CMU-CS-85-123

Comparing How Atomicity Mechanisms Support Replication

Maurice Herlihy
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213
3 May 1985

Abstract

Most pessimistic mechanisms for implementing atomicity in distributed systems fall into three broad categories: two-phase locking schemes, timestamping schemes, and hybrid schemes employing both locking and timestamps. This paper proposes a new criterion for evaluating these mechanisms: the constraints they impose on the availability of replicated data.

A replicated data item is a typed object that provides a set of operations to its clients. A quorum for an operation is any set of sites whose co-operation suffices to execute that operation, and a quorum assignment associates a set of quorums with each operation. Constraints on quorum assignment determine the range of availability properties realizable by a replication method.

This paper compares the constraints on quorum assignment necessary to maximize concurrency under generalized locking, timestamping, and hybrid concurrency control mechanisms. This comparison shows that hybrid schemes impose weaker constraints on availability than timestamping schemes, and locking schemes impose constraints incomparable to those of the others. Because hybrid schemes permit more concurrency than locking schemes, these results suggest that hybrid schemes are preferable to the others for ensuring atomicity in highly available and highly concurrent distributed systems.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

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. [10, 17, 26]), timestamping schemes (e.g. [25, 24, 23]), and hybrid schemes employing both locking and timestamps (e.g. [7, 8, 2, 3]). 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 provide better support for highly available and highly concurrent distributed systems than either locking or timestamping mechanisms.

Our analysis uses a *quorum consensus* replication method proposed by the author [14, 15]. This method systematically exploits type-specific properties of the data to support better availability and concurrency than comparable methods in which operations are classified only as reads or writes. Associated with each operation of the data type is a set of *quorums*, which are collections of sites whose cooperation suffices to execute the operation. A *quorum assignment* associates a set of quorums with each operation. An analysis of the atomic data type's specification (which includes the level of concurrency supported) yields a set of constraints on quorum assignment necessary and sufficient to ensure the correctness of the replicated implementation. The constraints on quorum assignment determine the range of availability properties realizable by quorum consensus replication.

The three-part classification of atomicity mechanisms is formalized using a model developed by Weihl [28]. 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.

Elsewhere [14, 15], we have shown that quorum consensus replication cannot simultaneously minimize the constraints on both concurrency and availability. At one extreme in the availability/concurrency trade-off, all three properties support the same minimal set of constraints on quorum choice, 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. A direct comparison of replicated objects that lie between these extremes is difficult, since any valid set of constraints on quorum assignment typically yields

incomparable constraints on concurrency.

This paper presents the following results:

- 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.

- A quorum assignment that supports full strong dynamic atomicity does not necessarily support full hybrid atomicity, and vice-versa. Thus, maximizing concurrency under strong dynamic atomicity yields constraints on availability incomparable to those of hybrid atomicity

- A quorum assignment that supports full strong dynamic atomicity does not necessarily support full static atomicity, and vice-versa. Thus, maximizing concurrency under strong dynamic atomicity yields constraints on availability incomparable to those of static atomicity

- Static atomicity is ensured by a unique weakest set of constraints on quorum assignment, as is strong dynamic atomicity, but the weakest set of constraints sufficient to ensure hybrid atomicity is not necessarily unique.

These relations are illustrated schematically in Figure 1-2. Hybrid atomicity is the only property that is undominated for both availability and concurrency, suggesting that hybrid schemes may be preferable to the others for implementing atomicity in distributed systems supporting high levels of availability and concurrency.

2. Other Related Work

In the *available copies* replication method [12], failed sites are dynamically detected and configured out of the system, and recovered sites are detected and configured back in. Clients may read from any available copy, and must write to all available copies. Systems based on variants of this method include SDD-1 [13] and ISIS [5]. Unlike quorum consensus methods, the available copies method does not preserve serializability in the presence of communication link failures such as partitions.

