CMU-CS-86-110

Concurrency and Availability as Dual Properties of Replicated Atomic Data

Maurice Herlihy
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213
19 February 1986

Abstract

A replicated object is a typed object that provides a set of operations to its clients. A quorum for an operation is any set of sites whose co-operation suffices to execute that operation. An operation's quorums determine its availability, and constraints on quorum assignment determine the range of availability properties realizable by replication. This paper compares the constraints on availability and concurrency imposed by three classes of atomicity mechanisms, respectively encompassing generalized two-phase locking, multiversion timestamping, and hybrid techniques. An analysis of the constraints on quorum assignment necessary to maximize the concurrency permitted within each class shows that only the class encompassing hybrid mechanisms is undominated: (1) Hybrid schemes permit more quorum assignments than timestamping schemes, even though they place incomparable constraints on concurrency. (2) Hybrid and locking schemes place incomparable constraints on quorum assignment, even though hybrid schemes support more concurrency. (3) Timestamping and locking schemes are incomparable with respect to both concurrency and quorum assignment. (4) Bounding the maximum depth of transaction nesting tightens constraints on concurrency for all three classes, but reduces the constraints on quorum assignment for hybrid schemes only. (5) Bounding the number of versions retained by multiversion timestamping schemes reduces concurrency but enhances availability.

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 NOOO39-85-C-0134.

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

1. Introduction

Most pessimistic mechanisms for implementing atomicity in distributed systems fall into three broad categories: two-phase locking schemes (e.g. [12, 20, 29]), timestamping schemes (e.g. [28, 27, 26]), and hybrid schemes employing both locking and timestamps (e.g. [6, 8, 1, 2]). This paper proposes a new criterion for evaluating these mechanisms: the constraints they impose on the availability of replicated data. Our results suggest that hybrid schemes may be the most promising as a basis for constructing highly available and highly concurrent distributed systems.

Our analysis of availability is based on *quorum consensus* replication [19, 17]. A replicated object is a typed object that provides a collection of operations to its clients. Associated with each operation is a set of *quorums*, which are collections of sites whose cooperation suffices to execute the operation. An operation's quorums determine its availability, and constraints on quorum assignment determine the range of availability properties realizable by replication. An analysis of the object's data type specification yields a set of constraints on quorum assignment necessary and sufficient to ensure the correctness of the replicated implementation. This replication method 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.

The three-part classification of atomicity mechanisms is formalized using a model developed by Weihl [32], generalized here to encompass nested transactions. Each category is identified with a local property of objects that suffices to ensure the atomicity of a system encompassing multiple objects. *Static atomicity* encompasses the timestamping mechanisms cited above, *strong dynamic atomicity* encompasses the locking mechanisms, and *hybrid atomicity* encompasses the hybrid mechanisms. These properties are type-specific; constraints on concurrency are expressed in terms of the abstract operations provided by the data type, not in terms of primitive read and write operations. Hybrid and static atomicity support incomparable levels of concurrency, hybrid atomicity permits more concurrency than strong dynamic atomicity, and static and strong dynamic atomicity support incomparable levels of concurrency. These relations are shown in Figure 1-1.

The constraints on concurrency and availability (i.e. quorum assignment) cannot be minimized simultaneously [17]. At one extreme in the availability/concurrency trade-off, all three properties support the same minimal set of constraints on quorum assignment, each at suboptimal levels of concurrency. This paper considers the other extreme, comparing the constraints on quorum assignment necessary to realize the optimal level of concurrency permitted by each property. In practice, optimal schedulers are unlikely to be cost-effective, since the complexity of recognizing all permissible interleavings may outweigh the benefit of the additional concurrency (c.f. [25, 26]). 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, and once made, it is difficult to change.

This paper presents the following results:

1. Although hybrid and static atomicity are incomparable with respect to concurrency, they are comparable with respect to quorum assignment: any quorum assignment that supports full static atomicity also supports full hybrid atomicity, but not necessarily vice-versa. Thus, maximizing concurrency under hybrid atomicity permits a wider range of availability trade-offs than under static atomicity.

2. Although hybrid atomicity permits strictly more concurrency than strong dynamic atomicity, they are incomparable with respect to concurrency; each supports quorum assignments the other does not.

3. Static and strong dynamic atomicity are incomparable with respect to both concurrency and availability.

4. Under hybrid atomicity, constraints on quorum assignment can be relaxed by bounding the maximum depth of transaction nesting. The smaller the bound, the larger the set of permissible quorum assignments. By contrast, quorum assignments for static and dynamic atomicity are unaffected by transaction nesting.

5. Under static atomicity, constraints on quorum assignment can be relaxed by bounding the number of out-of-date object versions retained. The fewer the versions, the larger the set of permissible quorum assignments.

These relations are illustrated schematically in Figures 1-2, 1-3, and 1-4.

These results show that there is a complex relation between the availability and concurrency supported by various atomicity properties. Availability and concurrency are neither completely independent nor completely dependent: one cannot simultaneously minimize the constraints on both, but tightening the constraints on one does not necessarily relax the constraints on the other. Instead, we argue that availability and concurrency are best considered as complementary properties, each permitting comparisons the other does not. A complete analysis of an atomicity property should take both into account. Finally, of the properties considered here, only hybrid atomicity is undominated with respect to both availability and concurrency, suggesting that hybrid techniques merit further scrutiny as a basis for implementing atomicity in distributed systems supporting high levels of availability and concurrency.

