NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

CMU-CS-84-164

510.7808 C28r 84-164 c.2

General Quorum Consensus: A Replication Method for Abstract Data Types

Maurice Herlihy
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213
12 December 1984

Abstract

Replication can enhance the availability of data in distributed systems. This paper introduces General Quorum Consensus: 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. 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. 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, General quorum consensus can realize a wider range of availability properties and more flexible reconfiguration than existing replication methods.

Copyright © 1984 Maurice Herlihy

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.

University Libraries
Carnegie Mellon University
Pittsburgh PA 15213-3890

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 it is needed. Replication is a useful technique for systems in which availability is important, such as banking systems, airline reservation systems, authentication servers, and mail systems. This paper introduces *general quorum consensus*: a new method for managing replicated data. Unlike many methods that support replication only for uninterpreted files, general quorum consensus makes use of type-specific properties of objects (such as sets, queues, or directories) to provide more effective replication. The method is both general and systematic. It is general because it is applicable to objects of arbitrary type, and it is systematic because constraints on correct implementations are derived directly from algebraic structure of the data type in question. These constraints are both necessary and sufficient: any replicated implementation satisfying these constraints is correct, and no smaller set of constraints guarantees correctness.

Our model of computation is presented in Section 2, and related work is discussed in Section 3. In Section 4, we review a replication method for files due to Gifford, observing that certain difficulties arise when one attempts to implement objects of arbitrary abstract type on top of a replicated file. In Section 5, we introduce general quorum consensus: a revised replication method that is capable of realizing a wider range of availability properties. In Section 6, we present the correctness arguments for our method, and Section 7 presents a series of examples. Section 8 proposes a scheme for on-the-fly reconfiguration, and Section 9 concludes with a discussion.

2. Model of Computation

A distributed system consists of a collection of *sites* that communicate through a network. A site consists of one or more processors, one or more levels of memory, and any number of devices. We assume that any site can communicate with any other when the network is functioning properly. We make no assumptions about the speed, connectivity, or reliability of the network.

Our model admits two kinds of failures: site crashes and communication failures. When a

site crashes, its resident data becomes temporarily or permanently inaccessible. We assume that all communication failures take the form of 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 render functioning sites unable to communicate. For example, a network *partition* results if the sites are divided into disjoint sets such that functioning members of distinct sets cannot communicate. Similar failures may leave one site able to send messages to another, but not vice-versa.

A failure is detected when a site that has sent a message fails to receive a response after a certain duration. Although we assume that site crashes and lost messages can be detected by the absence of a response, we do not assume that the different kinds of failure can be distinguished: the absence of a response may mean 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 pieces of data, but also to distributed sets of data. Our approach to this problem is to ensure that activities are *atomic*: that is, indivisible and recoverable. By *indivisible*, we mean the execution of one activity never appears to overlap (or contain) the execution of another, and by *recoverable*, we mean the complete effect of an activity is all-or-nothing: it either succeeds completely, or it has no effect. Atomic activities are called *actions* (or transactions). An action that completes all its changes successfully *commits*; otherwise it *aborts*, and the data objects it has modified are restored to their previous states. The effect of executing multiple concurrent actions is *serializable* [10, 25]: their complete effect is as if they had been executed in some sequential order.

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. Actions may be serialized either in a predetermined timestamp order [26], or dynamically through shared locks [10]. Elsewhere [17], we show that general quorum consensus can support a higher level of concurrency if it is integrated with the action mechanism. Such integration does

not, however, extend the range of availability properties realizable by our method. This paper focuses on availability properties; a consideration of the interaction of concurrency control with replication lies beyond the scope of this paper.

The basic containers for data are called *objects*. Each object has a *type*, which characterizes its behavior by defining a set of possible *states* together with a set of primitive *operations* that provide the (only) means to create and manipulate objects of that type. Each type has an accompanying *specification* that gives the meaning of the operations provided by the type. 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. In the terminology of Bernstein and Goodman [2], front-ends correspond roughly to transaction managers and repositories correspond roughly to data managers. To apply an operation to a replicated object, a client sends an invocation to a front-end for the object. The front-end reads the data from some collection of repositories, carries out a local computation, sends updates to some collection of repositories, and returns the response to the client. Each operation is executed as part of an atomic action.

An operation's availability to a client is determined by two factors: the client must first be able to locate a front-end for the object, and the front-end must in turn locate enough repositories to execute the operation. Because the front-ends do not interact directly with one another, new front-ends can be created without affecting existing ones. Consequently, front-ends can be replicated to an arbitrary extent, implying that the availability of the replicated object is dominated by the availability of the repositories.

3. Related Work

The replication method described in this paper can be viewed as a generalization of a file replication method due to Gifford [12, 13]. A detailed discussion of Gifford's method appears in Section 4.

Bloch, Daniels, and Spector [8, 6] have adapted Gifford's file replication method to implement replicated directories. This directory replication method can also be viewed as a specially optimized instance of the general method described here (see Section 7.3). These

papers focus on minimizing storage consumption and message traffic for directories, while we focus on how an object's type structure determines its range of realizable availability properties.

Early file replication methods did not attempt to preserve serializability: the value read from a file is not necessarily the value most recently written [1, 18, 27]. Replication methods for directories have been proposed with the same property: clients may observe obsolete bindings between pairs of values [24, 5, 11]. By sacrificing serializability, these schemes enhance performance and availability at the cost of more complex behavior by the replicated object. General quorum consensus permits a similar trade-off through the introduction of non-determinism, as discussed in Section 7.4.

In the *true-copy token* scheme [23], a replicated file is represented by a collection of versions. Versions that reflect the file's current state are called *true copies*. There are two kinds of true copies: there may be a unique *exclusive* copy, used for both reading and writing, or there may be multiple *shared* copies, used only for reading. A true copy is marked by a *true-copy token*, which also indicates whether the copy is shared or exclusive. Under our assumption that site failures cannot be distinguished from partitions, the failure of a single site containing a true-copy token limits the availability of the entire replicated file. If an exclusive copy token becomes unavailable, the replicated file can neither be read nor written, and if a shared copy token becomes unavailable, then the file can be read but cannot be reconfigured for writing. Consequently, the true-copy token scheme can be used to enhance performance by allowing actions to operate on local copies of the file, but it does not enhance availability in the presence of communication failures.

In the *available copies* method [14], 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 [16] and ISIS [4]. These methods are not directly comparable to ours because they make more optimistic assumptions about faults. While our methods tolerate both site crashes and communication failures, the available copies method tolerates only site crashes. Following a communication failure such as a partition, the available copies method would permit each partition's sites to configure out the others, allowing data

in distinct partitions to diverge irreconcilably.

Bernstein and Goodman have proposed a formal model for concurrency control in replicated databases [3]. This model is used to show the correctness of several replication methods for files. Implicit in this model are two *a priori* assumptions that unnecessarily restrict availability. The basic correctness criterion, "one-copy serializability," assumes that a replicated object is implemented by multiple copies, and that all information about operations can be captured by a simple read/write classification. Neither assumption is valid for the replication method proposed in this paper.

