CMU-CS-84-164R

A Quorum-Consensus Replication Method for Abstract Data Types

Revised Version

Maurice Herlihy
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213
15 October 1985

Abstract

Replication can enhance the availability of data in distributed systems. This paper introduces a new method for managing replicated data. Unlike many methods that support replication only for uninterpreted files, this method systematically exploits type-specific properties of objects such as sets, queues, or directories to provide more effective replication. Each operation requires the co-operation of a certain number of sites for its successful completion. A quorum for an operation is any such set of sites. Necessary and sufficient constraints on quorum intersections are derived from an analysis of the data type's algebraic structure. A reconfiguration method is proposed that permits quorums to be changed dynamically. By taking advantage of type-specific properties in a general and systematic way, this method can realize a wider range of availability properties and more flexible reconfiguration than comparable replication methods.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

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

1. Introduction

Replicated data is data that is stored redundantly at multiple locations. Replication can enhance the availability of data in the presence of failures, increasing the likelihood that the data will be accessible when needed. For example, the availability of a bank account might be enhanced by keeping additional copies of the records at multiple sites. If one set of records becomes temporarily or permanently inaccessible, activities might be able to progress using a different set. Care must be taken that the replicated records are managed properly: enhanced availability may be of little use if activities erroneously observe obsolete or inconsistent data. Consequently, replication is assumed to be *transparent*: its only observable effect is to make the data more available.

This paper introduces *general quorum consensus*, a new method for managing replicated data. A novel aspect of this method is that it systematically exploits type-specific properties of the data to achieve better availability and more flexible reconfiguration than is possible using the conventional read/write classification of operations. Necessary and sufficient constraints on realizable availability properties are derived from an analysis of the data type's algebraic structure. Although our analysis focuses on availability, the techniques introduced here can also be used to enhance reliability and performance.

Section 2 presents a brief overview of related work, and Section 3 describes our assumptions and terminology. Section 4 gives an informal description of our method, and Section 5 gives a formal description with correctness arguments. Section 6 presents some examples. Section 7 proposes a reconfiguration method, and Section 8 discusses some pragmatic considerations. Section 9 summarizes our results.

2. Related Work

Numerous proposals exist for non-transparent replication methods that permit clients to view transient inconsistencies [1, 20, 31, 5, 27, 12]. A discussion of the interaction between transparency and availability appears in Section 6.1.

The replication method proposed here is a generalization of a file replication method due to Gifford [14, 15]. A discussion of Gifford's method appears in Section 4.3. Our method also encompasses a

replication method for directories proposed by Bloch, Daniels, and Spector [6]. Extensions to quorum consensus that further enhance availability in the presence of partitions have been proposed for files by Eager and Sevcik [10] and for arbitrary data types by the author [18]. Garcia-Molina and Barbara [13] have proposed criteria for evaluating the fault-tolerance provided by quorum consensus methods.

In the *true-copy token* scheme [25], 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 is transparent 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.

In the *available copies* replication method [16], 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 [17], and Circus [8]. Unlike quorum consensus methods, these methods do not prevent inconsistencies in the presence of communication link failures such as partitions.

The ISIS project [4, 21] at Cornell is investigating techniques for automatically transforming conventionally structured programs to programs that manipulate replicated data. The ISIS technique preserves consistency in the presence of partitions, but it allows operations to be executed only in the partition that encompasses a majority of 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 [3]. This model, however, relies on two assumptions that do not apply to the replication method proposed 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. We will see that availability is enhanced by violating these assumptions.

A longer and more thorough discussion of replication methods for abstract data types is given in the author's Ph.D. thesis [18], which addresses several issues that lie beyond the scope of this paper,

such as integrating concurrency control with replication, and techniques for further enhancing availability in the presence of partitions. The work described in this paper was originally undertaken as part of the Argus project at M.I.T. [24]. Other projects investigating replication methods include TABS [30], ISIS [4], and Circus [8].

3. Assumptions and Terminology

A *distributed system* consists of multiple computers (called sites) that communicate through a network. Distributed systems are typically subject to two kinds of faults: site crashes and communication link failures. A crash renders a site's data temporarily or permanently inaccessible, while a communication link failure causes messages to be lost. Garbled and out-of-order messages can be detected (with high probability) and discarded. Transient communication failures may be hidden by lower level protocols, but longer-lived failures can cause *partitions*, in which functioning sites are unable to communicate. A failure is detected when a site that has sent a message fails to receive a response after a certain duration. The absence of a response may indicate that the original message was lost, that the reply was lost, that the recipient has crashed, or simply that the recipient is slow to respond.

General quorum consensus relies on certain consistency constraints that must be preserved in the presence of failures and concurrency. These constraints apply not only to individual data items, but also to distributed sets of data. Our approach to this problem is to ensure that activities are *atomic*: that is, indivisible and recoverable. *Indivisible* means that activities appear to execute in a serial order [28], and *recoverable* means that an activity either succeeds completely or has no effect. Atomic activities are called *actions* (or transactions). The replication method presented in this paper is built on top of an atomic action mechanism which we assume is provided by the underlying system. Our replication method is largely independent of the underlying atomicity mechanism; dependencies are discussed in Section 8.

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 FIFO queue might be represented by an object of type *Queue* providing the following operations. *Enq* places an item at the end of the queue:

Enq = Operation(i: Item)

and *Deq* returns the item at the head of the queue:

Deq = Operation() Returns(Item) Signals (Empty).

Deq signals an exception [23] if the queue is empty.

An *event* is a paired operation invocation and response. An object's state is modeled by a sequence of events called a *history*. For example,

Enq(x)/Ok()
Enq(y)/Ok()
Deq()/Ok(x)

is a history for a *Queue*. A *specification* for an object is the set of possible histories for that object. For example, the specification for a *Queue* object consists of histories in which items are dequeued in FIFO order. A *legal* history is one that is included in the object's specification. Specifications are assumed to be prefix-closed: any prefix of a legal history is legal.