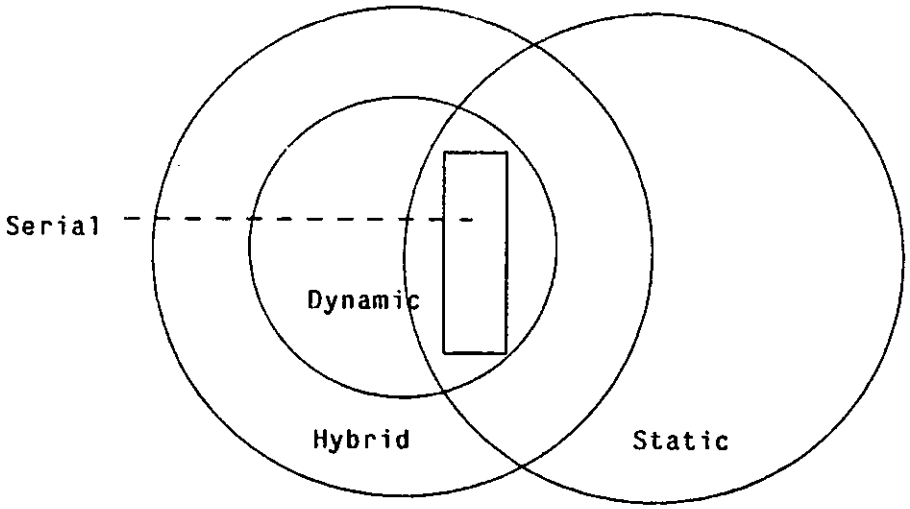

Figure 1-1: Concurrency

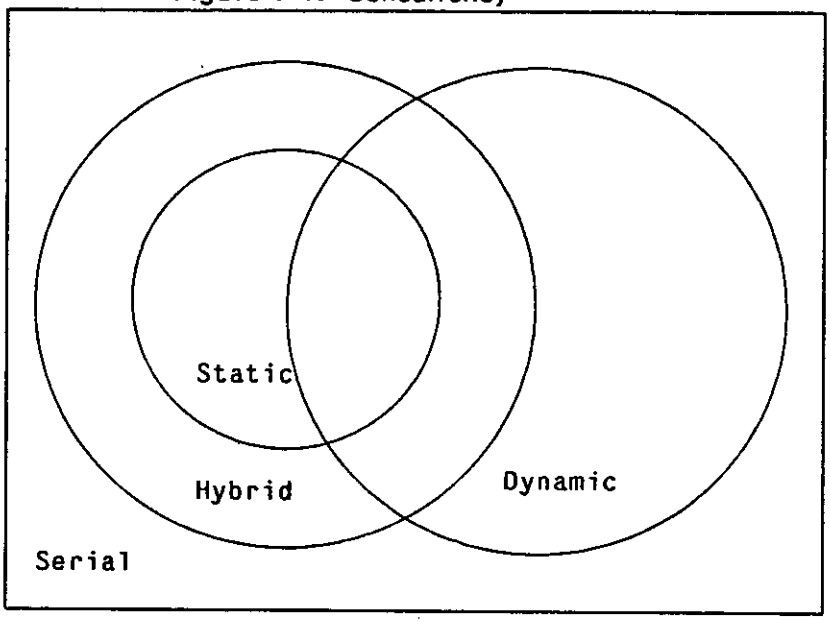

Figure 1-2: Availability

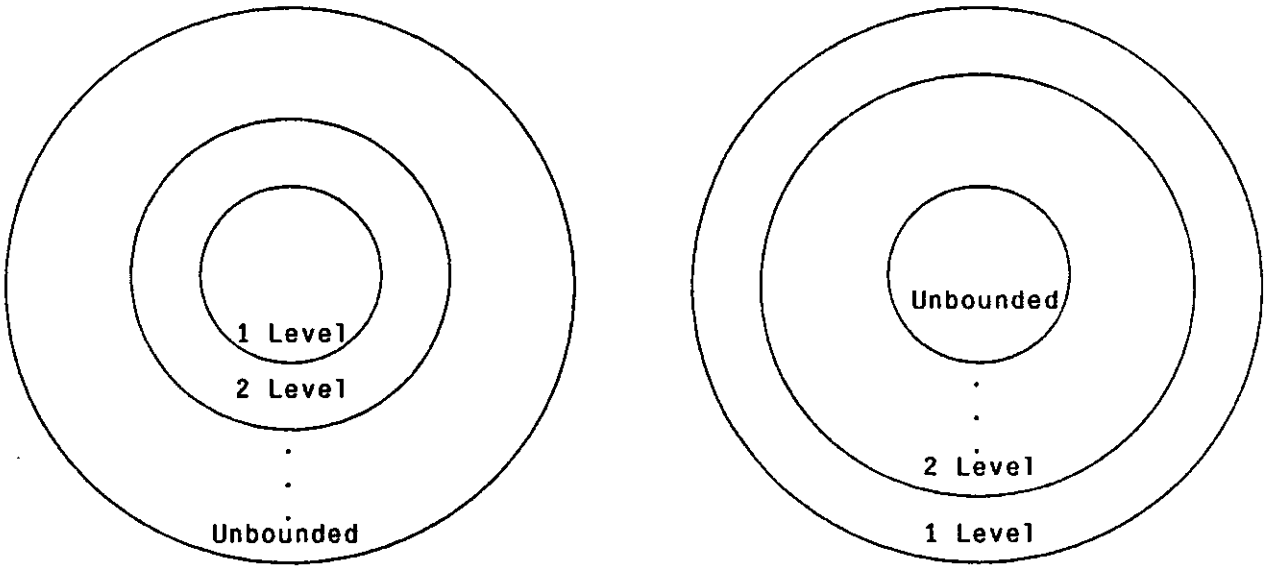

Figure 1-3:
Hybrid Atomicity: Concurrency (L) and Availability (R) for Nested Transactions

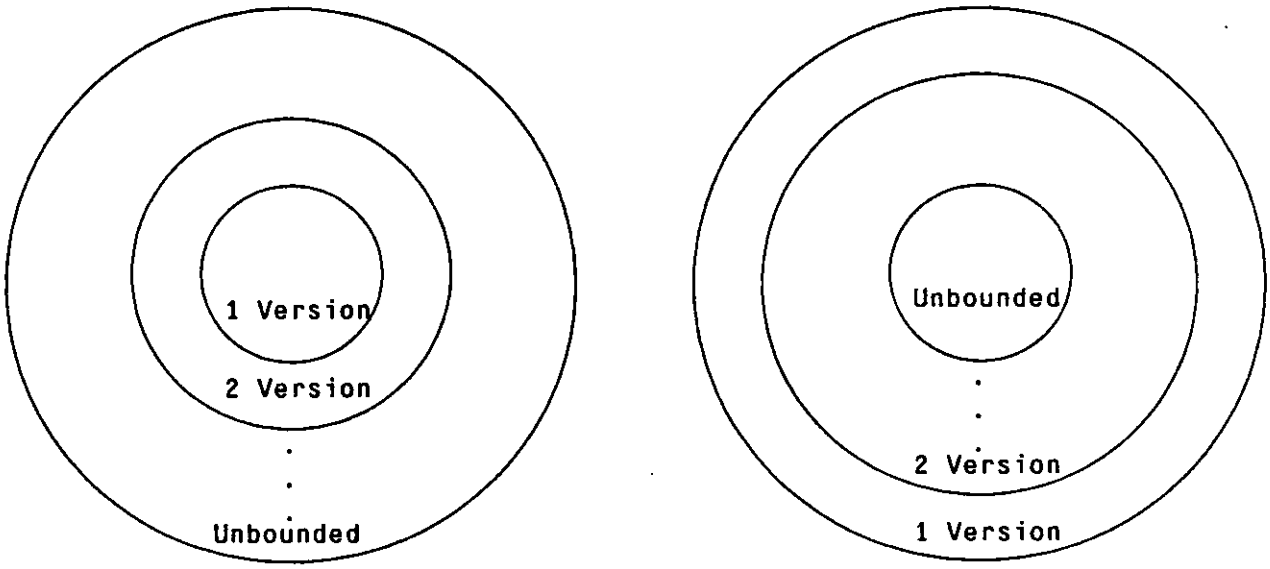

Figure 1-4:
Static Atomicity: Concurrency (L) and Availability (R) for Bounded Versions

2. Related Work

Numerous replication methods for files have been proposed. Methods that preserve serializability in the presence of site crashes but not communication failures include SDD-1 [16], Available Copies [15], and Circus [7]. Methods that tolerate partitions include those proposed for the ISIS system [4], by Gifford [14], by Eager and Sevcik [9], by Abbadi, Skeen, and Cristian [10], and by Abbadi and Toueg [11].

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, Daniels, and Spector [5]. The quorum-consensus file replication methods of Gifford and of Eager and Sevcik can be generalized to exploit type information [19, 18]. The replication method considered here is the *consensus scheduling* method of [17], which is described in more detail below. Garcia-Molina and Barbara [13] have proposed criteria for evaluating the fault-tolerance provided by quorum consensus methods.

A formal model for replicated databases proposed by Bernstein and Goodman [3] can be used to show the correctness of several file replication methods, but it is not immediately applicable to techniques in which replicated objects are not represented by multiple copies, and information about operations cannot be captured by a simple read/write classification. As discussed above, the atomicity properties investigated here were identified by Weihl [31]. Our generalization of these properties to nested transaction systems employs formalisms proposed by Lynch [23]. Our discussion of bounded versions draws on work of Papadimitriou and Kannelakis [26].

3. Model of Computation

3.1. Transactions

A *distributed system* consists of multiple computers (called sites) that communicate through a network. The physical components of a distributed system can fail independently: sites can crash, and communication links can be interrupted. Nonetheless, the data managed by a distributed program may be subject to consistency constraints that must be preserved in the presence of failures and concurrency. Such constraints can apply not only to individual pieces of data, but also to

distributed sets of data. For example, a distributed banking system might be subject to the constraint that the books balance: money is neither created nor destroyed, only transferred from one ledger to another. A widely-accepted approach to ensuring consistency is to make the activities that manage the data *atomic*. Atomicity encompasses two properties: serializability and recoverability. *Serializability* [25] 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*.

Instead of treating transactions as monolithic entities, it is often useful to provide hierarchically structured nested transactions or *subtransactions* [24, 28]. A subtransaction's commit is dependent on that of its parent; aborting the parent will undo the child's effects. A transaction's effects become permanent only when it commits at the top level. A transaction can commit only when all its subtransactions have either committed or aborted.