In the *true-copy token* scheme [21], a replicated file is represented by a collection of copies. Copies that reflect the file's current state are called *true copies*, and are marked by *true-copy tokens*. The set of true copies can be reconfigured to permit activities to operate on local copies of files. This method preserves serializability in the presence of crashes and partitions, but the availability of a replicated file is limited by the availability of the sites containing its true copies.

A formal model for concurrency control in replicated databases proposed by Bernstein and Goodman can be used to show the correctness of several replication methods [4]. This model is based on two

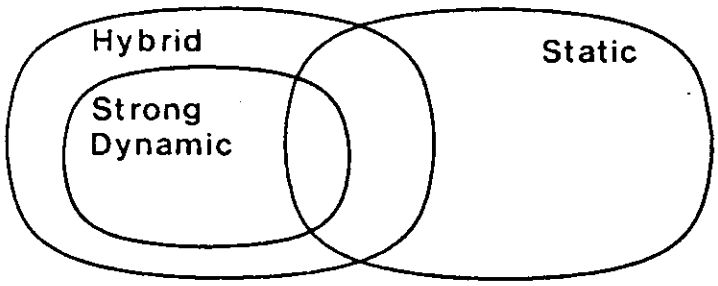

Figure 1-1: Concurrency

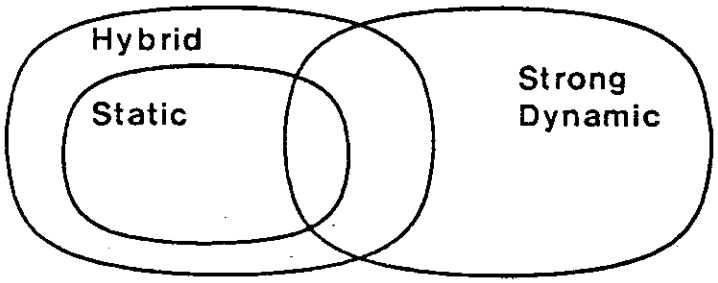

Figure 1-2: Availability

assumptions that do not apply to the replication methods used in this paper: that a replicated object is represented by multiple copies, and that all information about operations is captured by a simple read/write classification. These assumptions unnecessarily restrict availability and concurrency.

Several recent proposals for replication methods treat concurrency control and replication independently [11, 9, 1]. The replication method used here takes a different approach by integrating replication and concurrency control in a single mechanism. Although independent methods are simpler, integrated methods support better concurrency.

The earliest use of quorum consensus is a file replication method due to Gifford [11]. A quorum-consensus replication method for directories has been proposed by Bloch, Daniels, and Spector [6]. These methods can be viewed as specially optimized instances of the method used in this paper. Extensions to quorum consensus that further enhance availability in the presence of partitions have been proposed for files by Eager and Sevcik [9] and for arbitrary data types by the author [14, 16].

3. Assumptions and Terminology

This section summarizes our model of computation. A more detailed exposition appears in [14, 15].

We admit two kinds of faults: sites may crash and communication links may be interrupted. When a site crashes, it becomes temporarily or permanently inaccessible. Communication link failures result in lost messages; garbled and out-of-order messages can be detected (with high probability) and discarded. Transient communication failures may be hidden by lower level protocols, but longer-lived failures can cause *partitions*, in which functioning sites are unable to communicate. A failure is detected when a site that has sent a message fails to receive a response after a certain duration. The absence of a response may indicate that the original message was lost. that the reply was lost, that the recipient has crashed, or simply that the recipient is slow to respond.

The basic units of computation are sequential processes called *actions*, or transactions. Actions are *atomic*, that is, serializable and recoverable. Serializability means that actions appear to execute in a serial order [22], and recoverability means that an action either succeeds completely, or has no effect. An action that completes all its changes successfully *commits*; otherwise it *aborts*, and any changes it has made are undone. An action that has neither committed nor aborted is *active*.

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 of creating and manipulating objects of that type. For example, an object of type *Queue* has two operations: *Enq* places an item in the queue, and *Deq* removes the least recently enqueued item, raising an exception [19] if the queue is empty.