4. The Replication Method

This section presents an informal description of our replication method. A formal description is given in Section 5.

4.1. Availability

A *replicated object* is one whose state is stored redundantly at multiple sites. Replicated objects are implemented by two kinds of modules: *repositories* and *front-ends*. Repositories provide long-term storage for the object's state, while front-ends carry out operations for clients. In the terminology of Bernstein and Goodman [2], front-ends correspond roughly to transaction managers and repositories correspond roughly to data managers. 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.

Each operation requires the co-operation of a certain number of repositories for its successful completion. A *quorum* for an operation is any such set of repositories. It is convenient to divide a quorum into two parts: a front-end executing an invocation reads from an *initial quorum* of

repositories, performs a local computation to choose a response, and records the new event at a *final quorum* of repositories. The initial quorum may depend on the invocation, and the final quorum may depend on the response. Either the initial or final quorum may be empty. A *quorum assignment* associates each event with a set of initial and final quorums.

Each event is associated with a logical timestamp [22], which is a value taken from a totally ordered domain. Timestamps associated with different events reflect the order in which they are serialized. For example, if action *A* is serialized before action *B*, then every timestamp generated by *A* is less than every timestamp generated by *B*. Techniques for generating timestamps are discussed in Section 8.

A replicated object's state is represented as a *log*, which is a sequence of *entries*, each consisting of a logical timestamp and an event. The log entries are partially replicated among a set of repositories. For example, the following is a schematic representation of a *Queue* replicated among three repositories. For readability, a "missing" entry at a repository is shown as a blank space.

R1	R2	R3
1:00 Enq(x)/Ok()	1:00 Enq(x)/Ok()	
	1:15 Enq(y)/Ok()	1:15 Enq(y)/Ok()
1:30 Enq(z)/Ok()		1:30 Enq(z)/Ok()

Three items have been enqueued, but no single repository has an entry for all three events.

An operation is executed in the following steps:

1. The client sends the invocation to a front-end, which forwards it to an initial quorum of repositories for that invocation.

2. Each repository in the initial quorum sends its log to the front-end.

3. The front-end merges the logs in timestamp order, discarding duplicates, to construct a history called the *view*. The front-end reconstructs an object state from the view, and chooses a response to the invocation. (The view may not completely reflect the object's current state, but it will contain enough information to choose a correct response.)

4. The front-end generates a new timestamp, appends the new entry to the view, and sends the updated view to a final quorum of repositories for the event. Each repository in the final quorum merges the updated view with its resident log in timestamp order, discarding duplicates, and returns an acknowledgement to the front-end.

5. When a final quorum of repositories has acknowledged the update, the front-end returns

the response to the client.

An action must be aborted if it is unable to complete an operation execution. If the failed operation is executed as a nested action [29, 26], however, the enclosing action need not be aborted.

We remark that logs represent a conceptual model for the replicated data, not a literal design for an implementation. Section 8 describes some optimizations that permit more compact and efficient representations, as well as smaller message sizes. Nevertheless, to avoid digression, we focus on the unoptimized method for now.

4.2. An Example

To illustrate this method, we trace a brief history for a *Queue* replicated among three repositories. In Section 5.1 we give a precise characterization of the constraints governing quorum intersections, but for now we rely on informal arguments. To ascertain the item at the head of the queue, a front-end executing a *Deq* must observe: (i) which items have been enqueued, and (ii) which of these items have since been dequeued. To ensure that these entries appear in the view, each initial quorum for a *Deq* invocation must intersect each final quorum for earlier *Enq* and *Deq* events. Initial *Enq* quorums may empty because *Enq* returns no information about the queue's state.

As discussed in [14], one convenient way to characterize quorums is to assign weighted votes to repositories so that a collection of repositories is a quorum if and only if the sum of its votes exceeds a threshold value. Two quorums will intersect if the sum of their threshold values exceeds the sum of the votes assigned to all repositories. Our examples use voting schemes in which repositories have equal weight. Nevertheless, Barbara and Garcia-Molina [13] have shown that not all quorum assignments can be characterized by weighted voting.

The following example uses *Enq* and *Deq* quorums of (0,2) and (2,2) respectively, where (m,n) means that any m and n repositories respectively constitute an initial and final quorum. The queue is initially empty. An item x is enqueued by appending a log entry with timestamp *1:00* to the empty logs at two repositories, say R1 and R2:

R1	R2	R3
1:00 Enq(x)/Ok()	1:00 Enq(x)/Ok()	

To dequeue x, a front-end merges the logs from R2 and R3, observing that x is the only item in the

queue. The front-end creates a *Deq* entry with timestamp *1:15*, and writes out the entire log to R2 and R3:

R1	R2	R3
1:00 Enq(x)/Ok()	1:00 Enq(x)/Ok()	1:00 Enq(x)/Ok()
	1:15 Deq()/Ok(x)	1:15 Deq()/Ok(x)

Item *y* is then enqueued at R1 and R2 with timestamp *1:30*, and *z* is enqueued at R1 and R3 with timestamp *1:45*.

R1	R2	R3
1:00 Enq(x)/Ok()	1:00 Enq(x)/Ok()	1:00 Enq(x)/Ok()
	1:15 Deq()/Ok(x)	1:15 Deq()/Ok(x)
1:30 Enq(y)/Ok()	1:30 Enq(y)/Ok()	
1:45 Enq(z)/Ok()		1:45 Enq(z)/Ok()

Although the log at each repository defines a legal queue, no single repository contains all items in the queue. Finally, a front-end dequeues *y* by reading from and updating R1 and R3.