Let TRANS denote a universal set of transactions. Transactions have an *a priori* tree structure, with a distinguished transaction U as the root. For a transaction A distinct from U, let *parent(A)* denote A's unique parent, *anc(A)* and *desc(A)* denote A's ancestors and descendants (which include A), *proper-anc(A)* and *proper-desc(A)* denote A's proper ancestors and descendants (which do not include A), and *lca(A,B)* denote the least common ancestor of A and B. Let *siblings* denote the set $\{(A,B) \in \text{TRANS}^2 \mid parent(A) = parent(B)\}$.

3.2. Serial and Concurrent Specifications

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 FIFO Queue might provide Enq and Deq operations.

An *event* is a paired operation invocation and response (e.g. Enq(x)/Ok() or Deq()/Empty()). A *serial history* for an object is a sequence of events that models a computation in the absence of failures and concurrency. An object's *serial specification* is its set of *legal* serial histories. For example, the serial specification for a FIFO queue includes all and only histories in which items are enqueued and dequeued in first-in-first-out order. A *system* is a set of objects. A serial history for a system is a sequence of *steps* of the form $[x\ e]$, where x is an object, and e is an event of x. A serial history for a system is legal if the subhistory associated with each individual object is legal. Serial histories are denoted by lower-case italic letters (g, h).

A concurrent history for a system is a sequence of *steps* of the form: $[x\ e\ A]$, [begin A], [commit A], or

[abort A], where x is an object, e an event, and A a transaction. Concurrent histories are denoted by upper-case italic letters (G, H). A concurrent history is *well-formed* if it satisfies the following properties:

- Each transaction begins exactly once, after its parent has begun, but before executing any other steps.
- Each transaction commits at most once, executing no steps after it commits.
- No transaction both commits and aborts.
- Invocations and responses are associated only with leaf transactions.
- No transaction commits until all its children have either committed or aborted.

For brevity, we will sometimes omit begin steps from examples. Henceforth, all concurrent histories are assumed to be well-formed.

Let H be a concurrent history for a system, and let *commit(H)* be the set of transactions that have committed in H. A transaction B has *committed to* A in H if $anc(B) \cap proper\text{-}desc(lca(A,B)) \subseteq commit(H)$. Let *perm(H)* be the subhistory of transactions committed to the top level transaction U. A partial order $\gg \subseteq siblings$ is *linearizing* if it totally orders all siblings in TRANS. A linearizing order thus induces a total order (also denoted by $\gg$) on the events of the leaf transactions. A concurrent history is *serializable* if there exists a linearizing order $\gg$ that reorders leaf transactions' object-event pairs to form a legal serial history. A concurrent history H is *atomic* if *perm(H)* is serializable.

A concurrent specification is *atomic* if each history in the specification is atomic. We assume all histories permitted by a specification are *prefix-closed*: any prefix of a legal history is itself a legal history. To model schedulers that have no advance knowledge of transactions, we assume that an active transaction can choose to commit at any time. A concurrent specification S is *on-line atomic* if it is atomic, and whenever H is in S and $H' = H \bullet [x \text{ commit } A]$ is well-formed, then H' is also in S. All specifications considered here are assumed to be on-line and prefix-closed.

A system encompassing multiple objects is not necessarily atomic just because each individual object in the system is atomic. A property $\mathcal{P}$ is a *local atomicity property* [32] if a system is atomic provided that each individual object satisfies $\mathcal{P}$. 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.

3.3. Quorum Consensus Replication

A more complete discussion of quorum consensus replication appears in [17]. A *replicated object* is an object whose state is stored redundantly at multiple sites. Replicated objects are implemented by two kinds of modules: *repositories* and *front-ends*. Repositories provide long-term storage for the object's state, while front-ends perform operations for clients. Front-ends correspond roughly to transaction managers and repositories correspond roughly to data managers [2].

Event orderings are determined by logical clocks [21], which provide a simple and efficient technique for extending the natural partial order of events in a distributed system to a total order.

R1	R2	R3
0:00 begin A	0:00 begin A	
	0:01 begin B	0:01 begin B
0:02 Enq(x)/Ok() A	0:02 Enq(x)/Ok() A	
	0:03 Enq(y)/Ok() B	0:03 Enq(y)/Ok() B
0:04 begin C		0:04 begin C
0:05 Enq(z)/Ok() C		0:05 Enq(z)/Ok() C
0:06 commit A	0:06 commit A	
	0:07 abort B	0:07 abort B

Figure 3-1: A Replicated Queue

A replicated object's state is represented as a *log*, which is a sequence of *entries*, each consisting of a timestamp, an event, and a transaction identifier. The log entries are partially replicated among the repositories. Figure 3-1 shows a schematic representation of a queue replicated among three repositories.

It should be emphasized that logs are intended to serve as a conceptual model for the replicated data, not as a literal design for an implementation. As discussed in [19], objects can be represented more compactly and efficiently when replication is implemented on top of an arbitrary atomicity mechanism. Whenever a repository accumulates a prefix of the object's complete history, that prefix can be replaced by a single timestamped version of the object. Each repository thus stores a single version together with a (potentially incomplete) sequence of log entries with later timestamps. Section 7 discusses how compaction techniques interact with particular atomicity properties.

A client executes an operation by sending the invocation to a front-end. The front-end merges the logs from an *initial quorum* for the invocation to construct a *view*. If the view indicates that no synchronization conflicts exist, the front-end chooses a response legal for the view, appends a timestamped entry to the view, and sends the updated view to a *final quorum* of repositories for that event.

Two conditions are necessary to execute an operation: the client must locate an available front-end for the object, and the front-end must locate a quorum of available 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. Consequently, each operation's availability is determined by its quorums, and constraints on quorum assignment determine the range of availability properties realizable by quorum consensus replication.

As discussed below, constraints on quorum assignment are expressed as requirements that certain initial and final quorums intersect. If each initial quorum for an invocation is required to intersect each final quorum for an event, then their levels of availability are inversely related, because if the quorums for one are made smaller (increasing availability) then the quorums for the other must be made correspondingly larger (decreasing availability). The weaker the constraints on quorum intersection, the wider the range of realizable availability properties.

We close this section with a precise definition of the constraints governing quorum assignment. Let $\succ$ be a relation between invocations and events. Let *e.inv* denote the invocation part of the event *e*. Informally, a subhistory is *closed* under $\succ$ if whenever it contains a step [*e A*] it also contains every earlier step [*e' A'*] such that *e.inv* $\succ$ *e'*, where neither *A* nor *A'* have aborted. More precisely, let *H(i)* denote the *i*-th step of *H*.

> **Definition 1:** *G* is a *closed subhistory* of *H* under $\succ$ if there exists an injective order-preserving map *s* such that $G(i) = H(s(i))$ for all *i* in the domain of *G*, and if *e.inv* $\succ$ *e'*, $H(j) = [e\ A]$, $H(j') = [e'\ A']$, $j > j'$, $s(i) = j$, and neither *A* nor *A'* has aborted, then there exists *i'* such that $s(i') = j'$.

Informally, $\succ$ is an *atomic dependency relation* if a response to an invocation is legal for a complete history whenever it is legal for a closed subhistory that includes the events on which the invocation depends. More precisely, let "•" denote concatenation:

> **Definition 2:** A relation $\succ$ between invocations and events is an *atomic dependency relation* if
>
> $G \bullet [e\ A]$ is legal implies that $H \bullet [e\ A]$ is legal
>
> for all events *e* and all legal concurrent histories *H*, whenever *G* is a closed legal subhistory containing all events *e'* such that *e.inv* $\succ$ *e'*.

A relation is a *serial dependency relation* if *H* and *G* are replaced by serial histories. Of principal interest are *minimal* dependency relations, having that property that no smaller relation is an atomic dependency relation. The basic correctness condition for quorum consensus replication is the following, which is proved in [17]:

> **Theorem 3:** A replicated object satisfies a concurrent specification *S* if and only if the quorum intersection relation is an atomic dependency relation for *S*.

Because the constraints on quorum intersection are expressed directly in terms of the concurrent specification, and not the serial specification, it is an immediate consequence of this theorem that

concurrency and availability are interdependent properties. This paper explores that interdependence in more detail.

