{n+1}(\text{Tree}_n)$.

PROOF. Construct a transaction tree isomorphic to t_n, where transaction L_i corresponds to Left(i) and R_i corresponds to Right(i), for $0 \le i \le n$. Let H be the following schedule,

$$\begin{array}{c} \text{Left}(n)\,L_n \\ \text{Right}(n)\,R_n \\ \vdots \\ \text{Left}(1)\,L_1 \\ \text{Right}(1)\,R_1 \\ \text{Left}(0)\,L_0 \end{array}$$

and let G be the prefix of H that omits the last step. G is a false **D**-view of H for Right(0), since G, H, and $G \cdot \langle \text{Right}(0)\,R_n \rangle$ are in $\text{Hybrid}_{n+1}(\text{Tree}_n)$, but $H \cdot \langle \text{Right}(0)\,R_n \rangle$ is not, since the forbidden history is one of its hybrid serializations. □

Example 36. $\text{Hybrid}^*_n(\text{Tree}_n) \not\subseteq \text{Hybrid}^*_{n+1}(\text{Tree}_n)$.

We show that **D** is in $\text{Hybrid}^*_n(\text{Tree}_n)$ but not $\text{Hybrid}^*_{n+1}(\text{Tree}_n)$ by showing that if G is a false **D**-view of H for some operation, then H includes a transaction nested deeper than n.

Since **D** is not a dependency relation for $\text{Hybrid}_{n+1}(\text{Tree}_n)$ (Lemma 35), there exist G, H, and q such that G is a false **D**-view of H for q. Let L_i and R_i be the transactions that, respectively, executed Left(i) and Right(i) in these schedules. Because Lemma 20 makes no assumptions about the depth of transaction nesting, we may assume G is missing exactly one step $\langle p\,P \rangle$ of H. Since Right(0) and

Left(0) are the only operations not related by **D**, either q = Right(0) and p = Left(0), or vice-versa. Moreover, since G is **D**-closed, H has the form $G \cdot \langle pP \rangle$.

$H \cdot \langle qQ \rangle$ can fail to be in Tree$_n$ in two ways: (1) it may have a hybrid serialization that is not a partial preorder traversal of t_n, or (2) it may have the forbidden history as a hybrid serialization. We claim the first case is impossible. The illegal serialization of $H \cdot \langle qQ \rangle$ induces legal serializations of $G \cdot \langle pP \rangle$ and $G \cdot \langle qQ \rangle$ that must have the form $h_1 \cdot$ Left(1) $\cdot\ p \cdot h_2$ and $h_1 \cdot$ Left(1) $\cdot\ q \cdot h_2$, since every traversal visits Left(1) immediately before Left(0) or Right(0). Because Left(1) appears only once in H, the illegal serialization of $H \cdot \langle qQ \rangle$ must have the form $h_1 \cdot$ Left(1) $\cdot\ p \cdot q \cdot h_2$ or $h_1 \cdot$ Left(1) $\cdot\ q \cdot p \cdot h_2$, each of which is a partial preorder traversal of t_n.

We now show that if $H \cdot \langle qQ \rangle$ has the forbidden history as a serialization, then some transaction is nested to a depth greater than n. Step 1 is to show that laa(L_i) = lca(L_i, R_0), for $i > 0$. Since Left(i) precedes Right(0) in every history in Tree$_n$, laa(L_i) $\in$ anc(laa(R_0)) by Lemma 33. Because R_0 is active, laa(L_i) = lca(L_i, R_0).

Step 2 is to show that laa(L_{i+1}) $\in$ anc(R_i). Since Left(i + 1) precedes Right(i) in every history in Tree$_n$, laa(L_{i+1}) $\in$ anc(laa(R_i)) by Lemma 33; hence, laa(L_{i+1}) $\in$ anc(R_i).