A longer and more thorough discussion of replication methods for abstract data types is given in the author's Ph.D. thesis [17], 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. [21, 22]. Argus is a programming language and system that supports the construction of robust distributed programs.

4. Gifford's Quorum Consensus Method

In this section we describe a replicated method due to Gifford [12, 13]. This description serves two purposes: it introduces some important ideas, and it illustrates the novel aspects of general quorum consensus. We first show how Gifford's method can be used to construct both a replicated file and a replicated FIFO queue. In the next section we introduce a revised replication method that supports a wider range of availability properties for both files and queues, and we then generalize the revised replication method to arbitrary data types.

A *quorum* for an operation is defined to be any set of repositories whose cooperation suffices to execute that operation. It is convenient to divide a quorum into two parts: a front-end executing an operation reads from an *initial quorum* and writes to a *final quorum*. (Either the initial or final quorum may be empty.) A quorum for an operation is any set of repositories that includes both an initial and a final quorum.

For our purposes, a *File* is a container for a string. Files provide two primitive operations: *Read* returns the file's current value, and *Write* sets the file to a new value. To simplify our discussion, we assume that *Write* completely replaces the file's previous contents. This assumption will be relaxed in Section 7.

An *invocation* consists of an operation name and argument values, a *response* consists of a termination condition and results, and an *event* is a pair consisting of an invocation and its associated response. For example, *Read()* and *Write(x)* are invocations, *Ok(x)* and *Ok()* are their respective responses, and *[Read();Ok(x)]* and *[Write(x);Ok()]* are events.

4.1. A Replicated File Implementation

In Gifford's method, each repository stores a version of the file together with a *version number*, which is used to recognize the most recent version. To read from a file, the front-end reads the versions from an initial *Read* quorum of sites, returning the version with the greatest version number to the client. (The final quorum for *Read* is empty.) To write to a file, the front-end first reads the version numbers from an initial *Write* quorum of repositories. The front-end then creates a new version of the file with a version number greater than any it has observed, and writes out the new version to a final *Write* quorum.

Quorums for file operations are subject to two constraints:

- Each final quorum for *Write* must intersect each initial quorum for *Read*.
- Each final quorum for *Write* must intersect each initial quorum for *Write*.

The first constraint ensures that a front-end executing a *Read* invocation will observe the effects of the most recent *Write*, and the second constraint ensures that each new version created by a *Write* invocation will have a greater version number than its predecessors. As a consequence, each *Read* quorum must intersect each *Write* quorum, and each pair of *Write* quorums must intersect.

These constraints can be summarized by a *quorum intersection graph*. The vertices of the graph correspond to classes of events. A directed edge from one vertex to another means that each final quorum for operations in the first class must intersect each initial quorum for operations in the second class. Informally, the direction of the arrow can be considered the

direction of information flow.

Figure 4-1 contains the quorum intersection graph for a file implemented by Gifford's replication method. One vertex corresponds to the class of *Read* events, and the other vertex corresponds to the class of *Write* events. The directed edge from the *Write* vertex to the *Read* vertex means that every final quorum for *Write* must intersect every initial quorum for *Read*, and the directed edge from the *Write* vertex to itself means that every pair of initial and final quorums for *Write* must intersect.

A quorum for an operation is said to be *minimal* if no smaller set of repositories is also a quorum for that operation. Henceforth, we restrict our attention to minimal quorums. Figure 4-2 displays the range of minimal quorum assignments for an object replicated among five identical repositories. An entry of the form (m,n) means that an initial quorum is any set of m repositories and a final quorum is any set of n repositories. Each column corresponds to a distinct minimal quorum assignment. For example, the first column corresponds to a quorum assignment in which an initial quorum for *Read* consists of any one repository and a final quorum for *Write* consists of all five repositories. (Given n identical repositories, Gifford's replication method permits $\lceil n/2 \rceil$ distinct quorum assignments.)

As discussed in [12], 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 chosen threshold value. Two quorums will intersect if the sum of their threshold values exceeds the sum of the votes assigned to all repositories. As an aside, we remark that some permissible quorum assignments cannot be characterized by voting schemes: consider a file replicated among four repositories, where *Read* quorums must contain either R1 and R2 or R3 and R4, and *Write* quorums must contain either R1 and R3 or R2 and R4.

Gifford's method illustrates three points that deserve emphasis. First, an object's fault-tolerance is best characterized by the (potentially different) levels of availability of each of its operations. Second, an operation's set of quorums determines its availability: an operation execution will succeed if and only if an appropriate quorum is available. Third, the constraints on quorum intersections determine the availability properties that can be

Figure 4-1: Quorum Intersection Graph for Files (Gifford's Method)

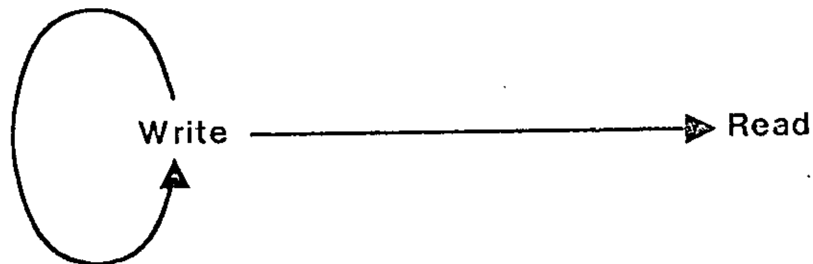

Figure 4-2: Quorum Assignments for Five Identical Repositories

Read	(1,0)	(2,0)	(3,0)
Write	(1,5)	(2,4)	(3,3)

realized by this method. For example, because each *Read* quorum must intersect each *Write* quorum, their levels of availability are inversely related: if the quorums for one event are made smaller (rendering it more available) then the quorums for the other event must be made correspondingly larger (rendering it less available). Similarly, because each pair of *Write* quorums must intersect, a *Write* quorum must encompass a majority of repositories, implying that *Write* cannot be more highly available than *Read*.

4.2. A Replicated FIFO Queue Implementation

A *FIFO Queue* has two operations: *Enq* places an item in the queue, and *Deq* removes the least recently enqueued item, raising an exception [20] if the queue is empty. A replicated queue might be used as a highly available spooler or scheduler. In this section we consider a naive implementation of a replicated FIFO queue where the queue's state is stored in a replicated file, and the *Enq* and *Deq* operations are implemented by *Read* and *Write* operations. The purpose of this example is to illustrate the benefits of an alternative replication method that takes advantage of type-specific properties of queues.

Let us consider the constraints on quorum assignment that result from implementing a FIFO queue on top of a replicated file. To dequeue an item, the front-end reads the versions from an initial *Read* quorum of repositories. If the queue is non-empty, the appropriate item is removed, and the shortened queue is written out to a *Write* quorum. Otherwise, an exception is returned to the client, but the repositories need not be updated. *Enq* is implemented similarly. For brevity, we henceforth restrict our discussion to *Deq* events that do not terminate with an exception.

The constraints on *Read* and *Write* quorums impose the following constraints on quorums for *Enq* and *Deq* events.