4. Local Atomicity Properties

This section introduces four local atomicity properties [31]: serial atomicity, hybrid atomicity, static atomicity, and strong dynamic atomicity. For a fixed serial specification, we then consider the atomic dependency relations induced by the largest concurrent specifications satisfying each of these properties. These dependency relations represent the constraints on availability induced by maximizing the concurrency permitted by each property. In the next section, we compare these constraints.

Although Definition 2 fully characterizes a concurrent specification's atomic dependency relations, it may be difficult to apply this definition directly to particular data types. For hybrid, static, and strong dynamic atomicity, we give alternative characterizations in which the constraints on quorum intersection are expressed directly in terms of the object's serial specification. The alternative characterizations are easier to work with, and facilitate direct comparisons.

We will make extensive use of the following lemma.

Definition 4: Let G and H be histories, and e an event. G is a *false view* of H for e if G is a subhistory of H such that $G \bullet [e\ A]$ is legal but $H \bullet [e\ A]$ is not.

Lemma 5: If $\succ$ is not an atomic dependency relation, then there exist legal histories G and H and events e and e' such that G is a closed subhistory of H missing only e', it is false that $e.inv \succ e'$, and G is a false view of H for e.

Proof: Suppose G is missing k events of H. Consider the sequence of histories $\{H_i \mid i = 0,...,k\}$, where $H_0 = G$, $H_k = H$, and H_{i+1} is derived from H_i by restoring its earliest missing step.

If there exists an i such that H_i is legal but H_{i+1} is not, then H_i can be written as $G_0 \bullet G_1 \bullet [e_2\ A_2] \bullet G_2$, and H_{i+1} as $G_0 \bullet [e_1\ A_1] \bullet G_1 \bullet [e_2\ A_2] \bullet G_2$, where $G_0 \bullet [e_1\ A_1] \bullet G_1$ is legal, and $G_0 \bullet [e_1\ A_1] \bullet G_1 \bullet [e_2\ A_2]$ is not. But $G_0 \bullet G_1$ is a closed legal subhistory of $G_0 \bullet [e_1\ A_1] \bullet G_1$ containing all events related to $e_2.inv$ by $\succ$, proving the lemma.

Otherwise, suppose H_i is legal for all i between 0 and k. Because $H_0 \bullet [e\ A]$ is legal and $H_k \bullet [e\ A]$ is not, there must exist an i such that $H_i \bullet [e\ A]$ is legal but $H_{i+1} \bullet [e\ A]$ is not. This H_i is a closed legal subhistory of H_{i+1} containing all e' such that $e.inv \succ e'$, proving the lemma.

4.1. Serial Atomicity

Perhaps the simplest local atomicity property is to prohibit all interleaving, requiring transactions to execute serially, and to be serializable in the order they execute. While *serial atomicity* is clearly not a practical foundation for concurrency control in a distributed system, it provides a convenient baseline for evaluating more practical techniques. Let Serial(T) denote the set of strictly serial concurrent histories for the serial specification T, and let Serial*(T) denote the set of *serial dependency* relations for T. Serial*(T) need not have a unique minimal element [19].

4.2. Hybrid Atomicity

Hybrid atomicity encompasses techniques that combine locking with timestamps [6, 8, 1, 2].

> **Definition 6:** Let H be a concurrent history, and let A and B be committed sibling transactions. The linearizing order $\gg_H$ is defined as follows: $B \gg_H A$ if B's commit step follows A's. H is *hybrid atomic* if $Perm(H)$ is serializable in the order $\gg_H$.

Let Hybrid(T) denote the largest hybrid atomic concurrent specification for the serial specification T. Let Hybrid*(T) denote the set of atomic dependency relations for Hybrid(T). A *hybrid serialization* of a concurrent history H is a serial history constructed by committing some set of active transactions in H and serializing them in the order of their commit steps. A concurrent specification is on-line hybrid atomic if and only if each history's hybrid serializations are legal.

An *event tree* is a tree of events. Event trees model the status of a concurrent computation as follows: e' is an ancestor of e if the transaction that executed e' has committed to the transaction that executed e. An event tree t is *legal* if all serial histories generated by preorder traversals of t are legal. If t is a tree, and e a leaf event in t, let t/e denote t with e removed.

> **Definition 7:** The relation $\succ_H$ between invocations and events is defined as follows: $e.inv \succ_H e'$ if there exists a tree t having e and e' as leaf events, where t/e, t/e', and $t/e/e'$ are legal, but t is not.
>
> **Theorem 8:** $\succ_H$ is the unique minimal element of Hybrid*(T).
>
> **Proof:** The proof has two parts: we first show that every element of Hybrid*(T) contains $\succ_H$, and then that $\succ_H$ is itself a member of Hybrid*(T).
>
> For the first part, let t be an event tree, and e and e' leaf events of t satisfying the conditions of Definition 7. Pick a preorder numbering $e_1, e_2, \ldots, e_n$ for the events of t, where $e' = e_r$ and $e = e_s$. Let $\{B_i\}$ be a set of transactions with the property that if $e_i = parent(e_j)$, then $B_i = parent(B_j)$. Because operations can be executed only by leaf transactions, let $\{A_i\}$ be a set of transactions with the property that $B_i = parent(A_i)$. Let the concurrent history G be:

begin A_1
e_1 A_1
commit A_1
...
begin A_n
e_n A_n
commit A_n

including all the A_i in order except for A_r and A_s. G, $G \bullet [e_r\ A_r]$, and $G \bullet [e_s\ A_s]$ are each in Hybrid(T) because each of their hybrid serializations is a prefix of a (legal) preorder traversal of $t/e'/e$, t/e, or t/e' respectively. $G \bullet [e_r\ A_r] \bullet [e_s\ A_s]$ is not in Hybrid(T), however, because t is not legal, thus G is a false view of H for e.

For the second part, we assume G is a false view of H for e, where G is missing only e', and conclude that $e.inv \succ_H e'$. Construct the following event tree t from the active and committed events of $H \bullet [e\ A]$. Associate with each such event its least active ancestor transaction. (For committed events, the least active ancestor is U.) Events with the same least active ancestors are ordered in t by their serialization order, and others inherit the tree ordering of their least active ancestors.

Let t' be the tree consisting of the events of t, but with the following modified parent relation:

parent'(e'') = if parent(e'') = e' then parent(parent(e'))
else parent(e'')

Informally, t' makes e' a leaf without otherwise changing the structure of t by "splitting" the transaction that executed e' into two subtransactions. We claim that t'/e is legal, arguing by induction on the number of descendants of e' in t. The result is immediate if e' has no descendants in t, because then $t'/e = t/e$, which is legal. Otherwise, let e'' be a leaf descendant of e' in t. By induction, $t'/e/e''$ is legal. But it is false that $e''.inv \succ_H e'$, because e'' follows e' in H, G is closed under $\succ_H$, and G includes e'' but not e'. Since $t'/e/e''$, $t'/e/e'$, and $t'/e/e'/e''$ are legal, so is t'/e. Finally, since t'/e, t'/e', and $t'/e/e'$ are legal, but t' is not, it follows that $e.inv \succ_H e'$.

4.3. Static Atomicity

Static atomicity encompasses multiversion timestamping techniques [28, 27, 26].

Definition 9: Let H be a concurrent history, and let A and B be committed sibling transactions. Define the linearizing order $\gg_S$ as follows: $B \gg_S A$ if B's begin step is later than A's. H is *static atomic* if $Perm(H)$ is serializable in the order $\gg_S$.

Let Static(T) and Static*(T) be defined by analogy to Hybrid(T) and Hybrid*(T). An element of Static*(T) is called a *static dependency relation*, and a *static serialization* of a concurrent history H is a serial history constructed by committing some set of active transactions in H and serializing them in the order of their begin steps. H is on-line static atomic if and only if all its static serializations are legal.

Definition 10: The relation $\succ_S$ between invocations and events is defined as follows: if there exist serial histories h_1, h_2, and h_3 such that $h_1 \bullet h_2 \bullet h_3$ is legal, and either:

1. $h_1 \bullet e' \bullet h_2 \bullet h_3$ and $h_1 \bullet h_2 \bullet e \bullet h_3$ are legal, but $h_1 \bullet e' \bullet h_2 \bullet e \bullet h_3$ is illegal, or

2. $h_1 \bullet e \bullet h_2 \bullet h_3$ and $h_1 \bullet h_2 \bullet e' \bullet h_3$ are legal, but $h_1 \bullet e \bullet h_2 \bullet e' \bullet h_3$ is illegal,

then $e.inv \succ_S e'$.