3.1. Atomicity

An *event* is pair consisting of an operation invocation and a response. In the absence of failures and concurrency, an object's state is given by a sequence of events called a *serial history*. For example, the following is a serial history for a *Queue*:

```
Enq(x);Ok()
Enq(y);Ok()
Deq();Ok(x)
Deq():Empty()
```

Serial histories are denoted by lower-case letters (e.g. *h*, *g*). A *serial specification* for an object is a set of possible serial histories for that object. For example, the serial specification for *Queue* includes all and only the histories in which items are dequeued in first-in-first-out order. A *legal* history is one that is included in the object's serial specification. Serial specifications are assumed to be prefix-closed: any prefix of a legal serial history is legal.

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

Begin A
Enq(x);Ok() A
Begin B
Enq(y);Ok() B
Commit A
Deq();Ok(x) B
Commit B

The ordering of operation executions in a behavioral history reflects the order in which the the object returned the responses, not necessarily the order in which it received the invocations. Behavioral histories are denoted by upper-case letters (e.g. *H*, *G*).

A *behavioral specification* for an object is a set of possible behavioral histories for that object. Just as for serial histories, a *legal* behavioral history is one that is included in the object's behavioral specification. All behavioral specifications are assumed to be prefix-closed and *on-line*: the result of appending a *Commit* entry for an active action to a legal behavioral history is legal.

The serial and behavioral specifications for the objects considered in this paper are related by the notion of *atomicity*. Let $\gg$ denote a total order on committed and active actions, and let *H* be a behavioral history. The *serialization* of *H* in the order $\gg$ is the serial history *h* constructed by reordering the events in *H* so that if $B \gg A$ then the subsequence of events associated with *A* precedes the subsequence of events associated with *B*. *H* is *serializable* in the order $\gg$ if *h* is legal. *H* is *serializable* if it is serializable in some order. *H* is *atomic* if the subhistory associated with committed actions is serializable. An object is atomic if every history in its behavioral specification is atomic.

A *system* encompassing multiple objects is atomic if all component objects are atomic and serializable in a common order. A property $\mathcal{P}$ is a *local atomicity property* [28] if a system is atomic provided that each individual object satisfies $\mathcal{P}$. If a system-wide local atomicity property is agreed upon 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.

3.2. Replication

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 carry out operations for clients. Front-ends correspond roughly to transaction managers and repositories correspond roughly to data managers [3].

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 an action identifier. Timestamps are generated by a system of Lamport clocks [18]. The log entries are partially replicated among the repositories. Figure 3-1 shows a schematic representation of a queue replicated among three repositories.

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. Informally, a subhistory is *closed* under $\succ$ if whenever it contains an event *[e A]* it also contains every earlier event *[e' A']* such that *e.inv* $\succ$ *e'*, and neither *A* nor *A'* have aborted. More precisely, let *H(i)* denote the *i*-th event of *H*, and let *e.inv* denote the invocation part of the event *e*.

> **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 for a behavioral specification if for all invocations *inv*, all responses *res*, all legal histories *H*, and all closed subhistories *G* containing the events *e* of *H* such that *inv* $\succ$ *e*, *H•[inv;res A]* is legal whenever *G•[inv;res A]* is legal.

A replicated object satisfies its behavioral specification if and only if its quorum intersection relation is an atomic dependency relation [15]. An atomic dependency relation is *minimal* if no smaller relation is an atomic dependency relation. A minimal dependency relation corresponds to the weakest set of constraints sufficient to satisfy that behavioral specification. In the remainder of this paper, we compare minimal atomic dependency relations for several classes of behavioral specifications.

4. Static vs. Hybrid Atomicity

A system of Lamport Clocks [18] can be used to impose an unambiguous ordering on *Begin* and *Commit* events.