- Every initial *Enq* quorum must intersect every final *Enq* quorum.
- Every initial *Enq* quorum must intersect every final *Deq* quorum.
- Every initial *Deq* quorum must intersect every final *Enq* quorum.
- Every initial *Deq* quorum must intersect every final *Deq* quorum.

Figure 4-3 displays the required quorum intersections for queue events. Figure 4-4 illustrates the only quorum assignment for a queue replicated among five identical repositories. In fact, there is exactly one quorum assignment for *n* identical repositories, because both *Enq* and *Deq* quorums must encompass a majority of repositories.

These quorum assignments are highly constrained, and it is natural to ask whether these constraints can be relaxed. In the next section we introduce an alternative replication method that places fewer constraints on quorum intersections for replicated queues. Every quorum permitted by the file-based approach is permitted by the revised method, but the latter permits quorum assignments that the file-based method does not.

5. General Quorum Consensus

In this section we introduce a quorum consensus replication method that places fewer constraints on quorum intersection, permitting a wider range of availability properties to be realized. In the first part of this section, we propose a less restrictive file replication method in which timestamps are used in place of version numbers. In the next part, we propose a less restrictive queue replication method in which timestamped logs are used in place of

Figure 4-3: Quorum Intersection Graph for Queues (Gifford's Method)

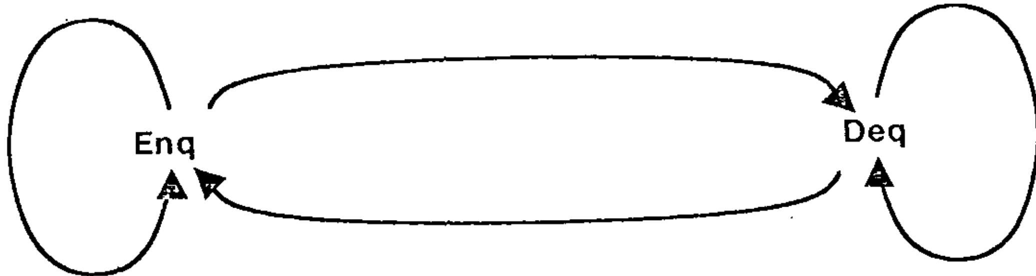

Figure 4-4: Quorum Assignments for Five Identical Repositories

Enq	(3,3)
Deq	(3,3)

versions. In the final part, we show how the revised replication methods for files and queues can be generalized to encompass objects of arbitrary type.

5.1. Timestamps vs. Version Numbers

In this section, we eliminate the constraint that *Write* quorums for files must intersect. This improvement is achieved by replacing the version numbering scheme with a scheme based on timestamps. In addition to reducing the constraints on quorum intersection, the timestamp-based scheme requires fewer messages. Gifford's thesis [13] includes the suggestion that timestamps could be used in place of version numbers; we have worked out the consequences of this suggestion in detail.

Timestamps are generated by system calls. Timestamps are totally ordered, having the following properties:

- The ordering of timestamps generated by a single action reflects the order in which they were generated.
- The ordering of timestamps generated by distinct committed actions reflects the order in which the actions are serialized.

These properties can be realized by structuring timestamps as follows:

- The high-order bits are occupied by an *action timestamp* defining the action's serialization ordering 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.

It remains to be shown how action timestamps are generated. We describe two 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 [26], each action is given a *pseudotime* on initialization, and scheduling constraints ensure that actions remain serializable in pseudotime order. Under this scheme, the action pseudotimes are precisely the action timestamps needed.

Under two-phase locking [10], 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, it reads a value from a Lamport clock [19]. This clock value is used as the action timestamp. Similar timestamp generation schemes have been proposed by Dubourdieu [9], by Chan et al. [7], and by Weihl [28].

The file replication method is changed as follows. Instead of reading and advancing the most recently generated version number, a front-end executing a *Write* generates a new timestamp and appends it to the newly created version. Instead of choosing the version with the latest version number, a front-end executing a *Read* chooses the version with the most recent timestamp.

The timestamp-based scheme imposes fewer constraints on quorum assignment. Because a front-end executing a *Write* need not observe any previous version numbers, *Write*

quorums are no longer required to intersect. The quorum intersection graph for a replicated file employing timestamps is shown in Figure 5-1. Figure 5-2 shows the possible quorum assignments for a file replicated among five identical repositories. The number of quorum assignments for n identical repositories has effectively doubled, going from $\lceil n/2 \rceil$ to n.

Figure 5-1: Quorum Intersection Graph for Files (Revised Method)

Figure 5-2: Quorum Assignments for Five Identical Repositories

Read	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)
Write	(0,5)	(0,4)	(0,3)	(0,2)	(0,1)

By assigning non-intersecting *Write* quorums, *Write* can be made more highly available than *Read*. Such quorum assignments facilitate "blind writes," in which a file is written without first being read. For example, if actions ***A*** and ***B*** write to disjoint final quorums, then a later action would choose the version with the later timestamp. A similar technique has been used for database synchronization, where it is known as the *Thomas Write Rule* [27].

Because files are typically read before they are written, we do not believe that quorum assignments facilitating blind writes are of significant practical interest for files. Nevertheless, when we turn our attention from files to other data types, we will see that there are interesting types having operations that resemble blind writes, such as the *Enq* operation for queues. The use of timestamps significantly improves replication methods for such types.

For files, a more practical advantage of the timestamp-based scheme is that *Write* invocations require half as many messages as in the version-numbering scheme, because the initial round of messages needed to ascertain the current version number is no longer

needed. The improved method is also of theoretical interest, as it shows that the requirement that *Write* quorums intersect is not really a fundamental constraint, but rather an artifact of a particular version numbering scheme.

5.2. Logs vs. Versions

Instead of replicating versions of the queue, consider the consequences of storing a *log* of events at each repository. A *log* is a sequence of *entries*, where an entry is a timestamped record of an event. Individual log entries need not appear at all repositories, and the log at any particular repository may have arbitrary gaps.

To enqueue an item *x*, the front-end creates a timestamped *Enq* entry naming the item to be enqueued: "1:00 Enq(x);Ok();Ok()." Here *1:00* is a timestamp, *Enq*(*x*) is the invocation, and *Ok*() is the response indicating a normal termination. This entry is appended to each log at a final *Enq* quorum of repositories.

When a front-end receives a *Deq* invocation, it executes the following steps:

- The front-end reads the logs from an initial *Deq* quorum of repositories. These entries are merged in timestamp order, discarding duplicates. The resulting log is called a *view*.

- The state of the queue is reconstructed from the view, and the item to be returned is ascertained.

- If the queue is non-empty, the front-end records the *Deq* event by appending a new entry to the view and sending the modified view to a final *Deq* quorum of repositories, where it is merged with the resident logs. An exception is returned if the queue is empty.

Once these steps have been successfully completed, the response (the dequeued item or an exception) is returned to the client.

It should be emphasized that this technique is intended to serve as a conceptual model for replication, not as a literal design for an implementation. An implementation could be considerably more efficient. For example, it is not really necessary to maintain a complete log of all queue events, nor is it necessary to write out the entire view in Step Three. In Section 7 we present a replicated queue implementation that includes measures to reduce storage consumption and message traffic. Nevertheless, we focus on the unoptimized

method for now because it can be more readily generalized to other data types.