R1	R2	R3
1:00 Enq(x)/Ok()	1:00 Enq(x)/Ok()	1:00 Enq(x)/Ok()
1:15 Deq()/Ok(y)	1:15 Deq()/Ok(x)	1:15 Deq()/Ok(x)
1:30 Enq(y)/Ok()	1:30 Enq(y)/Ok()	1:30 Enq(y)/Ok()
1:45 Enq(z)/Ok()		1:45 Enq(z)/Ok()
2:00 Deq()/Ok(y)		2:00 Deq()/Ok(y)

This example illustrates some important points. An operation's quorums determine its availability. Different operations may have different sets of quorums, and hence different levels of availability. Constraints on quorum intersection determine the range of availability trade-offs supported by quorum consensus methods. Given n identical repositories, a replicated *Queue* permits $\lceil n/2 \rceil$ distinct quorum assignments. Because *Enq* and *Deq* quorums must intersect, the availability of one operation can be increased only if the availability of the other is correspondingly decreased. Similarly, because pairs of *Deq* quorums must intersect, *Deq* cannot be more available than *Enq*.

4.3. Remarks

It is instructive to compare general quorum consensus with replication methods for files. A *File* is a container for an uninterpreted value. Files provide two primitive operations: *Read* returns the file's current value, and *Write* replaces the file's current value. (An analysis of a more complex file type permitting access individual pages appears in [18]).

Under general quorum consensus, initial *Read* quorums must intersect final *Write* quorums to ensure

that *Read* returns the most recently written value. Initial and final *Write* quorums need not intersect; if actions *A* and *B* write to disjoint final quorums, then later actions would read the version with the later timestamp. A similar technique has been used for database synchronization, where it is known as the *Thomas Write Rule* [31]. As an obvious optimization, each repository may discard all but the most recent *Write* entry, replacing the log with a timestamped version. A file replicated among n identical repositories permits n distinct quorum assignments.

Gifford's replication method [14] uses version numbers instead of logical timestamps.[1] A front-end writing to a file reads the version numbers from an initial *Write* quorum, generates a version number higher than any it has observed, and records the new version at a final *Write* quorum. To ensure that each new version number is greater than its predecessor, initial and final *Write* quorums must intersect. This additional constraint reduces the number of distinct quorums assignments from n to $\lceil n/2 \rceil$ (*Write* requires a majority). Nevertheless, because files are typically read before they are written, we do not believe that minority *Write* quorums are of major practical interest for files. A more practical advantage of logical timestamps is that *Write* invocations require half as many messages, since there is no need to ascertain the current version number.

More significant advantages of general quorum consensus emerge when we turn our attention to data types other than files. For example, if a FIFO queue were implemented on top of a replicated file, then *Enq* and *Deq* would each be implemented as a *Read* followed by a *Write*, reducing the number of distinct quorum choices from $\lceil n/2 \rceil$ to exactly one (both *Enq* and *Deq* require a majority). General quorum consensus permits additional quorum assignments that facilitate *Enq* at the cost of *Deq*, a trade-off that might be useful in applications such as spoolers. Replacing versions and version numbers with logs and timestamps reduces the constraints on quorum intersection and increases the range of realizable availability properties.

A file replication method proposed by Eager and Sevcik [10] does not require all *Read* and *Write* quorums to intersect. Actions execute in one of two modes: normal or partitioned. In normal mode, actions read from any copy of a file and write to all copies. In partitioned mode, actions use Gifford's

---

[1] In his thesis [15], however, Gifford suggests that logical timestamps might replace version numbers, but no details are presented.

method to read and write a majority of copies. Transparency is preserved by ensuring that partitioned-mode actions are serialized after normal-mode actions. Elsewhere [18, 19], we have shown that general quorum consensus can be integrated with Eager and Sevcik's method to further enhance availability by exploiting type-specific properties.

5. Correctness Arguments

In this section we identify a correct and optimal set of constraints on quorum intersection. By *correct*, we mean that any replicated object whose quorum assignment is consistent with these constraints yields only legal histories. By *optimal*, we mean that there exists no smaller set of constraints that also yields only legal histories.

5.1. Quorum Intersection

Let $\succ$ be a relation between invocations and events. Informally, a subhistory is *closed* under $\succ$ if whenever it contains an event e it also contains every earlier event e' such that $e.inv \succ e'$, where $e.inv$ denotes the invocation part of an event e. More precisely, let $h(i)$ denote the i-th event of h:

> **Definition 1:** A history g is a *closed subhistory* of h with respect to $\succ$ if there exists an injective order-preserving map s such that $g(i) = h(s(i))$ for all i in the domain of g, and if $e.inv \succ e'$, $j > j'$, $h(j) = e$, $h(j') = e'$, and $s(i) = j$, then there exists i' such that $s(i') = j'$.

Informally, $\succ$ is a *serial 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 "$\bullet$" denote concatenation:

> **Definition 2:** A relation $\succ$ is a *serial dependency* relation if
>
> $g \bullet e$ is legal $\Rightarrow h \bullet e$ is legal
>
> for all events e and all legal histories h, whenever g is a closed subhistory containing all events e' such that $e.inv \succ e'$.

For example, the quorum intersection relations for queues and files given in the previous section are serial dependency relations. Additional examples appear in Section 6.

We will show that a replicated object satisfies its specification if and only if its quorum intersection relation is a serial dependency relation. A serial dependency relation is *minimal* if no smaller relation is a serial dependency relation. Minimal relations correspond to minimal sets of constraints. As

illustrated in Section 6.3, a data type may have several distinct minimal serial dependency relations.

5.2. An Automaton Model

This section introduces a formalism for modeling replication. We use the following primitive domains: INV is the set of invocations, RES is the set of responses, REPOS is the set of repositories, and TIMESTAMP is the set of timestamps. We also use the following derived domains: EVENT = INV × RES is the set of events, and QUORUM = 2^{REPOS} is the set of quorums. If X and Y are domains, (X → Y) denotes the set of partial maps from X to Y.

A *log* L is a map from a finite set of timestamps to events.

L: TIMESTAMP → EVENT

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

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

Because the merge operation is defined only for coherent logs, it is commutative and associative. Every log corresponds to a history in the obvious way. For brevity, we sometimes refer to a log L in place of its history, e.g. "L is legal" instead of "the history represented by L is legal." The exact meaning should be clear from context.