Step 3 is to show that laa(L_i) $\notin$ anc(R_i), for $i > 0$. Suppose otherwise. Since laa(L_i) = lca(L_i, R_0) by Step 1, there exist $L_i' \in$ anc(L_i), $R_i' \in$ anc(R_i), and $R_0' \in$ anc(R_0) such that parent(L_i') = parent(R_i') = parent(R_0') = lca(L_i, R_0), and L_i' is committed. Because Left(i) precedes Right(0) in every serialization, (L_i', R_0') $\in$ known(H), but because the forbidden history is a hybrid serialization of H, (R_i', L_i') $\notin$ known(H) and (R_i', R_0') $\notin$ known(H). There are two cases to consider; each leads to a contradiction. First, if lca(L_i, R_i) $\in$ proper-desc(laa(L_i, R_0)), then $L_i' = R_i'$ and (R_i', R_0') $\in$ known(H), ruling out the forbidden history as a hybrid serialization. Second, if lca(L_i, R_i) = lca(L_i, R_0), then $L_i' \neq R_i'$, and the ordering $L_i < R_i < R_0$ is compatible with known(H), an order that does not correspond to any partial preorder traversal of t_n.

The final step is to show that laa(L_i) $\in$ proper-desc(laa(L_{i+1})). By Lemma 33, laa(L_i) $\in$ desc(laa(L_{i+1})). Laa(L_{i+1}) is an ancestor of R_i (Step 2) and laa(L_i) is not (Step 3); therefore, laa(L_i) and laa(L_{i+1}) are distinct. Since laa(L_n) has depth at least 1, laa(L_0) has depth at least n + 1.

Static atomicity is insensitive to depth of transaction nesting: Static$^*_{n+1}(T)$ = Static$^*_n(T)$ = Static$^*(T)$. This result follows from the proof of Lemma 24, which actually shows a slightly stronger result: if **D** does not contain $\mathbf{D}_S$, then **D** is not a dependency relation for Static$_1(T)$.

5. *Conclusions*

Atomicity in a decentralized distributed system is ensured by choosing a local atomicity property that every atomic object must satisfy. For example, the Swallow distributed data storage system is based on static atomicity [41], Argus [30], and TABS [40] are based on dynamic atomicity, and Avalon [28] is based on hybrid atomicity. Choosing such a local atomicity property is a design decision of critical importance, since the decision must be taken in advance, and once made, it is difficult to change. An inappropriate choice may place unnecessary restrictions on the availability and concurrency realizable within the system.

This paper has introduced a new criterion for evaluating local atomicity properties: the constraints they impose on availability. In particular, our results suggest that hybrid atomicity is more promising than static atomicity as a foundation for highly available and highly concurrent distributed systems. Much work remains to be done in this area, however, since our analysis does not address such issues as implementation techniques or language support.