Theorem 11: $\succ_S$ is the unique minimal element of Static*(T).

Proof: The proof has two parts: we first show that every element of Static*(T) contains $\succ_S$, and then that $\succ_S$ is itself a member of Static*(T).

The first part proceeds by contradiction. We will show that any relation $\succ$ that fails to satisfy the first condition of the theorem cannot be in Static*(T). (The argument when $\succ$ fails to satisfy the second condition is almost identical, and is omitted for brevity.) If h is a serial history, let $[h\ A]$ denote the concurrent history in which transaction A executes each event in h in turn. Let h_1, h_2, h_3, e, and e' be histories and events satisfying Definition 10, and let H be the following static atomic concurrent history:

begin A
begin B
begin C
begin D
begin E
h_1 A
commit A
h_2 C
commit C
h_3 E
commit E
e' B

Let G include all but the last step. G is a subhistory of H closed under $\succ$, but G is a false view of H for e, thus $\succ$ is not an element of Static*(T).

We now show that $\succ_S$ is itself an element of Static*(T). Let G be a false view of H for e, where G is missing a single event e'. Assume the illegal static serialization of $H \bullet [e\ A]$ has the form $h_1 \bullet e' \bullet h_2 \bullet e \bullet h_3$. (A similar contradiction can be derived by assuming the illegal serialization has the form $h_1 \bullet e h_2 \bullet e' \bullet h_3$.) The serial histories $h_1 \bullet h_2 \bullet h_3$, $h_1 \bullet e' \bullet h_2 \bullet h_3$, and $h_1 \bullet h_2 \bullet e \bullet h_3$ are legal as static serializations of G, H, and $G \bullet [e\ A]$, implying that $e.inv \succ_S e'$.

4.4. Strong Dynamic Atomicity

Strong dynamic atomicity is a generalization of locking schemes based on commutativity [12, 20, 29]. Two serial histories h and h' are *equivalent* (denoted $h \simeq h'$) if they cannot be distinguished by any future computations: $h \bullet s$ is legal if and only if $h' \bullet s$ is legal for all sequences of events s.

Definition 12: Let H be a concurrent history, and let A and B be committed sibling transactions. Define the partial order $\gg_D$ as follows: $B \gg_D A$ if B executes an event after A commits. A concurrent history is *strong dynamic atomic* if it is serializable in every linearizing order consistent with $\gg_D$, and if all such serializations are equivalent. ($\gg_D$ is not itself a linearizing order.)

Let Dynamic(T) and Dynamic*(T) be defined in the usual way. A *dynamic serialization* of a concurrent history H is constructed by committing some set of active transactions in H and serializing them in an order consistent with $\gg_D$.

Definition 13: Two events e and e' *commute* if for all serial histories h, whenever $h \bullet e$ and $h \bullet e'$ are both legal, then $h \bullet e \bullet e'$ and $h \bullet e' \bullet e$ are equivalent legal histories.

Definition 14: The relation $\succ_D$ between invocations and events is defined as follows: if e and e' do not commute, then $e.inv \succ_D e'$.

Lemma 15: If H and $H \bullet [e\ A]$ are concurrent histories in Dynamic(T), and h' and h are dynamic serializations of $H \bullet [e\ A]$ and H respectively, then $h' \simeq h \bullet e$.

Proof: Because A is active, $H \bullet [e\ A]$ has a dynamic serialization $h'' \bullet e$ in which A is serialized last, where h'' is a dynamic serialization of H. Because H is in Dynamic(T), all dynamic serializations of H are equivalent, thus $h'' \simeq h$, and $h' \simeq h \bullet e$.

Theorem 16: $\succ_D$ is the unique minimal element of Dynamic*(T).

Proof: The proof has two parts: we first show that every element of Dynamic*(T) contains $\succ_D$, and then that $\succ_D$ is itself a member of Dynamic*(T).

If e and e' do not commute, there exists a history h such that $h \bullet e$ and $h \bullet e'$ are both legal, but either $h \bullet e \bullet e'$ and $h \bullet e' \bullet e$ are not equivalent or neither is legal. Let H be the following concurrent history:

h A
Commit A
e' B

and let G include all but the last event. G is a false view of H for e.

To show that $\succ_D$ is itself in Dynamic*(T), we assume G is a false view of H for e, where G is missing only the event e', and we derive a contradiction from the assumption that there exist serialization orders $\gg$ and $\gg'$ compatible with $\gg_D$ that do not produce equivalent legal serializations of $H \bullet [e\ A]$. Let h and h' be their respective serializations of H, g and g' their serializations of G.

We claim that $h \simeq g \bullet e'$. The proof is by induction on the number of steps that follow $[e'\ B]$ in H. If $[e'\ B]$ is the last step in H, the result follows from Lemma 15. Suppose e' is followed by a single step. The result is immediate if that step is a *Commit* or *Abort*, because no new dynamic serializations are introduced. Otherwise, $H = G_0 \bullet [e'\ B] \bullet [e''\ C]$ and $G = G_0 \bullet [e''\ C]$. Let g_0 be the dynamic serialization of G_0 induced by $\gg$. The serial history $g_0 \bullet e''$ is legal, since it is equivalent to g (Lemma 15), and $g_0 \bullet e'$ is legal as a dynamic serialization of $G_0 \bullet [e'\ B]$. Because G is closed under $\succ_D$, e' and e'' commute, thus $h \simeq g_0 \bullet e' \bullet e'' \simeq g_0 \bullet e'' \bullet e' \simeq g \bullet e'$. As a serialization of $G \bullet [e\ A]$, $g \bullet e$ is legal. Because e and e' commute, $g \bullet e' \bullet e \simeq h \bullet e$ is legal. An analogous argument shows that $g' \bullet e' \bullet e \simeq h' \bullet e$ is legal. But because H is in Dynamic(T), $h \simeq h'$, hence $h' \bullet e \simeq h \bullet e$, a contradiction.

5. Comparisons

In this section we compare the constraints on availability and concurrency imposed by the local atomicity properties defined above. These comparisons show that availability and concurrency have a complex relation: they are neither completely independent nor completely dependent. Instead, they are complementary properties: each permits comparisons the other does not.

5.1. Serial Atomicity vs. The Others

Of the local atomicity properties considered here, strict serialization imposes the strongest constraints on concurrency and the weakest constraints on quorum assignment. Clearly, Serial(T) lies in the intersection of Hybrid(T), Dynamic(T), and Static(T). We now show that any atomic dependency relation is a serial dependency relation, but not necessarily vice-versa.

Theorem 17: Hybrid*(T), Dynamic*(T), and Static*(T) are each subsets of Serial*(T), and there exist T for which the inclusion is strict.

Proof: Let $\succ$ be any relation between invocations and events that is not in Serial*(T). Let g and h be serial histories such that $g = g_0 \cdot g_1$ is a subhistory of $h = g_0 \cdot e' \cdot g_1$ closed under $\succ$, such that g is a false view of h for e. If $H = [g_0\ A] \cdot [e'\ A] \cdot [g_1\ A]$ and $G = [g_0\ A] \cdot [g_1\ A]$, then G is a false view of H for e in all three of Hybrid(T), Static(T), and Dynamic(T).

To see that the inclusion is strict, consider an *Account* data type that provides two operations: *Credit* and *Debit*. Credit increments the account balance:

```
Credit = Operation(sum: Dollar).
```

Debit attempts to decrement the balance:

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

If the account balance is less than the amount to be debited, the invocation returns with an exception, leaving the account balance unchanged.

Under strict serialization, each initial quorum for Debit must intersect each final quorum for Credit and for Debit, as shown in Table 5-1. For an Account replicated among n identical sites, strict serialization permits $\lceil n/2 \rceil$ distinct quorum assignments: Debit requires any m sites, where $m > n/2$, and Credit requires any $n-m+1$ sites.