What constraints must we impose on quorum intersections for queue operations? In Section 6 we present a systematic method for establishing constraints on quorum intersections, but for now we rely on informal arguments. If each initial quorum for an invocation is constrained to intersect each final quorum for an event, then all prior entries for that event will appear in any view constructed for the invocation. To choose a correct set of constraints on quorum intersection, we must determine how much of an object's complete log must be observed to choose a correct response to the invocation. 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. The requirements induce the following constraints on quorum intersection for *Enq* and *Deq* events:

- Every initial *Deq* quorum must intersect every final *Enq* quorum.
- Every initial *Deq* quorum must intersect every final *Deq* quorum.

By contrast, the view for an *Enq* invocation need not include any prior events, because *Enq* returns no information about the queue's state. As a consequence, initial *Enq* quorums are empty.

To illustrate this method, let us trace a short execution for a queue replicated among three repositories, where *Enq* has quorums of (0,2) and *Deq* has quorums of (2,2). The queue is initially empty. An item *x* is enqueued by appending a log entry with timestamp *1:00* to the 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 reads the logs from R2 and R3, merging the empty log from R3 with the single-entry log at R2. The resulting view indicates 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*. 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()	1:00 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()

At this point, the log at each repository defines a legal queue, but 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(x)	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)

Figure 5-3 shows the quorum intersection graph for the revised implementation of a replicated queue, and Figure 5-4 shows the quorum assignments for a queue replicated among five identical repositories. As one can see by comparing Figures 4-4 and 5-4, the log-based implementation places fewer constraints on quorum intersection than the file-based implementation, and hence permits a wider range of quorum assignments and a wider range of availability trade-offs. Given *n* identical repositories, there is only one quorum assignment for the file-based implementation (both *Enq* and *Deq* quorums encompass a majority of repositories) but there are $\lceil n/2 \rceil$ quorum assignments for the log-based implementation (only *Deq* quorums require a majority). In each of the new quorum assignments, the availability of the *Enq* operation is increased at the cost of decreasing the availability of the *Deq* operation. This trade-off could be useful in a highly available printer spooler, where the availability of the *Enq* operation used to spool items may be considered more important than the availability of the *Deq* operation used by the printer controller.

Our method has the disadvantage that logs (and messages) continue to grow. This problem can be alleviated by garbage collection: entries for events that can no longer affect later events can be discarded. For example, file versions can be viewed as optimized logs in which each *Write* entry is discarded as soon as it is superseded. Each repository can

Figure 5-3: Quorum Intersection Graph for Queues (Revised Method)

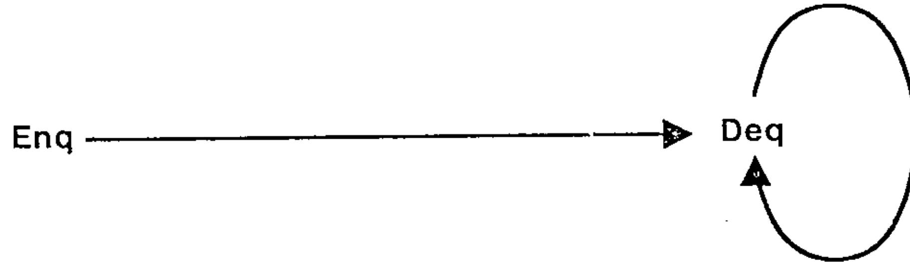

Figure 5-4: Quorum Assignments for Five Identical Repositories

Enq	(0,1)	(0,2)	(0,3)
Deq	(5,1)	(4,2)	(3,3)

garbage collect its own log, or collections of repositories can cooperate to identify and discard obsolete entries. Message sizes can also be reduced if front-ends maintain a cache of log entries, requesting only the missing entries from repositories. These optimizations are type-specific, in the sense that an optimization appropriate for queues is not necessarily appropriate for other data types. Examples of such optimizations are presented in Section 7.

5.3. Objects of Arbitrary Type

We are now ready to address the main point of this paper. In this section, we generalize the replication method described above for files and queues to objects of arbitrary type. The treatment is informal; a more precise description together together with correctness arguments is given in Section 6.

An object may be viewed as an automaton that accepts invocations and returns responses. As before, an invocation consists of an operation name and argument values, a response consists of a termination condition and result values, and an event is a paired invocation and response. A *history* is a sequence of events. Each object has a *specification* that characterizes a set of *legal* histories for that object. Objects having the same specification

are said to belong to the same *data type*.

Just as for queues, an object's history is represented by a collection of logs residing at repositories. Log entries are partially replicated; each log entry might be stored at several repositories, and each repository might store only a fragment of the entire log. Each invocation is assigned a set of initial quorums, and each event is assigned a set of final quorums. When a front-end receives an invocation from a client, the operation is executed in the following three steps.

- The front-end reads the logs from an initial quorum of repositories for that invocation. These logs are merged in timestamp order to construct a larger log called a *view*.

- The front-end chooses a response consistent with the object state as defined by the view.

- The front-end records the new invocation and response by appending an entry to the view, sending the modified view to a final quorum of repositories for that event. Each repository in the final quorum merges the view with its resident log.

The response may be returned to the client once all three steps have been completed successfully. These steps must be executed atomically; the enclosing action must be aborted if any step fails.

An instance of this replication method is said to be *correct* if it satisfies the following properties:

- Every view constructed in Step One is a legal history.

- Any response legal for the view is legal for the object as a whole.

The first property ensures that it is possible to choose a response in Step Two, and the second property ensures that the response chosen is indeed the "right" one. General quorum consensus actually satisfies a stronger property: the result of merging the logs from *any* collection of repositories is legal. As an incidental benefit, this property provides *catastrophe insurance*: if it becomes impossible to locate a quorum for an operation because some set of repositories has become permanently inaccessible, the remaining repositories still define a legal state that corresponds to a subhistory of the entire (lost) history. The object's surviving history can be rendered action-consistent if each action

records the same set of entries at each repository whose state it modifies.

Just as for files and queues, the correctness of the replication method is ensured by constraining certain pairs of initial and final quorums to have non-empty intersections. These constraints determine the level of availability that can be provided for each individual event, as well as the trade-offs among the levels of availability of various events. Given a type specification, we wish to derive constraints that are both correct and optimal. By *correct*, we mean that any quorum assignment consistent with the constraints yields a correct implementation. By *optimal*, we mean that there exists no smaller set of constraints that also yields only correct implementations. Note that both correctness and optimality are defined with respect to a particular replication method. For example, Gifford's original constraints on *Read* and *Write* quorums are correct and optimal for the method based on version numbers, but they are not optimal for the method based on timestamps. In the next section, we present a systematic way to derive correct and optimal constraints on quorum intersection for general quorum consensus directly from an object's type specification.