REFERENCES

1. ALSBERG, P. A., AND DAY, J. D. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd Annual Conference on Software Engineering* (Oct.). IEEE Press, Washington, D.C., 1976, pp. 627-644.
2. BEERI, C., BERNSTEIN, P. A., AND GOODMAN, N. A model for concurrency in nest transaction systems. Tech. Rep. TR-86-03. Wang Institute, Tyngsboro, Mass., Mar. 1986.
3. BERNSTEIN, P. A., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Comput. Surv. 13*, 2 (June 1981), 185–221.
4. BERNSTEIN, P. A., AND GOODMAN, N. The failure and recovery problem for replicated databases. In *Proceedings of the 2nd Annual Symposium on Principles of Distributed Computing* (Montreal, Que., Canada, Aug. 17–19). ACM, New York, 1983, pp. 114–122.
5. BERNSTEIN, P. A., AND GOODMAN, N. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. Datab. Syst. 9*, 4 (Dec. 1984), 596–615.
6. BERNSTEIN, P. A., GOODMAN, N., AND LAI, M. Y. Two-part proof schema for database concurrency control. In *Proceedings of the 5th Berkeley Workshop on Distributed Data Management and Computer Networks*, (Feb.). Lawrence Berkeley Laboratory, Berkeley, Calif., 1981, pp. 71–84.
7. BIRMAN, K. P. Replication and fault-tolerance in the ISIS system. In *Proceedings of the 10th Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1–4). ACM, New York, 1985, pp. 79–86.
8. BIRRELL, A. D., LEVIN, R., NEEDHAM, R., AND SCHROEDER, M. Grapevine: An exercise in distributed computing. *Commun. ACM 25*, 4 (Apr. 1982), 260–274.
9. BLOCH, J. J., DANIELS, D. S., AND SPECTOR, A. Z. A weighted voting algorithm for replicated directories. *J. ACM 34*, 4 (Oct. 1987), 859–909.
10. CHAN, A., FOX, S., LIN, W. T., NORI, A., AND RIES, D. The implementation of an integrated concurrency control and recovery scheme. In *Proceedings of the 1982 SIGMOD Conference: International Conference on Management of Data* (Orlando, Fla., June 2–4). ACM, New York, 1982, pp. 184–191.
11. COOPER, E. C. Circus: A replicated procedure call facility. In *Proceedings of the 4th Symposium on Reliability in Distributed Software and Database Systems* (Oct.). IEEE Press, Washington, D.C., 1984, pp. 11–24.
12. DUBOURDIEU, D. J. Implementation of distributed transactions. In *Proceedings of the 1982 Berkeley Workshop on Distributed Data Management and Computer Networks.* Lawrence Berkeley Laboratory, Berkeley, Calif., 1982, pp. 81–94.
13. EAGER, D. L., AND SEVCIK, K. C. Achieving robustness in distributed database systems. *ACM Trans. Datab. Syst. 8*, 3 (Sept. 1983), 354–381.
14. EL-ABBADI, A., AND TOUEG, S. Availability in partitioned replicated databases. Tech. Rep. TR 85-721. Dept. of Comput. Sci., Cornell University, Dec. 1985.
15. ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.
16. FISCHER, M., AND MICHAEL, A. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Los Angeles, Calif., Mar. 29–31). ACM, New York, 1982, pp. 7–75.
17. GIFFORD, D. K. Weighted voting for replicated data. In *Proceedings of the 7th Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec. 10–12). ACM, New York, 1979, pp. 150–162.
18. GOLDMAN, K. J., AND LYNCH, N. A. Quorum consensus in nested transaction systems. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 10–12). ACM, New York, 1987, pp. 27–41.
19. GRAY, J. N. Notes on database operating systems. In Lecture Notes in Computer Science, vol. 60. Springer-Verlag, Berlin, West Germany, 1978, pp. 393–481.
20. HAMMER, M. M., AND SHIPMAN, D. W. Reliability mechanisms for SDD-1: A system for distributed databases. *ACM Trans. Datab. Syst. 5*, 4 (Dec. 1980), 431–466.
21. HEDDAYA, A., HSU, M. C., AND WEIHL, W. E. Two-phase gossip: Managing distributed event histories. Tech. Rep. TR-04-87. Harvard Univ., Cambridge, Mass., Dec. 1987.