A replicated object is modeled as a non-deterministic automaton that accepts certain histories. The automaton's state has two components:

- Log: REPOS → (TIMESTAMP → EVENT)
- Clock: TIMESTAMP

The *Log* component associates a log (initially empty) with each repository, and the *Clock* component models a system of logical clocks, establishing an unambiguous ordering for events.

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

- A serial specification *Spec* ∈ EVENT*.
- *Initial*: INV → 2^{QUORUM} assigns initial quorums to invocations.

- *Final*: EVENT $\rightarrow 2^{\text{QUORUM}}$ assigns final quorums to events.

Initial and *Final* define a *quorum intersection relation* $\succ_Q \subseteq$ INV $\times$ EVENT.

An event e is accepted if the automaton's state satisfies the following precondition: there exists an *initial quorum* IQ $\in$ *Initial(e.inv)*, such that if *Log(IQ)* is the merger of the logs from IQ, then *Log(IQ)* • e is legal. Accepting an event has the following effects, where x' denotes the new value of component x. The clock is advanced:

Clock' > *Clock*.

The new entry is appended to the view, and the updated view is merged with the log at each repository in a *final quorum* FQ $\in$ *Final*(e):

Log'(R)(t) = If R $\notin$ FQ then Log(R)(t)
elseif t = Clock' then e
else (Log(R) $\cup$ Log(IQ))(t)

5.3. Correctness Arguments

We use the following technical lemma.

Lemma 3: If $\succ$ is an arbitrary relation between invocations and events, the result of merging logs closed under $\succ$ is itself closed under $\succ$.

We now identify some invariant properties of quorum consensus automata. Each property clearly holds in the initial state when all logs are empty; we show that each property is preserved across state transitions.

The first step is to show that the view for each invocation is closed under the quorum intersection relation $\succ_Q$.

Lemma 4: The result of merging logs from any set of repositories is closed under $\succ_Q$.

Proof: It suffices to show the property holds for any single repository R; the more general result follows from Lemma 3. The argument is by induction on the length of the accepted history. The base case is trivial, hence the proof focuses on the induction step.

Let e be an event accepted in a state satisfying the lemma. If R is outside the final quorum for e, then *Log(R)* = *Log'(R)*. Otherwise,

Log'(R) = Log(R) $\cup$ Log(IQ) • e

Log(IQ) is closed because it is the merger of closed logs (induction hypothesis and Lemma 3). *Log(IQ)* • *e* is closed by construction, and *Log(R)* is closed (induction hypothesis), therefore *Log'(R)* is closed (Lemma 3).

Because the view for an invocation is the result of merging the logs from the repositories in an initial quorum:

Corollary 5: Each invocation's view is closed under $\succ_Q$.

The next step is to show that the view for each invocation is legal.

Lemma 6: If the quorum intersection relation $\succ_Q$ is a serial dependency relation, then the result of merging logs from any collection of repositories is legal.

Proof: As before, the proof is by induction on the length of the accepted history. Let *S* be an arbitrary set of repositories, and let *Log(S)* and *Log'(S)* be the results of merging the logs from the repositories in *S* respectively before and after accepting a new event *e*. We show that if *Log(S)* is legal, so is *Log'(S)*. If *S* does not intersect the final quorum for *e*, then *Log(S)* = *Log'(S)*, and the result is immediate. Otherwise,

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

Both *Log(IQ)* and (*Log(IQ)* ∪ *Log(S)*) are closed (Corollary 5) and legal (induction hypothesis). *Log(IQ)* is a closed subhistory of (*Log(S)* ∪ *Log(IQ)*) that contains all events *e'* such that *e.inv* $\succ_Q$ *e'*. Because $\succ_Q$ is a serial dependency relation, and *Log(IQ)*•*e* is legal by construction, *(Log(S)* ∪ *Log(IQ))* • *e* = *Log(S)* ∪ *Log(IQ)* • *e* is legal.

This theorem reveals a fail-safety property of quorum consensus: even if a catastrophic failure makes it permanently impossible to assemble a quorum for certain operations, the result of merging the surviving logs yields a legal subhistory of the true (lost) history.

Corollary 7: If $\succ_Q$ is a a serial dependency relation, each invocation's view is legal.

We are now ready to present the basic correctness result:

Theorem 8: If the quorum intersection relation $\succ_Q$ is a serial dependency relation, every history accepted by a quorum consensus automaton is legal.

Proof: Let *Log(IQ)* be the view for *e*, and let *h* be the accepted history. *Log(IQ)* is a closed subhistory of *h* under $\succ_Q$ (Lemma 4), *Log(IQ)* is legal (Lemma 7), and *Log(IQ)* contains every event *e'* such that *e.inv* $\succ_Q$ *e'*. Because *Log(IQ)*•*e* is legal and $\succ_Q$ is a serial dependency relation, *h*•*e* is also legal.

We now show that no set of constraints on quorum intersection weaker than serial dependency

guarantees that all histories accepted by a quorum consensus automaton are legal.

Theorem 9: Given a relation $\succ_Q$ that is not a serial dependency relation, there exists an automaton whose quorum intersection relation satisfies $\succ_Q$ that accepts an illegal history.

Proof: Given such a relation, we construct a scenario in which an illegal history is accepted.

If $\succ_Q$ is not a serial dependency relation, there exists an event e, a legal history h having a closed history g containing all e' of h such that $e.inv \succ_Q e'$, with the property that:

$g \cdot e$ is legal but $h \cdot e$ is illegal.

We construct an automaton with quorum intersection relation $\succ_Q$ that accepts $h \cdot e$.

The automaton has two repositories: R1 and R2. It accepts h, choosing the following quorums for each event. For events in g, it chooses an initial quorum of R1 and a final quorum of both R1 and R2. For events in h but not in g, it chooses an initial quorum of R1 and R2 and a final quorum of R2. The view for each event in g thus contains all and only the prior events in g, and the view for every other event contains all prior events.

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

When a front-end assembles the view for $e.inv$, it reads g from R1, which is a valid initial quorum for $e.inv$ because it intersects the final quorum for each event in g. The front-end then generates the event e, which by assumption is legal for g, but illegal for h.