Hybrid, static, and strong dynamic atomicity each place additional constraints on quorum assignment, as shown in Table 5-2. Because initial and final Credit quorums must intersect, the number of distinct quorum assignments is reduced from $\lceil n/2 \rceil$ to exactly one: both Credit and Debit require a majority.

	Credit/Ok	Debit/Ok	Debit/Overdraft
Credit			
Debit	X	X	

Table 5-1: Quorum Intersections for Serial(Account)

	Credit/Ok	Debit/Ok	Debit/Overdraft
Credit	X		X
Debit	X	X	

Table 5-2: Quorum Intersections for Hybrid(Account),Static(Account), and Dynamic(Account)

5.2. Hybrid vs. Static Atomicity

Hybrid(T) and Static(T) are incomparable: each admits interleavings the other does not. One might therefore assume that Hybrid*(T) and Static*(T) would either be incomparable or identical, but instead Static*(T) is a subset of Hybrid*(T), and there exist T for which the inclusion is strict. In short, every quorum assignment that supports full hybrid atomicity also supports full static atomicity, but not vice-versa.

Theorem 18: For all T, Static*(T) $\subseteq$ Hybrid*(T).

Proof: Let $e.inv \succ_H e'$. By Definition 7, there exists an event tree t having e and e' as leaf events, such that t is illegal, but t/e, t/e', and $t/e/e'$ are each legal. The illegal preorder traversal of t can be written as either $h_1 \bullet e' \bullet h_2 \bullet e \bullet h_3$ or $h_1 \bullet e \bullet h_2 \bullet e' \bullet h_3$. The histories h_1, h_2, and h_3, and the events e and e' satisfy Definition 10, thus $e.inv \succ_S e'$.

Theorem 19: There exists a T such that Hybrid*(T) $\not\subseteq$ Static*(T).

Proof: By example. 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. There are three operations:

```
Write = Operation(item) Signals (Disabled)
```

stores a new item in the Prom if it has not been sealed, otherwise an exception is signaled.

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

returns the item in the Prom if it has been sealed, otherwise an exception is signaled.

```
Seal = Operation()
```

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

By Theorem 8, the unique minimal atomic dependency relation for Hybrid(Prom) is given in Table 5-3. To illustrate why Read invocations do not depend directly on successful Write events, we derive a contradiction from the assumption that t is an event tree and r and w are successful Read and Write events such that t, r, and w satisfy Definition 7. If t/r is legal, then there are no Seal events in t, because the leaf event w cannot be serialized after a Seal. If t/w is legal, however, then there must exist a Seal event in t that is an ancestor of r, because r can only be serialized after a Seal.

By Theorem 11, $\succ_S$ requires quorums for successful Read and Write operations to intersect, as shown in Table 5-4. These additional constraints translate directly into constraints on availability. Consider a Prom replicated among n identical sites to maximize the availability of the Read operation. Hybrid atomicity permits Read, Seal and Write quorums respectively consisting of any one, n, and one sites, while static atomicity would require Read, Seal and Write quorums to consist of any one, n, and n sites. In this example, static atomicity significantly reduces the availability of the Write operation.

	Seal/Ok	Write/Ok	Read/Disabled	Read/Ok
Read	X			
Write	X			
Seal		X	X	

Table 5-3: Quorum Intersections for Hybrid(Prom)

	Seal/Ok	Write/Ok	Read/Disabled	Read/Ok
Read	X	X		
Write	X			X
Seal		X	X	

Table 5-4: Quorum Intersections for Static(Prom)

5.3. Hybrid vs. Strong Dynamic Atomicity

Since $\gg_H$ is compatible with $\gg_D$, it follows that Dynamic(T) $\subseteq$ Hybrid(T), thus hybrid atomicity supports more concurrency than strong dynamic atomicity. For example, consider a FIFO Queue providing Enq and Deq operations. Hybrid(Queue) permits concurrent Enq events, because either commit order yields a serializable history, but Dynamic(Queue) does not, because the commit orders yield inequivalent histories. One might therefore expect that any quorum assignment that supports hybrid atomicity would also support static atomicity, but instead Hybrid*(T) and Dynamic*(T) are incomparable: each admits quorum assignments the other does not. On the one hand, hybrid atomicity obviates the need for certain quorum intersections because interleavings that had to be avoided under strong dynamic atomicity are no long illegal. On the other hand, hybrid atomicity introduces certain additional constraints on quorum intersection by making it necessary to distinguish between interleavings that never could have arisen under static atomicity.

Theorem 20: There exist T such that Dynamic*(T) and Hybrid*(T) are incomparable.

Proof: By example. An object of type *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 DoubleBuffer type provides three operations:

```
Produce = Operation(item)
```

copies an item into the producer buffer,

```
Transfer = Operation()
```

copies the item currently in the producer buffer to the consumer buffer, and

```
Consume = Operation() Returns (item)
```

returns a copy of the item currently in the consumer buffer.

The minimal element of Dynamic*(DoubleBuffer) is shown in Table 5-5, while the minimal element of Hybrid*(DoubleBuffer) is shown in Table 5-6. These dependency relations are incomparable: each permits quorum assignments the other does not.

	Produce/Ok	Transfer/Ok	Consume/Ok
Produce	X	X	
Transfer	X		X
Consume		X	

Table 5-5: Quorum Intersections for Dynamic(DoubleBuffer)

	Produce/Ok	Transfer/Ok	Consume/Ok
Produce		X	X
Transfer	X		X
Consume	X	X	

Table 5-6: Quorum Intersections for Hybrid(DoubleBuffer)

5.4. Static vs. Strong Dynamic Atomicity

Static(T) and Dynamic(T) are incomparable. We close this section by showing that Static*(T) and Dynamic*(T) are also incomparable. It is an immediate consequence of Theorems 18 and 20 that there exist T such that Dynamic*(T) $\not\subset$ Static*(T).

Theorem 21: There exist T such that Static*(T) $\not\subset$ Dynamic*(T).

Proof: By example. Consider a FIFO Queue that provides Enq and Deq operations. By Theorem 11, $\succ_S$ is the relation shown in Table 5-7, while by Theorem 16, however, $\succ_D$ is the relation shown in Table 5-8.

	Enq/Ok	Deq/Ok	Deq/Empty
Enq		X	X
Deq	X	X	

Table 5-7: Quorum Intersection for Static(Queue)

	Enq/Ok	Deq/Ok	Deq/Empty
Enq	X	X	X
Deq	X	X	

Table 5-8: Quorum Intersection for Dynamic(Queue)

6. 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 nested transactions incur a cost in availability for hybrid atomicity, although not for static or strong dynamic atomicity.

Let $\text{Hybrid}_n(T)$ be the collection of on-line hybrid atomic histories for T in which the depth of transaction nesting does not exceed *n*. Because $\text{Hybrid}_n(T)$ is a proper subset of $\text{Hybrid}_{n+1}(T)$, which is in turn a proper subset of Hybrid(T), the bounded-depth hybrid atomic specifications for T form a strict infinite hierarchy with respect to constraints on concurrency:

$$\text{Hybrid}_1(T) \subset \text{Hybrid}_2(T) \subset \ldots \subset \text{Hybrid}(T) \quad (1)$$

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

Any quorum assignment correct for $\text{Hybrid}_{n+1}(T)$ must also be correct for $\text{Hybrid}_n(T)$, thus the bounded-depth hybrid atomic specifications for T also form an infinite hierarchy with respect to constraints on availability.

$$\text{Hybrid}^*(T) \subseteq \ldots \subseteq \text{Hybrid}_2^*(T) \subseteq \text{Hybrid}_1^*(T) \quad (2)$$

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 show that this hierarchy is strict. Note that Theorem 8 does not hold for $\text{Hybrid}_n(T)$, because the proof required the ability to "split" part of a transaction into two subtransactions, potentially increasing the maximum depth of transaction nesting.