22. HERLIHY, M. P. Concurrency versus availability: Atomicity mechanism for replicated data. *ACM Trans. Comput. Syst. 5*, 3 (Aug. 1987), 249–274.
23. HERLIHY, M. P. Comparing how atomicity mechanisms support replication. In *Proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computing* (Minaki, Ont., Canada, Aug. 5–7). ACM, New York, 1985, pp. 102–110.
24. HERLIHY, M. P. A quorum-consensus replication method for abstract data types. *ACM Trans. Comput. Syst. 4*, 1 (Feb. 1986), 32–53.
25. HERLIHY, M. P. Extending multiversion timestamping protocols to exploit type information (special issue on parallel and distributed computing). *IEEE Trans. Comput. C-35*, 4 (Apr. 1987), 443–449.
26. HERLIHY, M. P. Dynamic quorum adjustment for partitioned data. *ACM Trans. Datab. Syst. 12*, 2 (June 1987), 170–194.
27. HERLIHY, M. P., AND WEIHL, W. E. Hybrid concurrency control for abstract data types. In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Austin, Tex., Mar. 21–23). ACM, New York, 1988, pp. 201–210.
28. HERLIHY, M. P., AND WING, J. M. Avalon: Language support for reliable distributed systems. In *Proceedings of the 17th Symposium on Fault-Tolerant Computer Systems* (July). IEEE Computer Society Press, Washington, D.C., 1987, pp. 89–95.
29. KORTH, H. F. Locking primitives in a database system. *J. ACM 30*, 1 (Jan. 1983), 55–79.
30. LISKOV, B., AND SCHEIFLER, R. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Prog. Lang. Syst. 5*, 3 (July 1983), 381–404.
31. LYNCH, N. A. Concurrency control for resilient nested transactions. In *Proceedings of the 2nd ACM Symposium on Principles of Database Systems* (Mar.). ACM, New York, 1983, pp. 166–181.
32. LYNCH, N. A., AND MERRITT, M. Introduction to the theory of nested transactions. Tech. Rep. MIT/LCS/TR-387. Laboratory for Computer Science. MIT, Cambridge, Mass., Apr., 1986.
33. MOSS, J. E. B. Nested transactions: An approach to reliable distributed computing. Tech. Rep. MIT/LCS/TR-260. Laboratory for Computer Science. MIT, Cambridge, Mass., Apr. 1981.
34. PAPADIMITRIOU, C. H. The serializability of concurrent database updates. *J. ACM 26*, 4 (Oct. 1979), 631–653.
35. PAPADIMITRIOU, C. H., AND KANELLAKIS, P. On concurrency control by multiple versions. *ACM Trans. Datab. Syst. 9*, 1 (Mar. 1984), 89–99.
36. REED, D. P. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst. 1*, 1 (Feb. 1983), 3–23.
37. SCHWARZ, P. M., AND SPECTOR, A. Z. Synchronizing shared abstract types. *ACM Trans. Comput. Syst. 2*, 3 (Aug. 1984), 223–250.
38. SKEEN, M. D. Crash recovery in a distributed database system. Ph.D. dissertation, Univ. California, Berkeley, Berkeley, Calif., May 1982.
39. SKEEN, D., CHRISTIAN, F., AND EL-ABBADI, A. An efficient, fault-tolerant protocol for replicated data management. In *Proceedings of the 4th ACM SIGACT-SIGMOD Conference on Principles of Database Systems* (Portland, Ore., Mar. 25–27). ACM, New York, 1985, pp. 215–229.
40. SPECTOR, A. Z., BUTCHER, J., DANIELS, D. S., DUCHAMP, D. J., EPPINGER, J. L., FINEMAN, C. E., HEDDAYA, A., AND SCHWARZ, P. M. Support for distributed transactions in the tabs prototype. *IEEE Trans. Softw. Eng. 11*, 6 (June 1985), 520–530.
41. SVOBODOVA, L. A reliable object-oriented repository for a distributed computer system. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec. 14–16). ACM, New York, 1981, pp. 47–58.
42. THOMAS, R. H. Consensus approach to concurrency control for multiple copy databases. *ACM Trans. Datab. Syst. 4*, 2 (June 1979), 180–209.
43. WEIHL, W. E. Data-dependent concurrency control and recovery. In *Proceedings of the 2nd Annual Symposium on Principles of Distributed Computing* (Montreal, Que., Canada, Aug. 17–19). ACM, New York, 1983, pp. 63–75.
44. WEIHL, W. E. Specification and implementation of atomic data types. Tech. Rep. TR-314. Laboratory for Computer Science. M.I.T., Cambridge, Mass., Mar. 1984.

RECEIVED MARCH 1986; REVISED JUNE 1987, JULY 1988, AND MAY 1989; ACCEPTED MAY 1989