6. Examples

This section presents three examples of serial dependency relations. In the first part, we discuss the constraints on replicated directories, both deterministic and non-deterministic. In the second part, we discuss a replicated counter type that will be used for reconfiguration, and in the last part we present an example of a data type whose minimal serial dependency relation is not unique.

6.1. Directories

A *Directory* stores pairs of values, where one value (the *key*) is used to retrieve the other (the *item*).

Insert = Operation(k: Key, i: Item) signals (Present)

inserts a new binding in the directory, signalling if the key is already present.

Change = Operation(k: Key, i: Item) signals (Absent)

alters the item bound to the given key, signalling if the key is absent.

Lookup = Operation(k: Key) returns(item) signals (Absent)

returns the item bound to the given key, signalling if the key is absent.

Size = Operation() returns(int)

returns the number of key-item pairs currently in the directory.

	Insert(k,*)	Change(k,*)	Size()	Lookup(k)
Insert(k,*)/Ok()	X	X	X	X
Change(k,*)/Ok()				X

Table 6-1: Dependency Relation for Directory

The minimal serial dependency relation for *Directory* is shown in Table 6-1. Quorums for *Directory* operations may depend on the key value. For example, one might use different sets of repositories to store payroll records for East and West coast employees. If quorums do not depend on key values, a simple combinatorial argument shows that a *Directory* replicated among n identical repositories permits $\lceil n/2 \rceil^2$ distinct quorum assignments. If version numbers are used in place of logical timestamps, as in [6], then *Change* quorums must intersect, reducing the number of distinct quorum assignments to $\lceil n/2 \rceil$.

For the *Directory* type, the levels of availability of *Lookup* and *Insert* for a particular key are inversely related. For example, if a binding is replicated among n repositories, and if the initial quorum for *Lookup* consists of a single repository, then the final quorum for *Insert* must consist of all n repositories. To circumvent this trade-off, several proposals for replicated directories have chosen to rely on probabilistic guarantees of correctness [20, 31, 5, 27, 12]. A binding is inserted or altered at a single repository, and the update is later propagated to the other repositories. This approach has the advantage that updates complete more quickly and are more likely to succeed, but it has the disadvantage that the behavior of the server becomes considerably more complex because clients may observe transient "inconsistencies" in the directory, and concurrent conflicting updates must

somehow be resolved.

There are two ways to view such "transient inconsistencies." One view is that atomicity has been sacrificed for increased availability. The other view holds that the resulting data type continues to be atomic, but that it can no longer be considered a deterministic map from keys to items. We prefer the second view for its economy of mechanism: serializability still characterizes the properties upon which the programmer may rely, and general quorum consensus can be used to implement a replicated non-deterministic directory having the same advantages as the "non-atomic" methods cited above.

A *SemiDirectory* is a map from keys to multisets of items. When the directory is created, it contains (conceptually, at least) a binding between every key and a multiset containing only the distinguished item *Absent*. The invocation *Insert(k,i)* adds the item *i* to the multiset associated with *k*, and the invocation *Lookup(k)* returns some item previously bound to *k*, or signals *Absent*. There is a probabilistic guarantee that any item returned is likely to be the most recently inserted one.

The minimal serial dependency relation for *SemiDirectory* is degenerate: no invocation depends on any event, and thus no quorums are required to intersect. The view constructed for *Lookup(k)* will include the most recent binding for that key if the initial quorum for *Lookup* happens to intersect the final quorum for the *Insert* or *Change*. The *probability* that *Lookup* will observe an item is thus the probability that the quorums will intersect. That probability is unity for the *Directory* type, and would be less for a non-deterministic implementation. If both *Insert* and *Lookup* choose quorums of single repositories, then the probability of intersection may be small. To cause the probability of intersection to rise with time, the log entry for *Insert* can be propagated by a background activity, effectively causing the final quorum for *Insert* to grow. In a satisfactory implementation of *SemiDirectory*, a *Lookup* invocation would choose the most recently inserted item from the view, and insertions would be propagated quickly enough so that the view is sufficiently likely to contain the most recently inserted item.

6.2. Counters

The *Counter* data type will be used in the reconfiguration method proposed in Section 7. A *Counter* stores an integer value, initially zero. The *Inc* operation increments the value by one:

Inc = Operation()

and the *Dec* operation decrements the value by one:

Dec = Operation().

The *Value* operation returns the counter's current value:

Value = Operation() Returns(Int)

Initial quorums for *Value* must intersect final quorums for *Inc* and *Dec*, but initial quorums for *Inc* and *Dec* may be empty, because neither operation returns any information. A *Counter* replicated among *n* identical repositories permits *n* distinct quorum assignments. (Just as for queues, a file-based implementation would permit only one quorum assignment.)

6.3. The DoubleBuffer Type

The *DoubleBuffer* type illustrates that a data type's minimal serial dependency relation need not be unique. A *DoubleBuffer* consists of a *producer* buffer and a *consumer* buffer, each capable of holding a single item. The object is initialized with a default item in each buffer. The *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 *DoubleBuffer* type supports two distinct serial dependency relations. In the first relation, shown in Table 6-2, *Consume* invocations depend on both *Transfer* and *Produce* events. In the second, shown in Table 6-3, *Consume* invocations depend on *Transfer* events, and *Transfer* invocations depend on *Produce* events.

	Transfer()	Consume()
Produce(x)/Ok()		X
Transfer()/Ok()		X

Table 6-2: First Dependency Relation for DoubleBuffer

	Transfer()	Consume()
Produce(x)/Ok()	X	
Transfer()/Ok()		X

Table 6-3: Second Dependency Relation for DoubleBuffer

The alternatives arise because *Produce* events affect later *Consume* invocations only if there has been an intermediate *Transfer*. Consequently, *Produce* entries can appear in the view constructed for a *Consume* invocation either because the final and initial quorums of *Produce* and *Consume* intersect directly, or because they intersect indirectly through *Transfer*. Quorums for this type may be chosen with two degrees of freedom: one first chooses a serial dependency relation, and then one chooses particular quorums subject to the constraints imposed by the serial dependency relation.