> **Definition 3:** A behavioral history is *static atomic* if committed actions are serializable in the order of their *Begin* events, and it is *hybrid atomic* if they are serializable in the order of their *Commit* events [28].

A behavioral specification is static (hybrid) atomic if all its histories are static (hybrid) atomic. Static and hybrid atomicity are local atomicity properties.

Let *Static(T)* denote the largest prefix-closed on-line static atomic behavioral specification for the

serial specification T, and let *Hybrid(T)* be defined similarly for hybrid atomicity. For brevity, an atomic dependency relation for *Static(T)* is called a *static dependency relation* for T, and similarly for *Hybrid(T)*. In this section we show that every hybrid dependency relation for T is also a static dependency relation, but not vice-versa. Moreover, a data type's minimal static dependency relation is unique, but its minimal hybrid dependency relation need not be.

We use the following terminology. A *static serialization* of a behavioral history H is a serial history constructed by committing some set of active actions in H and serializing them in the order of their *Begin* events. H is on-line static atomic if and only if all its static serializations are legal. A *hybrid serialization* of H is constructed similarly, except that actions are serialized in the order of their *Commit* events.

Theorem 4: Every static dependency relation for a data type is also a hybrid dependency relation.

Proof: We show the contrapositive: any relation that is not a hybrid dependency relation cannot be a static dependency relation. If $\succ$ is not a hybrid dependency relation, then there exist behavioral histories H, G, and $G \bullet [e\ A]$ in *Hybrid(T)* such that G is a closed subhistory of H containing all events e' such that $e.inv \succ e'$, but $H \bullet [e\ A]$ is not in *Hybrid(T)*. We show that $\succ$ is not a static dependency relation by constructing behavioral histories H', G', and $G' \bullet [e\ A]$ in *Static(T)* such that G' is a closed subhistory of H' containing all events e' such that $e.inv \succ e'$, but $H' \bullet [e\ A]$ is not in *Static(T)*.

Because $H \bullet [e\ A]$ is not in *Hybrid(T)*, it has an illegal hybrid serialization in which committed and active actions are serialized in an order $\gg$. Let H' and G' be the histories constructed from H and G by moving their *Begin* events to the start of each history, and reordering them in the order $\gg$. H' is in *Static(T)* because any static serialization of a prefix of H' is a hybrid serialization of the corresponding prefix of H, and is therefore legal. G' and $G' \bullet [e\ A]$ are in *Static(T)* by analogous arguments. $H' \bullet [e\ A]$, however, is not in *Static(T)* because it has an illegal static serialization in which committed and active actions are serialized in the order $\gg$.

The following example will be used to show that the converse of Theorem 4 is false. 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.

We claim that the following is a hybrid dependency relation for *PROM*.

Seal() $\succ_H$ Write(x);Ok()
Seal() $\succ_H$ Read();Disabled()
Read() $\succ_H$ Seal();Ok()
Write(x) $\succ_H$ Seal();Ok()

For brevity, we restrict our attention to the *Read* invocation. Let *H* and *G* be in *Hybrid(PROM)*, where *G* is a closed subhistory containing all events *e* such that *Read* $\succ_H$ *e*. If *G•[Read();Disabled() A]* is in *Hybrid(PROM)*, then *G* contains no *Seal* events, either committed or active. If there are no *Seal* events in *G*, there are none in *H*, because *Read* $\succ_H$ *Seal();Ok()*. If *H* contains no *Seal* events, then *H•[Read();Disabled() A]* is in *Hybrid(PROM)*, because an exceptional *Read* is always legal before the *PROM* is sealed.

If *G•[Read();Ok(x) A]* is in *Hybrid(PROM)*, then every hybrid serialization must satisfy the following conditions:

1. The *Read* is serialized after a *Seal*.

2. *x* is the last value written before the *PROM* is sealed.