A naive implementation of this scheme has the practical disadvantage that logs and messages grow as new entries accumulate. As before, this problem can be alleviated by the application of type-specific log compaction techniques. In practice, a replicated object should be designed in two steps: quorums are chosen as for straightforward log-based scheme, and log compaction techniques are then applied to reduce storage consumption. In Section 7 we show some examples of log compaction techniques for particular types.

6. Correctness Arguments

This section presents correctness arguments for general quorum consensus. We first describe our model for specifications, objects, and logs, and then present a model for our replication method. We then define the notion of a *serial dependency* relation for a data type, and show that a replicated object based on general quorum consensus is correct if and only if the quorum intersection relation is a serial dependency relation for the object's type.

6.1. Specifications

A *specification* for an object is a tuple <INV, RES, LEGAL>, where INV is a set of *invocations*, RES is a set of *responses*, and LEGAL is a set of *legal histories* for that object. An *event* is a pair [*inv*;*res*], where *inv* is an invocation and *res* is a response.

EVENT = INV $\times$ RES

For example, "Enq(x)" is an invocation, "Ok()" is a response, and "[Enq(x);Ok()]" is an event of a *Queue* object.

A *history* is a finite sequence of events that corresponds to a computation. The histories that correspond to valid computations are called *legal histories*. For example, the following queue history:

Enq(x);Ok()
Deq();Ok(x)

is legal, but the history:

Enq(x);Ok()
[Deq();Empty()]

is not. We assume that every prefix of a legal history is itself a legal history. One way to characterize a set of legal histories is to use *algebraic specifications* [15]. In this paper, we will rely on informally stated specifications.

The specifications employed in this paper model *serial computations* in which steps occur one at a time. The absence of an explicit model of concurrency is justified by the assumption that replication is performed on top of an atomic action system. Our specifications model computations that have already been serialized by the underlying action system. In [17], show that a high level of concurrency can be achieved if the concurrency control method is integrated with the replication method.

6.2. Logs

Timestamps are chosen from a totally ordered domain TIMESTAMP. A *log* l is a map from some finite set of timestamps to events.

l: TIMESTAMP $\rightarrow$ EVENT

The notation $l(t) = \perp$ means that the log l has no event associated with the timestamp t. We

say that a log *contains* an event e if there exists a timestamp t such that $l(t) = e$.

A log m is a *sublog* of l if $m(t) = l(t)$ for every timestamp for which m is defined. Two logs l and m are *coherent* if they agree at every timestamp for which they are both defined. The *merge* operation $\cup$ is defined on pairs of coherent logs by:

$$(l \cup m)(t) = \text{if } l(t) \neq \bot \text{ then } l(t) \text{ else } m(t).$$

Because the merge operation is defined only for coherent logs, it is commutative and associative.

If l is a log and e an event, the notation $l = m \cdot e$ means that l can be expressed as the merger of the log m and a log containing the single event e, with the additional property that the latter's timestamp for e is greater than any timestamp in m. Informally, we say that l is constructed by appending e to m. The notation $l = l_1 \cdot l_2$ indicates that the log l can be expressed as the merger of two logs l_1 and l_2, where each timestamp for which l_2 is defined is greater than each timestamp for which l_1 is defined.

Let l be a log, and let $\{t_0,\ldots,t_n\}$ be the set of timestamps for which l is defined, indexed in ascending order. Let $H(l)$ be the history whose i-th event is $l(t_i)$ for $0 \leq i \leq n$. The log l is said to be *legal* if the history $H(l)$ is a legal history.

6.3. Serial Dependency

Let $\succ$ denote a relation between invocations and events. It is convenient to use the same symbol to denote the relation between events defined by:

$$[inv;res] \succ [inv';res'] \Leftrightarrow inv \succ [inv';res'].$$

Informally, a sublog m of l is said to be *closed* in l with respect to $\succ$ if whenever m contains an event e of l, it also contains every prior event e' of l such that $e \succ e'$. More precisely:

> **Definition 1:** The sublog m is *closed* in l if given timestamps t and t' and events e and e' such that $t > t'$, $e \succ e'$, $l(t) = e$, $l(t') = e'$, and $m(t) = e$, then $m(t') = e'$.

We omit mention of $\succ$ and l when they are clear from context.

The correctness condition for general quorum consensus is based on the notion of a *serial dependency* relation between invocations and events.

Definition 2: The relation $\succ$ is a serial dependency relation if for all invocations *inv*, all responses *res*, all legal logs l, and all closed sublogs m of l containing all events *e* of l such that *inv* $\succ$ *e*:

$$m \cdot [inv;res] \text{ is legal} \Rightarrow l \cdot [inv;res] \text{ is legal.}$$

Informally, this definition states that a correct response to an invocation can be chosen by observing any closed legal sublog that contains all the events on which the invocation depends.

Perhaps the simplest example of a serial dependency relation is provided by the *File* type, for which *Read* invocations depend on *Write* events, but *Write* invocations need not depend on any prior events. If l is a log consisting of *Read* and *Write* events, then a correct response to *Read*() can be chosen from any closed sublog of l that contains all *Write* events, and the correct response to *Write*(*x*) (i.e. *Ok()*) can be chosen from any closed sublog, including the empty log. Similarly, *Deq* invocations depend on prior *Enq* events, and on prior *Deq* events that did not signal *Empty*. *Enq* invocations do not depend on any prior events, because *Enq*, like *Write*, has only one possible response.

For deterministic types, in which each invocation has a single correct response, the implication in Equation 2 goes in both directions, but for non-deterministic types, the implication goes in one direction only: there may be legal responses to *inv* in l that are not evident from the sublog m. An example of such a non-deterministic type is given in the *SemiTable* example in Section 7.4.

A given type may have several serial dependency relations. For example, the relation in which every invocation depends on every event is a serial dependency relation, although not an interesting one. More interesting serial dependency relations are the *minimal* relations, relations with the property that no smaller relation is a serial dependency relation. We remark that a type need not have a unique minimal serial dependency relation. An example of a data type with several distinct minimal relations is given in the *DoubleBuffer* example in Section 7.5.

We will see that serial dependency relations completely characterize the constraints on quorum intersections, in the sense that a replicated object will satisfy its specification if and

only if its quorum intersection relation is a serial dependency relation.

6.4. Replication Schemes

A replication scheme is a tuple ⟨S, REPOS, *Initial*, *Final*⟩, S is a specification and REPOS is a set of repositories. We define the domain QUORUM to be the domain of sets of repositories:

$$\text{QUORUM} = 2^{\text{REPOS}}$$

Initial is a map that defines the correspondence between each invocation and its set of initial quorums:

$$\mathit{Initial}: \text{INV} \rightarrow 2^{\text{QUORUM}}.$$

Final is a map that defines the correspondence between each event and its set of final quorums:

$$\mathit{Final}: \text{EVENT} \rightarrow 2^{\text{QUORUM}}.$$

The quorum assignments define a quorum intersection relation $\succ$ between invocations and events:

> $inv \succ [inv';res'] \Leftrightarrow$ each initial quorum for the invocation *inv* intersects each final quorum for the event $[inv';res']$.