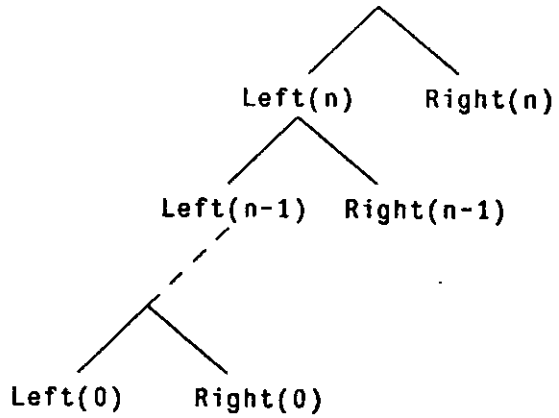

Figure 6-1: The Tree t_n

Theorem 22: For all *n*, there exists a T such that $\text{Hybrid}_n(T) \not\subset \text{Hybrid}_{n+1}(T)$.

Proof: Let t_n be the event tree shown in Figure 6-1. (For brevity, responses to invocations are omitted.) A *partial preorder traversal* of t_n is defined as follows:

1. Visit the root.

2. Visit 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 events of t_n in the order visited. Let Tree[n] be the serial specification consisting of the histories generated by partial preorder traversals of t_n, and let Tree'[n] be the specification constructed by removing from Tree[n] the "forbidden" history:

Left(n) • Left(n-1) • ... • Left(0) • Right(0) • ... • Right(n-1) • Right(n),

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

Let $\succ_n$ denote the unique minimal element of Hybrid*(Tree[n]). We claim that it is false that Right(0) $\succ_n$ Left(0). The argument proceeds by induction on n. The result is immediate when $n = 1$. Otherwise by Theorem 8, there exists an illegal event tree t with Right(0) and Left(0) as leaves, such that t/Right(0), t/Left(0), and t/Right(0)/Left(0) are legal. Let t' be the subtree of t rooted at but excluding Left(n). The tree t' has Right(0) and Left(0) as leaves, t'/Right(0), t'/Left(0), and t'/Right(0)/Left(0) are legal for Tree[n-1], but t' is not, thus Right(0) $\succ_{n-1}$ Left(0), contradicting the induction hypothesis.

Let $\succ_n'$ denote the unique minimal element of Hybrid*(Tree'[n]). Since t_n/Right(0), t_n/Left(0), and t_n/Left(0)/Right(0) are each legal, but t_n is illegal for Tree'[n], Right(0) $\succ_n'$ Left(0) by Theorem 8.

Let G be a false view of H for Right(0), where G is missing only the event Left(0). We show that A, the transaction executing Right(0), must be nested to a depth greater than n. A concurrent history can fail to be in Tree'[n] in two ways: by having a serialization that is not a partial preorder traversal of t_n, or by having the forbidden history as a serialization. H•[Right(0) A] must be in Tree[n], since G is closed under $\succ_n$, therefore H•[Right(0) A] must have the forbidden history as a hybrid serialization.

For $1 \geq i \geq n$, let L_i and R_i be the least active ancestors of the transactions that executed Right(i) and Left(i) in H. L_i and R_i cannot be the same transaction, because the forbidden history serializes Left(0) and Right(0) between them. R_i cannot be an ancestor of L_i, because Right(i) must then be serialized before Left(i), ruling out the forbidden history as a possible serialization. L_i cannot be an ancestor of R_i, because if R_i commits before A, Left(i) would be serialized before Right(i), which would be serialized before Right(0), an ordering not permitted by any preorder traversal.

L_{i+1} must be an ancestor of both L_i and R_i, since Left($i + 1$) precedes Left(i) and Right(i) in every traversal of t_n. Since L_i is concurrent with R_i and L_{i+1} is committed to R_i, L_{i+1} and L_i cannot be the same transaction, and therefore L_{i+1} is a proper ancestor of L_i. The depth of L_n is at least one, hence the depth of L_1 is at least n.

For both static and strong dynamic atomicity, we have the same dual hierarchies.

$\text{Static}_1(T) \subset \text{Static}_2(T) \subset \ldots \subset \text{Static}(T)$

$$\text{Static}^*(T) \subseteq \ldots \subseteq \text{Static}_2^*(T) \subseteq \text{Static}_1^*(T)$$

$$\text{Dynamic}_1(T) \subset \text{Dynamic}_2(T) \subset \ldots \subset \text{Dynamic}(T)$$

$$\text{Dynamic}^*(T) \subseteq \ldots \subseteq \text{Dynamic}_2^*(T) \subseteq \text{Dynamic}_1^*(T)$$

Here, however, the inclusions of $\text{Static}_{n+1}^*(T)$ in $\text{Static}_n^*(T)$ and $\text{Dynamic}_{n+1}^*(T)$ in $\text{Dynamic}_n(T)$ are not strict. The proofs of Theorem 11 and 16 actually show slightly stronger results: if $\succ_S$ is not a subset of $\succ$, then $\succ$ is not an element of $\text{Static}_1^*(T)$, and similarly, if $\succ_D$ is not a subset of $\succ$, then $\succ$ is not an element of $\text{Dynamic}_1^*(T)$. Under static and strong dynamic atomicity, quorum assignment is insensitive to transaction nesting.

7. Bounded Versions

The log compaction technique proposed in [19] for serial computations is readily applied to hybrid and strong dynamic atomicity. Whenever a repository has acquired a prefix of an object's committed state (or a log equivalent to such a prefix), it may replace that prefix with a single timestamped version. This compaction is possible because both hybrid and strong dynamic atomicity have the property that once a transaction *A* has committed at a repository, any later transaction that visits that repository is serializable after *A*. Static atomicity, however, is not so well-behaved. Since a transaction can be serialized arbitrarily far "in the past," support for full static atomicity requires retaining a complete log of events for active and committed transactions. For example, a file must retain a record of every Read and Write event. The record of Write events is necessary to accommodate an out-of-order Read, and the record of Read events is necessary to ensure that an out-of-order Write does not invalidate a value read by another transaction. A similar problem arises for a bank account, where it is necessary to keep a complete record of credits, debits, and attempted overdrafts to permit transactions to insert new events at arbitrary points in the committed history.

To circumvent this difficulty for files, Reed [28] suggested discarding all but the *n* most recently written file versions, for some fixed number *n*. The choice of *n* reflects a direct trade-off between concurrency (the set of permitted interleavings) and space (the volume of data representing the file). Papadimitriou and Kannelakis [26] have examined the trade-offs between efficiency and concurrency imposed by such a restriction.

How can the n-version limitation be extended to replicated objects other than files, objects represented by logs rather than versions? The key observation here is that the bound on the number of versions reflects a bound on how far the physical ordering of events may diverge from their serialization ordering. An n-version file does not permit a Read event to be serialized before the n-th physically preceding Write, and similarly for Write events. Such constraints establish a "horizon" before which transactions may no longer be serialized. Any prefix of a log that represents the object's

committed state prior to the horizon can be replaced by a single timestamped version. As the horizon advances, later entries are merged with the version and discarded.

The principal difficulty in generalizing the n-version limitation to arbitrary objects is recognizing which event orderings are important. For example, restricting the relative ordering of Read events places additional constraints on quorum assignments without facilitating log compaction. By contrast, restricting the relative ordering of Read and Write events permits older versions to be discarded without further restricting quorum assignment. In this section, we introduce a generalization of the n-version limitation satisfying the following properties:

- For files, one can discard all but the most recent n committed versions.
- For objects of arbitrary type, an appropriate prefix of the committed state can be compacted into a single timestamped version.
- The ability to compact the committed state introduces no extraneous constraints on quorum assignment.

Definition 23: A symmetric relation ~ between events is an *conflict relation* if

$g \cdot e$ is legal implies that $h \cdot e$ is legal

for all events e and all legal serial histories h, whenever g is a closed legal subhistory containing all events e' such that $e \sim e'$.

Such a relation is *minimal* if no smaller symmetric relation is a conflict relation. The proof of Lemma 5 is easily extended to yield:

Lemma 24: If ~ is a minimal conflict relation and $e \sim e'$, then e and e' can be relabeled so that there exist legal serial histories $h = g_1 \cdot e' \cdot g_2$ and $g = g_1 \cdot g_2$, such that g is a closed subhistory of h, and $g \cdot e$ is legal but $h \cdot e$ is not.