If *G* satisfies the first condition, so does *H*, because *Read()* $\succ_H$ *Seal();Ok()*. A *normal* event is one that terminates with *Ok*. All normal *Write* events must precede the first unaborted *Seal* event in *H* because normal *Write* events must be serialized before the first *Seal* event. *G* therefore includes all normal *Write* events of *H*, because *Read()* $\succ_H$ *Seal();Ok()*, *Seal()* $\succ_H$ *Write(x);Ok()*, and *G* is closed. Thus, if *G* satisfies the second condition, so does *H*.

Theorem 5: A hybrid dependency relation need not be a static dependency relation.

Proof: We show that the relation $\succ_H$ is not a static dependency relation for *PROM*. Let *H* be the following history:

Begin A
Begin B
Begin C
Begin D
Write(x);Ok() A
Commit A
Seal();Ok() C
Commit C
Read();Ok(x) D

and let *G* include all events of *H* except the last. *H*, *G*, and *G•[Write(y);Ok() B]* are in *Static(PROM)*, but *H•[Write(y);Ok() B]* is not, because the value read by *D* will be invalidated if *B* commits. Thus, $\succ_H$ is not a static dependency relation.

Theorem 6: *T* has a unique minimal static dependency relation $\succ_S$, defined as follows: *inv* $\succ_S$ *e* if there exists a response *res*, and serial histories h_1, h_2, and h_3 such that $h_1 \bullet h_2 \bullet h_3$ is legal, and either:

1. $h_1 \bullet$*[inv;res]*$\bullet h_2 \bullet h_3$ and $h_1 \bullet h_2 \bullet e \bullet h_3$ are legal, but $h_1 \bullet$*[inv;res]*$\bullet h_2 \bullet e \bullet h_3$ is illegal.

2. $h_1 \bullet e \bullet h_2 \bullet h_3$ and $h_1 \bullet h_2 \bullet [inv;res] \bullet h_3$ are legal, but $h_1 \bullet e \bullet h_2 \bullet [inv;res] \bullet h_3$ is illegal.

Proof: We first show that every static dependency relation $\succ$ contains $\succ_S$. Otherwise, suppose $\succ$ fails to satisfy the first condition. (An analogous argument shows that $\succ$ is not a static dependency relation if it fails to satisfy the second condition.) If h is a serial history, we use the notation $[h\ A]$ to denote the behavioral history in which action A executes each event in h in turn. Let H be the following static atomic behavioral 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

and let G include all but the last event. G is a closed subhistory of H containing all events e' such that $inv \succ e'$. H, G, and $G \bullet [inv;res\ D]$ are in *Static(T)*, but $H \bullet [inv;res\ D]$ is not, because it has the illegal static serialization $h_1 \bullet [inv;res] \bullet h_2 \bullet e \bullet h_3$. It follows that $\succ$ is not a static dependency relation.

We now show that $\succ_S$ is itself a static dependency relation. Otherwise, there exist behavioral histories H, G, and $G \bullet [e\ A]$ in *Static(T)* such that G is a closed subhistory of H under $\succ_S$ containing all events e' such that $e.inv \succ_S e'$, but $H \bullet [e\ A]$ is not in *Static(T)*. We derive a contradiction from these hypotheses.

The proof is by induction on the number of events of H missing from G. The result is immediate when $H = G$, so it suffices to assume that G is missing a single event e. Assume the illegal static serialization of $H \bullet [e\ A]$ has the form $h_1 \bullet e \bullet h_2 \bullet [inv;res] \bullet h_3$. (A similar contradiction can be derived by assuming the illegal serialization has the form $h_1 \bullet [inv;res] \bullet h_2 \bullet e \bullet h_3$.) The serial history $h_1 \bullet h_2 \bullet h_3$ is legal as a static serialization of G, $h_1 \bullet e \bullet h_2 \bullet h_3$ is legal as a static serialization of H, and $h_1 \bullet h_2 \bullet [inv;res] \bullet h_3$ is legal as a static serialization of $G \bullet [inv;res\ A]$. These observations imply that $inv \succ_S e$, a contradiction.