A *replicated object* is a non-deterministic automaton characterized by a replication scheme. The object's state is given by a collection of logs $\{l_d \mid d \in \text{REPOS}\}$. Each log is initially empty. Given an invocation *inv*, the object undergoes a state transition and returns a response *res* as follows:

- Choose an initial quorum I from *Initial*(*inv*). Let v be the log constructed by merging the l_d for all d in I. We refer to v as the *view*.

- Choose a timestamp greater than any chosen in a previous step, and a result *res* such that $v \cdot [inv;res]$ is legal.

- Choose a final quorum F from *Final*(*inv*;*res*), and merge $v \cdot [inv;res]$ with the log at each d in F.

- Output the response *res*.

In summary, after an object whose state is $\{l_d \mid d \in \text{REPOS}\}$ inputs an invocation *inv*, it outputs a response *res*, and enters a new state given by $\{l_d' \mid d \in \text{REPOS}\}$, where:

$$l_d' = \text{if } d \in \text{F then } l_d \cup v \cdot [inv;res] \text{ else } l_d$$

A computation involving a replicated object can be characterized by a *global log* containing each event that has occurred since the object's creation. Note that an object's global log is not necessarily the log that would be constructed by merging the logs at all repositories at the end of the computation, because events having empty final quorums (such as *Read* events) are contained in the global log but not in the log at any repository.

6.5. Correctness Results

The desired correctness property for general quorum consensus states that every global log generated by a replicated object is legal. We will show that general quorum consensus is correct if and only if the quorum intersection relation is a serial dependency relation. We argue by induction, demonstrating the invariance of certain properties relating an object's global log to the logs kept at the repositories. We begin with a few simple lemmas.

Lemma 3: The result of merging closed sublogs is a closed sublog.

Proof: Let m and n be closed sublogs of l, let $(m \cup n)(t) = e$ and $l(t') = e'$, such that $t > t'$ and $e \succ e'$. Without loss of generality, assume that $m(t) = e$. Because m is closed in l, $m(t') = e'$, and therefore $(m \cup n)(t') = e'$.

Lemma 4: If m is a closed sublog of l, then m is a closed sublog of $l \bullet e$.

Proof: Left to the reader.

Lemma 5: If l is a log, and v is the view constructed for an event e (Steps 1 and 2 above), then v contains all events e' of l such that $e \succ e'$.

Proof: The view constructed for e is the result of merging all logs from an initial quorum for e. If $e \succ e'$, then there exists at least one repository that lies in both the initial quorum for e and the final quorum for e', so e' must appear in the view for e.

In the next two theorems, we establish certain relations between an object's global log and its internal state. Each property clearly holds in the initial state when all logs are empty; we show that the property is preserved by state transitions.

Theorem 6: The result of merging the logs from any set of repositories is a closed sublog of the global log.

Proof: It suffices to show that the log residing at any single repository is closed; the more general result follows from Lemma 3. Let the global log be l, and let the states of the repositories be given by $\{l_d \mid d \in \text{REPOS}\}$. Now assume that the object undergoes a transition from l to $l' = l \bullet e$, and let the new states of the repositories be given by $\{l_d' \mid d \in \text{REPOS}\}$. We show that if the l_d are closed in l,

then the l_d' are closed in l'.

Let d be a repository. If d is not a member of the final quorum for e, then $l_d' = l_d$, so the result follows immediately from Lemma 4. Otherwise,

$$l_d' = l_d \cup \nu \bullet e$$

where ν is the view used to compute e. The view ν is the result of merging logs from repositories, each of which is closed in l (induction hypothesis). It follows that ν is closed in l (Lemma 3) and in l' (Lemma 4). The log $\nu \bullet e$ is closed in l' because ν contains every event on which e depends (Lemma 5). It follows that l_d' is closed because it is the result of merging the closed logs $\nu \bullet e$ and l_d (Lemma 3).

It follows that the result of merging logs from an arbitrary set of repositories is closed, and thus every view constructed for an invocation is closed.

Theorem 7: If the quorum intersection relation is a serial dependency relation, then the result of merging logs from any collection of repositories is legal.

Proof: Let S be an arbitrary set of repositories, and let l_S be the result of merging the logs from the repositories in S. let l_S' be the result of merging the logs from S following a transition from l to $l' = l \bullet e$. We show that if l_S is legal, so is l_S'.

There are two cases to consider: either S intersects the final quorum for the new event e, or it does not. If it does not, then $l_S = l_S'$, and the result is immediate. Otherwise, the new event and its view have been merged with the log of one or more repositories in S, yielding:

$$l_S' = l_S \cup \nu \bullet e$$

where ν is the view used to compute e. Both ν and $(\nu \cup l_S)$ are closed by Theorem 6 and legal by the induction hypothesis. Because ν is closed in l, it is also closed in the sublog $(l_S \cup \nu)$ of l, and it contains every event on which e depends (Lemma 5). By Definition 2, the legality of $\nu \bullet e$ implies the legality of $(l_S \cup \nu) \bullet e$. But $(l_S \cup \nu) \bullet e$ is the same as $l_S \cup \nu \bullet e = l_S'$.

The previous theorem has two corollaries of interest. The first is a well-formedness property: every view corresponds to a legal log. The second is the catastrophe insurance property: the view constructed by merging the logs from any collection of repositories is a legal sublog of the object's global log.

We are now ready to address the basic correctness property for general quorum consensus.

Theorem 8: If the quorum intersection relation is a serial dependency relation, then every transition permitted by quorum consensus carries a legal global log to

a legal global log.

Proof: When an object undergoes a transition from a legal log l to $l \cdot e$, the view v is closed in l (Theorem 6), legal (Theorem 7), and contains every event of l on which e depends (Lemma 5). We know that $v \cdot e$ is legal by construction; thus from Definition 2 we deduce that $l \cdot e$ is legal.

We have just shown that general quorum consensus is correct if the quorum intersection relation is a serial dependency relation. We now show the converse: any quorum intersection relation that is not a serial dependency relation permits quorum assignments that do not guarantee correctness.

Theorem 9: If the quorum intersection relation is not a serial dependency relation, then there exist quorum assignments that generate illegal global logs.

Proof: We assume that the quorum intersection relation does not satisfy Definition 2, and we construct a scenario in which an illegal global log is generated. If the quorum intersection relation is not a serial dependency relation, then there exists an event e = [*inv*;*res*], a log l, and a closed sublog m containing all events whose final quorums intersect the initial quorums for *inv*, with the property that:

$m \cdot e$ is legal but $l \cdot e$ is illegal.

To prove the theorem, it suffices to assign quorums in such a way that invoking *inv* when the global log is l causes m to be constructed as the view, resulting in the illegal global log $l \cdot e$.

The logs are replicated at two repositories: R1 and R2. Each event in m chooses an initial quorum of R1 and a final quorum of both R1 and R2. Each event in l but not in m chooses an initial quorum of R1 and R2, and a final quorum of R2. As a result of these choices, the view used to generate each event in m contains all and only the prior events in m, and the view used to generate every other event contains all prior events.