Neither serial dependency relation is strictly preferable to the other. For example, if *Produce* has quorums of (1,0), quorums for *Consume* and *Transfer* are (5,0) and (0,1) under the first relation, and (5,0) and (5,1) under the second. The first relation is thus preferable for maximizing the availability of *Produce*. On the other hand, if *Consume* has quorums of (1,0), quorums for *Produce* and *Transfer* are (0,5) and (0,5) under the first relation, and (0,1) and (5,5) under the second. The second relation is thus preferable for maximizing the availability of *Consume*. A *DoubleBuffer* replicated among n identical repositories permits $2n$ distinct quorum assignments, n for each relation. A *DoubleBuffer* implemented as a pair of replicated files with version numbers would permit $\lceil n/2 \rceil$ distinct quorum choices.

7. Reconfiguration

In this section we extend general quorum consensus to support on-the-fly *reconfiguration*: changing the quorum assignment for an existing object. Reconfiguration may be used to change the trade-offs among the levels of availability provided by an object's operations. For example, a census data base might be configured to facilitate updates while the census is in progress, and reconfigured to facilitate queries once the census is complete. Reconfiguration may also be used to move an object from one collection of sites to another, perhaps to replace malfunctioning or obsolescent hardware. A benefit of the reconfiguration method proposed here is that it incurs a negligible cost when it is not used. Reconfiguration results in a temporary period of decreased availability and increased message traffic as front-ends learn of the new configuration.

This reconfiguration scheme can be viewed as a generalization of a scheme proposed by Gifford [14]. Our scheme gains flexibility both by taking advantage of type information, and by incorporating a novel replicated reference counting scheme that facilitates the movement of objects from one set of sites to another.

A reconfigurable object is implemented by two kinds of components: a resource state and a chain of configuration states. Although these components are logically distinct, they are replicated at the same repositories. The *resource state* records information of direct interest to clients (e.g. the items in a queue), while the *configuration states* record information about quorum assignments (e.g. *Enq* and *Deq* quorums). A newly-created configuration state is marked *current*, and is used to store the quorum assignment for the resource state. A new configuration state is created each time the object is reconfigured, and the previous configuration state is marked *obsolete*, and is used to store the quorum assignment for the newer configuration state. Each configuration state provides *GetQuorum* and *SetQuorum* operations.

An object is reconfigured in the following steps:

1. Construct a view of the old resource state by merging the logs from an old initial quorum for each operation.

2. Initialize the new resource state by writing out the view to a new final quorum for each operation.

3. Initialize the new configuration state by writing out the new resource state quorum assignments to a new *SetQuorum* quorum.

4. Update the old configuration state by writing out the new configuration state quorum assignments to an old *SetQuorum* quorum.

A quorum for reconfiguring the object must include a quorum for each of these steps.

Each front-end keeps a local cache containing the quorum information for both the configuration state and the resource state. An operation is normally executed in two logical steps, although we will see that only one exchange of physical messages is required.

- Verify that the cached configuration state is current.

- Operate directly on the resource state using the cached quorums.

If the front-end discovers that its cached configuration state is out of date, it reads the quorum information for the new configuration state, reads the new configuration state into its cache, and starts over. If several reconfigurations have taken place, a front-end may have to follow a chain of configuration states to locate the resource state.

Because the configuration and resource states are replicated among the same set of repositories, the number of messages needed for this protocol can be kept to a minimum if each quorum for each resource state operation is also a *GetQuorum* quorum for the associated configuration state. Each front-end includes a timestamp for its cached configuration state in every message directed to a repository. When a repository receives a message, it compares the cache timestamp with its local configuration state timestamp. If they are the same, the repository carries out the front-end's request, otherwise it responds with an exception identifying the newer configuration state.

A shortcoming of the scheme described so far is that there is no mechanism for safely discarding obsolete configuration states. For example, if a replicated object is moved from one set of repositories to another, the configuration state at the old set of repositories cannot be discarded as

long as there is a possibility that some front-end's cache is still out of date. To remedy this problem, we propose a reference counting scheme that enables the object's maintainers to detect when all front-ends have updated their configuration states. This scheme has the desirable property that it does not affect the availability of the replicated object, although it does require extra messages immediately following a reconfiguration.

A *Counter* as defined above in Section 6.2 is replicated at the object's repositories. When a front-end first records the configuration state in its cache, it records an *Inc* entry at a quorum of the configuration state's repositories. When a front-end observes that the configuration state has been rendered obsolete, it records a *Dec* entry at a quorum of repositories. Once a configuration state has been rendered obsolete, the object's maintainers may invoke the *Value* operation, merging the *Inc* and *Dec* entries to count the front-ends whose caches are still out of date. A configuration state may be discarded when its reference count and the reference counts of all earlier configuration states have reached zero. (Because there are no cycles of reference between configuration states, an unneeded configuration state will always have a reference count of zero.)

Reference counting need not reduce the object's availability if quorums are assigned so that every quorum for *GetQuorum* is also a quorum for *Inc* and *Dec*. A front-end must locate an initial *GetQuorum* quorum for a configuration state to discover that it has become obsolete. That same quorum can be used to decrement the old configuration state's reference counter, although a second round of messages is necessary. Before the front-end can use the new configuration state, it must check its currency by locating an initial *GetQuorum* quorum. That same quorum can be used to increment the new configuration state's reference counter.

We close this section with an example illustrating how a replicated queue might be reconfigured. Initially, the queue is replicated among R1, R2, and R3, having respective *Enq* and *Deq* quorums of (0,2) and (2,2). As described above, the resource state quorums determine the configuration state quorums: *GetQuorum* and *Value* have quorums of (2,0), while *SetQuorum*, *Inc*,and *Dec* each have quorums of (0,2). After reconfiguration, the queue is replicated among R1', R2', and R3', having *Enq* quorums of (0,1) and *Deq* quorums of (3,1). *GetQuorum* and *SetQuorum* must have respective quorums of (1,0) and (0,3), while *Inc*, *Dec*, and *Value* must have respective quorums of (0,1), (0,1), and

(3,0).