When applied to the *PROM* data type, Theorem 6 shows that static atomicity imposes two constraints that are not needed for hybrid atomicity:

Read() $\succ_S$ Write(x);Ok()
Write(x) $\succ_S$ Read();Ok(y)

These additional constraints have consequences for 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.

Although an object's minimal static dependency can be characterized directly in terms of its serial specification, hybrid dependency has a more complex structure because an object's minimal hybrid dependency relation is not necessarily unique. Quorum assignments under static atomicity have one degree of freedom: a quorum choice is valid if and only if its intersection relation satisfies the unique minimal static dependency relation for that data type. Quorum assignments under hybrid atomicity have an additional degree of freedom: a quorum choice is valid if and only if it satisfies some hybrid dependency relation for that data type. It is an immediate consequence of Theorem 4 that a data type's unique minimal static dependency relation must encompass the union of its minimal hybrid dependency relations.

We close this section with an example of an object having two distinct minimal hybrid dependency relations. A *FlagSet* is an object whose state is modeled by the following components: *opened* and *closed* are boolean flags, and *flags* is a four-element boolean array. All components are initially *False*. If the object has not already been opened, the *Open* operation enables the *Shift* operation and sets *flags[1]* to *True*. Otherwise, an exception is signalled and the invocation has no effect.

```
Open = Operation() Signals (disabled)
    if self.opened
        then signal (disabled)
        else self.opened := true
             self.flags[1] := true
             end
    end Open
```

If the object has been opened but not closed, the *Shift* operation assigns *flags[n]* to *flags[n + 1]*. Otherwise, an exception is signalled and the invocation has no effect. This operation is defined only if $0 < n < 4$.

```
Shift = Operation(n: int) Signals (disabled).
    if self.opened and not self.closed
        then self.flags[n+1] := self.flags[n]
        else signal (disabled)
        end
    end Shift
```

The *Close* operation returns the value of *flag[4]*. If the object has been opened, *Close* disables the *Shift* operation, otherwise it has no effect.

```
Close = Operation() returns (bool)
    self.closed := self.opened
    return (self.flags[4])
    end Close
```

A series of examples can be used to show that the following dependencies must be included in any hybrid dependency relation for *FlagSet*:

Open() $\succ$ Shift(n);Disabled().
Open() $\succ$ Open();Ok()
Close() $\succ$ Shift(n);Ok()
Close() $\succ$ Open();Ok()
Shift(n) $\succ$ Open();Ok() for n = 1,2,3
Shift(n) $\succ$ Close();Ok(x) for n = 1,2,3
Shift(3) $\succ$ Shift(2);Ok()

This relation can be extended to a hybrid dependency relation by the inclusion of either of the following dependencies:

Shift(3) $\succ$ Shift(1);Ok()
Shift(2) $\succ$ Shift(1);Ok()

For brevity, we restrict our attention to the most interesting case: the *Shift(3)* invocation. Let H, G, and $G \bullet$ *[Shift(3);Ok() A]* be behavioral histories in *Hybrid(FlagSet)* such that G is a closed subhistory of H that includes all events e such that *Shift(3)* $\succ e$. $H \bullet$ *[Shift(3);Ok() A]* is in *Hybrid(FlagSet)* if and only if the following conditions hold for every hybrid serialization:

1. The *Shift* is serialized after a normal *Open*, and there is no *Close* serialized between the *Open* and the *Shift*.

2. If the *Shift* sets *flags[4]* to *True*, then the *Shift* is not serialized before a *[Close();Ok(False)]* event.

If G satisfies the first condition, so does H, because *Shift* invocations depend on normal *Open* and *Close* events. The second condition is trivially satisfied if H includes no *Close* events executed by uncommitted actions distinct from A. Otherwise, $H \bullet$ *[Shift(3);Ok() A]* is legal only if the normal *Open* was executed by A itself, and if the *Shift* does not set *Flags[4]* to *True*. Each alternative dependency relation ensures that if A has executed a *Shift(1)* followed by a *Shift(2)*, then these events will appear in G. The alternatives arise because *Shift(1)* events affect the legality of later *Shift(3)* events only if there has been an intermediate *Shift(2)* event. Consequently, *Shift(1)* entries can appear in the view constructed for a *Shift(3)* invocation either because the final and initial quorums of *Shift(1)* and *Shift(3)* intersect directly, or because they intersect indirectly through *Shift(2)*.