We claim that these quorum choices must satisfy the object's quorum intersection relation. All initial and final quorums intersect except the final quorums for events not in m and the initial quorums for events in m. If any of these quorums were required to intersect, then m would not be closed, contradicting the assumption. When a front-end assembles a view for *inv*, it reads the log from R1, which is exactly the sublog m. R1 is an initial quorum for *inv* because it intersects the final quorums for every event in m. The front-end then chooses the response *res*, which by assumption is legal for m, but illegal for l.

Theorem 9 is an optimality result. It shows that a minimal set of constraints on quorum

intersection corresponds exactly to a minimal serial dependency relation for the type.

7. Examples

This section presents some examples of replicated objects. Our derivation of serial dependency relations in these examples is informal. A formal derivation would require introducing a formal specification method with proof rules strong enough to show the properties discussed in the previous section, a task beyond the scope of this paper. These examples also illustrate some type-specific optimizations for reducing the sizes of logs and messages.

7.1. Files

Our previous discussions have employed a somewhat idealized model for files, in which the *Write* operation completely overwrites the contents of the file. A more realistic model would permit partial updates, in which the *Write* operation modifies only part of the file. One possibility is to view a file as an extensible array of pages. The file is initially empty, and pages may be appended to the end of the file. Individual pages may then be read or written.

The *PagedFile* type provides the following operations:

    WritePage = operation(int, string) signals (bounds)

overwrites the page with the given index, and

    ReadPage = operation(int) returns(string) signals (bounds)

returns the current value of the page with a given index. Both *ReadPage* and *WritePage* signal an exception if the index lies beyond the end of the file. The *Append* operation allows new pages to be added to the end of the file:

    Append = operation(string)

and the *Size* operation returns the number of pages currently in the file.

    Size = operation() returns (int)

The view for a *ReadPage* invocation must include the previous *Append* events, to detect whether the page exists, as well as all *WritePage* events for that page, to determine the page's current value. Consequently, a *ReadPage* invocation depends on *Append* events and on *WritePage* events for that same page. The view for a *WritePage* invocation need include only previous *Append* events (to determine whether the index is in bounds).

Consequently, a *WritePage* invocation depends only on *Append* events. *Size* invocations depend only on *Append* events because *Append* is the only operation that affects the size of the table.

Figure 7-1 illustrates some of the required quorum intersections for paged files (to avoid cluttering the diagram, we have omitted mention of events that terminate with exceptions). The vertices marked *WritePage*(i,*) and *ReadPage*(i) respectively denote the class of *WritePage* and *ReadPage* events for the page with index i. Figure 7-2 shows the range of quorum assignments for a paged file replicated among five repositories based on the assumption that the quorums for *ReadPage* and *WritePage* do not depend on the page affected.

Instead of keeping logs at repositories, an efficient implementation of the *PagedFile* type might keep a timestamped version of each page. *ReadPage* invocations would request only the most recent version of the page being read.

Instead of reading and writing at the page level, one might allow arbitrary sections of a file to be read or written. Although such a file could be represented as a log of updates, just as the queue was represented by a log of *Enq* and *Deq* events, it might be more efficient to discard entries for superseded updates, effectively associating timestamps with variably-sized regions of the file.

7.2. Queues

A log representing a queue can be compacted by taking advantage of the observation that an item must have been dequeued if an item enqueued with an earlier timestamp has been dequeued. A repository that is part of a replicated queue implementation need retain only the following information:

- The *Enq* timestamp of the most recently dequeued item. This timestamp is called the repository's *horizon* timestamp.

- The *Enq* entries whose timestamps are later than the horizon timestamp.

Repositories do not store *Deq* entries or *Enq* entries for items known to be dequeued.

An item is dequeued as follows. The front-end reads the horizon timestamps and the *Enq*

Figure 7-1: Quorum Intersection Graph for Paged Files

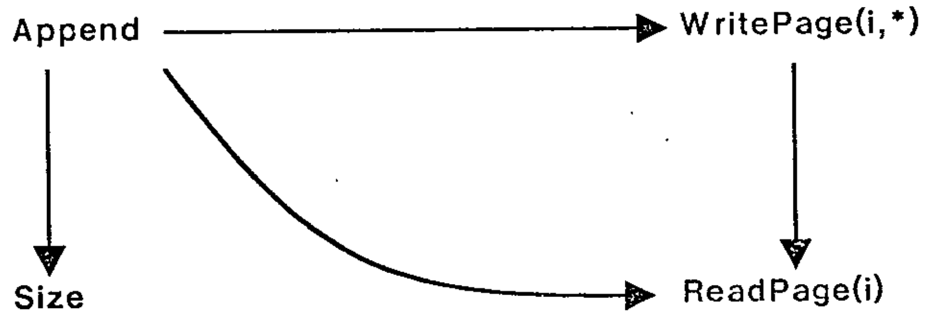

Figure 7-2: Quorum Assignments for Five Identical Repositories

Append	(0,5)	(0,5)	(0,5)	(0,5)	(0,5)	(0,4)
Size	(1,0)	(1,0)	(1,0)	(1,0)	(1,0)	(2,0)
ReadPage	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(2,0)
WritePage	(1,5)	(1,4)	(1,3)	(1,2)	(1,1)	(2,4)

Append	(0,4)	(0,4)	(0,3)	(0,2)	(0,1)
Size	(2,0)	(2,0)	(3,0)	(4,0)	(5,0)
ReadPage	(3,0)	(4,0)	(3,0)	(4,0)	(5,0)
WritePage	(2,3)	(2,2)	(3,3)	(4,2)	(5,1)

entries from an initial quorum. The *Enq* entries are merged in timestamp order, and all entries earlier than the latest observed horizon time are discarded. The oldest remaining *Enq* entry identifies the item to be dequeued, and thus the new horizon time. The operation is complete when the front-end installs the new horizon time at a final *Deq* quorum.

To illustrate the technique, we re-examine the three-site replicated queue example described above in Section 5.2. As before, we start with a queue where x has been enqueued at two sites.

R1	R2	R3
horizon: 0	horizon: 0	horizon: 0
1:00 Enq(x);Ok()	1:00 Enq(x);Ok()	

When *x* is dequeued, the horizon time for R2 and R3 is set to 1:00, and R2 discards the *Enq* entry for *x*.

R1	R2	R3
horizon: 0	horizon: 1:00	horizon: 1:00
1:00 Enq(x);Ok()		

Item *y* is enqueued at R1 and R2, and *z* is enqueued at R1 and R3.

R1	R2	R3
horizon: 0	horizon: 1:00	horizon: 1:00
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()

When *y* is dequeued, the horizon time is set forward to 1:15 at R1 and R3, and R1 discards the *Enq* entries for *x* and *y*.

R1	R2	R3
horizon: 1:15	horizon: 1:00	horizon: 1:15
	1:15 Enq(y);Ok()	
1:30 Enq(z);Ok()		1:30 Enq(z);Ok()

As a complementary compaction technique, repositories can periodically broadcast their horizon times. A repository that observes a later horizon time can advance its own, discarding all *Enq* entries with earlier timestamps. Note that propagating *Enq* entries in the same way does not aid log compaction, although it might provide better catastrophe insurance.