The queue's original configuration state is schematically represented as follows. (The resource state is not shown.)

R1	R2	R3
1:00	1:00	1:00
{R1,R2,R3}	{R1,R2,R3}	
Enq = (0,2)	Enq = (0,2)	
Deq = (2,2)	Deq = (2,2)	
1:15 Inc()/Ok()	1:15 Inc()/Ok()	
	1:30 Inc()/Ok()	1:30 Inc()/Ok()
1:45 Inc()/Ok()		1:45 Inc()/Ok()

The first line indicates that each configuration state has a timestamp of *1:00*. The next three lines represent the quorum information for the resource state. The resource state's quorum information is recorded at only two repositories, since these constitute a *SetQuorum* quorum. The final three lines record the status of the configuration state's reference count. In this example, the counter has been incremented by three front-ends, although no single repository records all three increments. Each front-end includes the cache timestamp *1:00* with every message it sends to a repository.

The queue is reconfigured as follows.

- Merge the logs from any two repositories in the old resource state.
- Write out the resulting view to all three repositories in the new resource state.
- Record the new resource state quorums at all three repositories in the new configuration state.
- Record the new configuration state's quorums at any two repositories from the old configuration state.

Following reconfiguration, the old and new configuration states might appear as follows.

R1	R2	R3
1:00	2:00	2:00
{R1,R2,R3}	{R1',R2',R3'}	{R1',R2',R3'}
Enq = (0,2)	GetQ = (1,0)	GetQ = (1,0)
Deq = (2,2)	SetQ = (0,3)	SetQ = (0,3)
1:15 Inc()/Ok()	1:15 Inc()/Ok()	
	1:30 Inc()/Ok()	1:30 Inc()/Ok()
1:45 Inc()/Ok()		1:45 Inc()/Ok()
R1'	R2'	R3'
2:15	2:15	2:15
{R1',R2',R3'}	{R1',R2',R3'}	{R1',R2',R3'}
Enq = (0,3)	Enq = (0,3)	Enq = (0,3)
Deq = (3,1)	Deq = (3,1)	Deq = (3,1)

Now suppose a front-end whose cache is out of date attempts to enqueue or dequeue an item. The front-end's message includes a cache timestamp, and any quorum it chooses must include at least one repository that will detect that the timestamp is obsolete. That repository will return an exception identifying the new configuration state. The front-end decrements the old configuration state's reference count at two repositories, and uses a single exchange of messages to read the new configuration state into its cache and to increment the new configuration state's reference count.

R1	R2	R3
1:00	2:00	2:00
{R1,R2,R3}	{R1',R2',R3'}	{R1',R2',R3'}
Enq = (0,2)	GetQ = (1,0)	GetQ = (1,0)
Deq = (2,2)	SetQ = (0,3)	SetQ = (0,3)
1:15 Inc()/Ok()	1:15 Inc()/Ok()	
	1:30 Inc()/Ok()	1:30 Inc()/Ok()
1:45 Inc()/Ok()		1:45 Inc()/Ok()
2:30 Dec()/Ok()	2:30 Dec()/Ok()	
R1'	R2'	R3'
2:15	2:15	2:15
{R1',R2',R3'}	{R1',R2',R3'}	{R1',R2',R3'}
Enq = (0,3)	Enq = (0,3)	Enq = (0,3)
Deq = (3,1)	Deq = (3,1)	Deq = (3,1)
2:45 Inc()/Ok()		

Here, the system maintainer can determine that it is not yet safe to discard the old configuration state because its reference count indicates that there are still two front-ends that have not updated their caches.

In summary, the reconfiguration method described here incurs a non-negligible cost only when

reconfiguration actually takes place. Under normal circumstances, availability is unaffected because each quorum for operating on the resource state alone is a quorum for the operation as a whole. No additional messages are needed because a front-end can use the same physical messages for two logically distinct tasks: to establish the currency of its cached configuration state, and to apply the operation to the resource state. Reconfiguration does impose a one-time penalty on a front-end whose cache is out of date: the next time it attempts to execute an operation it must conduct an additional exchange of messages and locate an additional quorum, and an extra round of messages is needed to update the configuration state's reference counter.

8. Pragmatic Issues

This section addresses two pragmatic issues raised by replication: logical timestamp generation, and log and message compaction.

8.1. Logical Timestamp Generation

Logical timestamps are structured as follows:

- The high-order bits are occupied by an *action timestamp* that defines how the generating action is serialized relative to other actions. This field is used to compare timestamps generated by distinct actions.

- The low-order bits are occupied by values read from a logical clock private to the generating action. This field is used to compare timestamps generated within a single action.

We describe two timestamp generation schemes, one intended for systems in which actions are serialized in a predetermined order, and one for systems in which actions are synchronized dynamically through conflicts over shared data.

Under Reed's multiversion timestamp scheme [29], each action is given a *pseudotime* on initialization, and scheduling constraints ensure that actions remain serializable in pseudotime order. Under this scheme, an action's pseudotime is its timestamp.

Under two-phase locking [11], a slightly more complicated scheme is required because the eventual serialization ordering is unknown in advance. At the time an action generates a timestamp, the high-order field is left empty. A timestamp generated by an uncommitted action is considered later

than a timestamp generated by a committed action (timestamps generated by distinct uncommitted actions are never compared). When the action commits, a value read from a logical clock [22] is used as the action timestamp. Similar timestamp generation schemes have been proposed by Dubourdieu [9], by Chan et al. [7], and by Weihl [33].