5. Strong Dynamic Atomicity

Although quorum consensus replication does not permit the constraints on availability and concurrency to be minimized simultaneously, these constraints do not have a simple inverse relation. Strengthening constraints on concurrency does not necessarily weaken constraints on availability. In this section we consider *strong dynamic atomicity* [28], a generalization of two-phase locking protocols. Strong dynamic atomicity is a special case of hybrid atomicity: if a behavioral history is strong dynamic atomic, then it is hybrid atomic, but not vice-versa. Nevertheless, strong dynamic atomicity and hybrid atomicity have incomparable minimal atomic dependency relations: each

permits quorum assignments not permitted by the other. Static atomicity and strong dynamic atomicity are incomparable with respect to both concurrency and availability.

A behavioral history induces a partial *precedes* order on actions as follows: *A precedes B* if *B* executes an operation after *A* commits. 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 7:** A behavioral history is *strong dynamic atomic* if it is serializable in every order consistent with the partial *precedes* order, and if all such serializations are equivalent.

Since the *precedes* order is clearly compatible with the *Commit* event order, any strong dynamic atomic behavioral history is hybrid atomic, but not vice-versa. A *dynamic serialization* of a behavioral history *H* is constructed by committing some set of active actions in *H* and serializing them in an order consistent with the *precedes* order. Let *Dynamic(T)* denote the largest on-line, prefix-closed strong dynamic atomic behavioral specification for *T*. An atomic dependency relation for *Dynamic(T)* is called a *dynamic dependency relation* for *T*.

> **Definition 8:** 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.

The following lemma is needed to characterize the unique minimal dynamic dependency relation for *Dynamic(T)*.

> **Lemma 9:** If *H* and $H \bullet [e\ A]$ are behavioral 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 ordered last, where *h''* is a dynamic serialization of *H*. Because *H* is in *Dynamic(T)*, $h'' \simeq h$, thus $h' \simeq h \bullet e$.
>
> **Theorem 10:** *T* has the following unique minimal dynamic dependency relation $\succ_D$, defined as follows: $inv \succ_D e$ if there exists a response *res*, such that *[inv;res]* and *e* do not commute.
>
> **Proof:** We first show that every dynamic dependency relation $\succ$ must contain $\succ_D$. 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 behavioral history:
>
> h A
> Commit A
> e' B
>
> and let *G* include all but the last event. *H*, *G*, and $G \bullet [e\ C]$ are in *Dynamic(T)*, but $H \bullet [e\ C]$ is not.
>
> We now show that $\succ_D$ is itself a dynamic dependency relation. Otherwise, there exist behavioral histories *H*, *G*, and $G \bullet [e\ A]$ in *Dynamic(T)* such that *G* is a closed subhistory of *H* under $\succ_D$ containing all events *e'* such that $e.inv \succ_D e'$, but $H \bullet [e\ A]$ is not in

Dynamic(T). We derive a contradiction from these hypotheses.

The proof is by induction on the number of events missing from G. If $G = H$, then the result is immediate, so it suffices to show the result when G is missing a single event $[e' B]$ of H. If $H \bullet [e A]$ is not in *Dynamic(T)*, then there exist serialization orders $\gg$ and $\gg'$ compatible with the *precedes* order 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 events that follow $[e' B]$ in H. The result is immediate if $[e' B]$ is the last event in H, so it suffices to show the result when there is exactly one such event. The result is immediate if that event is a *Begin*, *Commit*, or *Abort*, because no new dynamic serializations are introduced. Otherwise, $H = G_0 \bullet [e' B] \bullet [e'' C]$ and $G = G_0 \bullet [e'' C]$. The events e' and e'' must commute, otherwise G would not be closed. 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 9), and $g_0 \bullet e'$ is legal as a dynamic serialization of $G_0 \bullet [e' B]$. Because e' and e'' commute, $h \simeq g_0 \bullet e' \bullet e'' \simeq g_0 \bullet e'' \bullet e' \simeq g \bullet e'$.