Informally, one step is out-of-order with respect to another if their events are related by ~ and their physical order differs from their serialization order.

Definition 25: Pick a minimal conflict relation ~. The step $[a\ A]$ is *out-of-order* with respect to $[b\ B]$ in the concurrent history H if $a \sim b$, $[a\ A]$ follows $[b\ B]$ in H, but $B \gg_S A$.

The restriction of the out-of-order relation to events related by ~ will ensure that the n-version restriction imposes no extraneous constraints on quorum intersection.

Definition 26: A concurrent history H is *n-version static atomic* if it is static atomic, and no step is out of order with respect to n or more steps.

Let Static^n(T) denote the largest n-version static atomic concurrent specification for the serial specification T, and let Static^n*(T) denote its set of atomic dependency relations. (Each of these domains depends on the original choice of conflict relation.)

For Static^n(File), Read and Write events conflict, thus a transaction cannot read before the n-th most recent Write event, and earlier committed versions can be discarded. For Static^n(Account), Debit events conflict and Credit and Debit events conflict, thus a Credit event cannot be serialized before n Debit events, and a Debit event cannot be serialized before n Credit or Debit events. An account can

be represented as single balance followed by a sequence of Credit and Debit entries. (Adjacent Credit events can be further consolidated.)

$Static^n(T)$ is a proper subset of $Static^{n+1}(T)$, which is in turn a proper subset of Static(T), thus we have the infinite strict hierarchy:

$$Static^1(T) \subset Static^2(T) \subset ... \subset Static(T) \quad (3)$$

We now show that there exists a dual infinite hierarchy:

$$Static^*(T) \subseteq ... \subseteq Static^{2*}(T) \subseteq Static^{1*}(T) \quad (4)$$

Moreover, each inclusion is strict for certain types.

Lemma 27: If $\succ^n$ is an element of $Static^{n*}(T)$ and $e \sim e'$, then $e.inv \succ^n e'$ and $e'.inv \succ^n e$.

Proof: Since $\sim$ is minimal, there exists e, e', g_1, and g_2 satisfying the properties of Lemma 24. Consider the history H:

begin A
begin B
begin C
begin D
g1 A
commit A
g2 C
commit C
e' B

Let G be the history including all but the last step. H, G, and $G \bullet [e\ D]$ are in $Static^1(T)$, and hence in $Static^n(T)$, but G is a false view of H for e, hence $e.inv \succ^n e'$. Similarly, replacing e' by e in H yields $e'.inv \succ^n e$.

Theorem 28: The hierarchy 4 is valid, and for each inclusion, there exists a T for which the inclusion is strict.

Proof: First, we show that if $\succ^{n+1}$ is an element of $Static^{n+1*}(T)$, then it is an element of $Static^{n*}(T)$. Let H be in $Static^n(T)$, and let G be a subhistory of H closed under $\succ^{n+1}$ containing each event e' such that $e.inv \succ^{n+1} e'$. We claim that if $G \bullet [e\ A]$ is in $Static^n(T)$, then so is $H \bullet [e\ A]$. Because $G \bullet [e\ A]$ is in $Static^{n+1}(T)$, so is $H \bullet [e\ A]$, and since G contains all events related to e by $\sim$, $H \bullet [e\ A]$ must also be in $Static^n(T)$.

We next show that the inclusions are strict. Consider the following generalization of the Prom data type introduced above. A *Prom[i]* is a container for an item, providing *Write*, *Read*, and *Seal* operations. Prom[i] differs from Prom as follows: Seal disables Writes, but Reads are enabled only after i Seal operations have been executed. Prom is equivalent to Prom[1].

Let $\sim$ be the following minimal conflict relation:

	Seal()/Ok()	Read()/No()	Write(x)/Ok()
Seal()/Ok()		X	X
Read()/No()	X		
Write(x)/Ok()	X		

Let $\succ$ denote the relation between invocations and events induced by the following

	Seal()/Ok()	Read()/No()	Write(x)/Ok()
Seal	X	X	X
Read	X		
Write	X		

relation: We claim that $\succ$ is an element of Static^n*(Prom[n]), but not of Static^n*(Prom[n-1]).

The only part of $\succ$ that requires scrutiny is the relation between successful Reads and Writes. Suppose G is a false view of H under $\succ$ missing a Write. G•[Read()/OK(x) A] is legal only if G includes n Seal events committed to A. If H is in Static^n(Prom[n]), then the missing Write cannot be out of order with respect to all n Seals. It must precede at least one Seal, and since G is closed under $\succ$, the Write must appear in G, a contradiction. If H is in Static^n(Prom[n-1]), however, then the missing Write can follow the n-1 Seals, thus $\succ$ is not an element of Static^n*(Prom[n-1]).

8. 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 [27], and Argus [22] and TABS [30] are based on strong dynamic 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 quorum assignment. A comparison of the constraints on quorum assignment necessary to maximize concurrency under static, hybrid, and strong dynamic atomicity shows that availability and concurrency are complementary properties, each permitting comparisons the other does not. Although static and hybrid atomicity place incomparable constraints on concurrency, hybrid atomicity places fewer constraints on quorum assignment. Although hybrid atomicity places fewer constraints on concurrency than strong dynamic atomicity, they place incomparable constraints on quorum assignment. Bounding the maximum depth of transaction nesting tightens constraints on concurrency for all three properties, but reduces the constraints on concurrency for hybrid atomicity alone. Bounding the number of "versions" retained under static atomicity reduces concurrency but enhances availability. Hybrid atomicity is thus the only property undominated with respect to both availability and concurrency. Although such optimally concurrent schedulers are unlikely to be cost-effective in practice, an understanding of the trade-offs they impose is a prerequisite for achieving a systematic understanding of the fundamental constraints governing availability and concurrency for replicated data.

References

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

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

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

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

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

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

[7] Cooper, E. C.
Circus: a replicated procedure call facility.
In *Proceedings 4th Symposium on Reliability in Distributed Software and Database Systems,* pages 11-24. October, 1984.

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

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

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

[11] El-Abbadi A., and Toueg, S.
Availability in Partitioned Replicated Databases.
Technical Report TR 85-721, Dept. of Computer Science, Cornell University, December, 1985.

[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.
Communications ACM 19(11):624-633, November, 1976.

[13] Garcia-Molina, H. and Barbara, D.
How to assign votes in a distributed system.
To appear in JACM.

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

[15] Goodman, N., Skeen, D., Chan, A., Dayal, U., Fox, S, and Ries, D.
A recovery algorithm for a distributed database system.
In *Proceedings, 2nd ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*. March, 1983.

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

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

[18] Herlihy, M. P.
Using Type Information to Enhance the Availability of Partitioned Data.
Technical Report CMU-CS-85-???, Carnegie-Mellon University, April, 1985.

[19] Herlihy, M. P.
A quorum-consensus replication method for abstract data types.
ACM Transactions on Computer Systems 4(1), February, 1986.

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

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

[22] Liskov, B., and Scheifler, R.
Guardians and actions: linguistic support for robust, distributed programs.
ACM Transactions on Programming Languages and Systems 5(3):381-404, July, 1983.

[23] Lynch, N. A.
Concurrency control for resilient nested transactions.
In *Proc. 2nd ACM Symposium on Principles of Database Systems*. March, 1983.
Revised version to appear in Advances in Computing Research.

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

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

[26] Papadimitriou, C.H., and Kanellakis, P.
On concurrency control by multiple versions.
ACM transactions on database systems 9(1):89-99, March, 1984.

[27] Reed, D. P., and Svobodova, L.
SWALLOW: a distributed data storage system for a local network.
In *Proceedings of the International Workshop on Local Networks*. August, 1980.

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

[29] Schwarz, P. and Spector. A.
Synchronizing shared abstract types.
ACM Transactions on Computer Systems 2(3):223-250, August, 1984.

[30] Spector, A. Z., Butcher, J., Daniels, D. S., Duchamp, D. J., Eppinger, J. L., Fineman, C. E., Heddaya, A., Schwarz, P. M.
Support for distributed transactions in the TABS prototype.
Technical Report CMU-CS-84-132, Carnegie-Mellon University, July, 1984.

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

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