8.2. Log and Message Compaction

Logs have the disadvantage that they are neither compact nor efficient. For example, the size of a log representing a *Queue* has no relation to the number of items present in the queue, and the item at the head of the queue must be found by a linear search. These problems can be alleviated by replacing logs with more compact and efficient representations. We assume each object's state can be represented compactly and efficiently at a single site. Such a representation is called a *version*. For example, a *Queue* might be represented as an array or linked list, a *Counter* as an integer cell, and a *Directory* as a hash table or B-tree.

If a repository's log has gaps, it cannot be replaced by a version. Divergent logs can be reconciled by merger, but there is no systematic way to reconcile divergent versions. Under what circumstances may a version replace a log? Histories g and h are *equivalent* if $g \bullet \alpha$ is legal if and only if $h \bullet \alpha$ is legal, for all event sequences α. If a repository's log has a prefix equivalent to a prefix of the object's complete history, then that prefix may be replaced by a single version. A version that replaces a prefix is given the timestamp of the last entry it replaces.

Each repository stores a single timestamped version together with a (potentially incomplete) sequence of log entries with later timestamps. A front-end executing an operation collects the logs and versions from an initial quorum, reconstructing the object's state by selecting the most recent version and applying any later log entries, treating the entries as a *differential file* [32].

Logs are compacted by the following protocol. A front-end constructs a timestamped version from an initial quorum for each operation, and sents the version to as many repositories as possible. Each repository that receives the new version discards all earlier versions and entries. Once the new version is recorded at a final quorum for each operation, only the version's timestamp need be forwarded to the remaining repositories, because henceforth every invocation's initial quorum will include at least one repository with the new version.

Logs may be compacted by background processes, or as a side-effect of certain operations. For example, because an initial quorum for *Deq* is an initial quorum for any *Queue* operation, logs may be compacted whenever an item is dequeued. For example, consider the following queue:

R1	R2	R3
1:00 Enq(x)/Ok()	1:00 Enq(x)/Ok()	
	1:15 Enq(y)/Ok()	1:15 Enq(y)/Ok()
1:30 Enq(z)/Ok()		1:30 Enq(z)/Ok()

A front-end executing a *Deq* merges the logs from R1 and R2, creates a version, dequeues *x*, and writes out the updated version to R1 and R2 with a new timestamp.

R1	R2	R3
		1:15 Enq(y)/Ok()
		1:30 Enq(z)/Ok()
1:45 [y,z]	1:45 [y,z]	

When R1 and R2 have confirmed the update, the front-end can forward the version timestamp to R2, which may then discard all earlier entries and versions.

Message sizes can be reduced if each repository caches its most recently observed version. When a front-end requests a repository's log, it need only request entries or versions with later timestamps. If the object's state can be expressed as a Cartesian product, messages sizes can be further reduced by compacting each component independently. For example, in a proposal of Bloch, Daniels, and Spector [6], a replicated *Directory* is represented as the product of key/item bindings, each with its own version number. A type-specific compaction technique is used to prevent the accumulation of entries for deleted bindings.

These compaction methods are probabilistic, since they can be frustrated by an inopportune distribution of events and failures. For example, if a repository for a *Queue* joins *Enq* quorums but not *Deq* quorums, and if communication failures prevent later compaction by background processes, then that repository will continue to accumulate *Enq* entries. Reducing the likelihood of such scenarios to an acceptable level is an engineering problem best addressed experimentally.

9. Discussion

This paper has proposed a new replication method that systematically exploits type-specific properties to provide more effective replication in the presence of crashes and partitions. A summary of our results follows.

- Constraints on quorum intersection are reduced by replacing versions and version numbers by logs and logical timestamps.
- The problem of identifying an object's minimal constraints on quorum assignments (and availability) is reduced to the algebraic problem of identifying its minimal serial dependency relations.
- Following a catastrophic failure, the surviving data represents a legal subhistory of the complete (lost) history.
- Dynamic reconfiguration can be accomplished without reducing availability by replicating an object's resource and configuration states so that each quorum for a resource state operation is also a quorum for reading the configuration state.
- A novel replicated reference counting scheme permits objects to be moved from one set of repositories to another.

General quorum consensus is systematic, general, and effective. It is general because it is applicable to objects of arbitrary type; it is systematic because constraints on correct implementations are derived directly from the data type specification; it is effective because it imposes fewer constraints on availability and more flexible reconfiguration than permitted by the conventional read/write classification of operations.

Acknowledgments

I would like to thank Joshua Bloch, Dean Daniels, Amr El-Abbadi, Dave Gifford, Tommy Joseph, and the referees for careful readings of earlier drafts of this paper.

References

[1] Alsberg, P. A., and Day, J. D.
A principle for resilient sharing of distributed resources.
In *Proceedings, 2nd Annual Conference on Software Engineering*. October, 1976.

[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.
Replication and Fault-Tolerance in the ISIS System.
In *Proc. 10th Symposium on Operating Systems Principles*. dec, 1985.
Also TR 85-668, Cornell University Computer Science Dept.

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

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

[7] 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] 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.

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

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

[11] 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.

[12] Fischer, M., and Michael, A.
Sacrificing serializability to attain high availability of data in an unreliable network.
In *Proceedings, ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*. March, 1982.

[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] Gifford, D. K.
Information Storage in a Decentralized Computer System.
Technical Report CSL-81-8, Xerox Corporation, March, 1982.

[16] 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.

[17] 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, dec, 1980.

[18] 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.

[19] Herlihy, M. P.
Using type information to enhance the availability of partitioned data..
Technical Report CMU-CS-85-119, Carnegie-Mellon University, April, 1985.

[20] Johnson, P. R., and Thomas, R. H.
The maintenance of duplicate databases.
Technical Report RFC 677 NIC 31507, Network Working Group, January, 1975.

[21] Joseph, T. and Birman, K. P.
Low-cost management of replicated data in distributed systems.
To appear, ACM TOCS.

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

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

[24] 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.

[25] 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.

[26] 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.

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

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

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

[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.
TOSE 11(6):520-530, June, 1985.

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

[32] Verhofstad, J. S. M.
Recovery Techniques for Database Systems.
ACM Computing Surveys 10(2):167-196, June, 1978.

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