Because $G \bullet [e A]$ and H are in *Dynamic(T)*, $g \bullet e$ and $g \bullet e'$ are both 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 $h' \bullet e \simeq h \bullet e$, contradicting the assumption that $H \bullet [e A]$ is not in *Dynamic(T)*.

Theorem 11: A static dependency relation is not necessarily a dynamic dependency relation.

Proof: We use the *Queue* example from Section 3. By Theorem 6, the following is the unique minimal static dependency relation for *Queue*:

$\text{Enq(x)} \succ_S \text{Deq();Ok(y)}$
$\text{Enq(x)} \succ_S \text{Deq();Empty()}$
$\text{Deq()} \succ_S \text{Enq(x);Ok()}$
$\text{Deq()} \succ_S \text{Deq();Ok(x)}$

By Theorem 10, however, strong dynamic atomicity introduces an additional constraint:

$\text{Enq(x)} \succ_D \text{Enq(y);Ok()}$

The following example illustrates the relation between dynamic and hybrid dependency. 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.

Theorem 12: A dynamic dependency relation is not necessarily a hybrid dependency relation.

Proof: By Theorem 10, the following is the minimal dynamic dependency relation for the *DoubleBuffer* type:

Produce(x) $\succ_D$ Produce(y);Ok()
Produce(x) $\succ_D$ Transfer();Ok()
Transfer() $\succ_D$ Produce(x);Ok()
Consume() $\succ_D$ Transfer();Ok()
Transfer() $\succ_D$ Consume();Ok(x)

The following example shows that $\succ_D$ is not a hybrid dependency relation for *DoubleBuffer*. Let *H* be the following behavioral history:

Produce(x):Ok() A
Transfer();Ok() A
Commit A
Transfer();Ok() C
Produce(y);Ok() B

and let *G* be the history containing all but the last event. *H*, *G* and *G•[Consume();Ok(x) D]* are in *Hybrid(DoubleBuffer)*, and *G* is a closed subhistory of *H* containing all events *e* such that *Consume* $\succ$ *e*, but *H•[Consume();Ok(x) D]* is not in *Hybrid(DoubleBuffer)*, because an illegal serialization results if the active actions commit in the order *B*, *C*, and then *D*.

Theorems 4, 6, and 10 imply that dynamic dependency is incomparable to static dependency and hybrid dependency.

6. 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 [24], and Argus [20] and TABS [27] are based on strong dynamic atomicity. The choice of a system's underlying local atomicity property is an important design decision. The property must be agreed upon in advance, and once made, it is difficult to change.

This paper has proposed a new criterion for evaluating atomicity properties: support for quorum consensus replication. A comparison of the constraints on quorum assignment needed to maximize concurrency under static, hybrid, and strong dynamic atomicity yields different results than a comparison based on concurrency. 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. Availability thus complements concurrency as a criterion for evaluating atomicity mechanisms.

Availability and concurrency are not independent properties. One necessarily affects the other, and an inappropriate local atomicity property may place unnecessary restrictions on the availability and concurrency realizable within a distributed system. The results presented in this paper suggest that hybrid atomicity is preferable to the others because it places fewer constraints on availability then static atomicity, and fewer constraints on concurrency than strong dynamic atomicity.

References

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

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

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

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

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

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

[7] 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.

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

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

[12] 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.

[13] 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.

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

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

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

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

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

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

[20] 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.

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

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

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

[24] 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.

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

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

[27] 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.

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