Message sizes for *Deq* can be further reduced if the first round of messages is replaced by the following two-round protocol, which is similar to a protocol proposed in [6] for replicated directories. In the first round, the front-end reads the horizon times from an initial quorum. In the second round, the front-end sends the latest horizon time back to the initial quorum, and each site responds with its oldest *Enq* entry whose timestamp is later than the horizon time. The earliest of these entries is the item to be dequeued. Although this protocol requires an extra high-level message, each message has a fixed size, so fewer packets will

be transmitted at the lower level.

7.3. Tables

The *Table* type stores pairs of values, where one value (the *key*) is used to retrieve the other (the *item*). The operation:

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

inserts a new key/item pair in the table. An exception is signaled if a pair with the given key is already present in the table. The operation:

Delete = Operation(k: Key) signals (Absent)

deletes the pair with the given key from the table. An exception is signaled if a pair with the given key is not present in the table. The operation:

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

alters the item bound to the given key. An exception is signaled if a pair with the given key is not present in the table. The operation:

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

returns the item bound to the given key. An exception is signaled if a pair with the given key is not present in the table. The operation:

Size = Operation() returns(int)

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

Because *Insert* signals if the key is present in the table, *Insert* invocations depend on prior *Insert* and *Delete* events. *Delete* invocations also depend on prior *Insert* events and prior *Delete* events. Because *Change* signals if the key is absent from the table, *Change* invocations depend on prior *Insert* events and prior *Delete* events, but not on prior *Change* events. *Lookup* depends on prior *Insert*, *Delete*, and *Change* events for its result, and *Size* depends only on prior *Insert* and *Delete* events, because *Change* and *Lookup* do not alter the number of keys in the table.

The quorum sets for *Insert*, *Delete*, *Change*, and *Lookup* could depend on the key. For example, operation executions using East Coast employees as keys could have different quorums than operation executions involving West Coast employees. Of course, the front-

ends must be able to tell to which class a key belongs.

Part of the quorum intersection graph for the Table type is shown in Figure 7-3. (To avoid cluttering the diagram, events that terminate with exceptions are not shown.) The vertex marked *Insert*(k,*)/*Delete*(k) represents the class of *Insert* and *Delete* events for the key value *k*. The edge from *Insert*(*k*,*)/*Delete*(k) to *Lookup*(k) means that *Lookup* invocations depend on prior *Insert* and *Delete* events for the same key value. The range of quorum assignments for five identical repositories is shown in Figure 7-4 based on the assumption that quorums are independent of key values.

Because operations involving distinct keys do not affect one another, the relative ordering of their entries is unimportant. To avoid an inefficient linear search, the log representing a table at a repository can itself be represented as a table mapping each key to the sequence of entries involving that key. The size of the table can be further reduced by the observation that only the most recent entry for a particular key can affect future events; thus it suffices to bind each key to its most recent (committed) entry. Any key that does not appear explicitly in the repository's table is implicitly bound to a *delete* entry with a timestamp of zero, which is older than any timestamp generated by a normal action.

Message sizes can be reduced if a front-end executing an operation requests only the entry for a particular key. If each front-end maintains a cache, it can include the timestamp for the key's cached entry with each invocation. Each repository in the initial quorum either returns an entry with a later timestamp, or it returns a confirmation that the cached entry is up to date. If the timestamp sent by the front-end is later than the repository's timestamp, then the repository's entry is out of date and may be discarded. Obsolete entries can also be detected if background tasks at repositories periodically broadcast the timestamp associated with a particular key. Once a repository has discarded an entry for a key, it responds to later invocations with a *Delete* entry with a zero timestamp.

A possible disadvantage of the replicated table representation is that entries for *Delete* events may tend to accumulate. An up-to-date *Delete* entry cannot be discarded as long as there is a possibility that another repository has an obsolete *Insert* entry for that key. If a repository unilaterally discards an entry for a deleted key, a later *Lookup* invocation for that

Figure 7-3: Quorum Intersection Graph for Tables

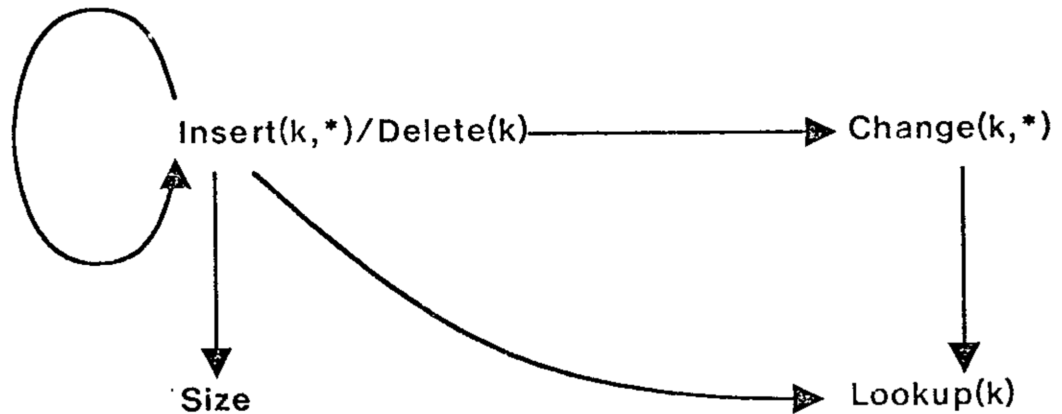

Figure 7-4: Quorum Assignments for Five Identical Repositories

Insert/Delete	(1,5)	(1,5)	(1,5)	(1,5)	(1,5)
Size	(1,0)	(1,0)	(1,0)	(1,0)	(1,0)
Lookup	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)
Change	(1,5)	(1,4)	(1,3)	(1,2)	(1,1)

Insert/Delete	(2,4)	(2,4)	(2,4)	(3,3)
Size	(2,0)	(2,0)	(2,0)	(3,0)
Lookup	(2,0)	(3,0)	(4,0)	(3,0)
Change	(2,4)	(2,3)	(2,2)	(3,3)

key may observe an earlier *insert* entry without realizing that it is out of date. Consequently, any compaction technique that can discard up-to-date *Delete* entries must be distributed, because it must preserve an invariant affecting the state of all repositories. One approach to this problem is a technique proposed by Bloch, Daniels, and Spector [8, 6] in which the table representation is rendered more compact by ordering the entries and merging adjacent entries for deleted bindings. In this scheme, Deletion requires two exchanges of fixed-size messages.

A complementary approach is to use long-running low-priority background actions to remove older entries from all repositories. A background task can be used to detect and discard obsolete *Insert* entries by propagating each key's current timestamp. If the background task also keeps track of the repositories it has notified, it can detect when all obsolete *Insert* entries for a deleted key have been discarded, and it can then make a second pass discarding the up-to-date *Delete* entries.

7.4. A Non-Deterministic Table

The specification for the *Table* type implies that the availability of the *Lookup*(k) and *Insert*(k,i) events are inversely related: one event can be made more likely to succeed only at the expense of the other. For example, if a key/item pair is replicated among n repositories, and if the initial quorum for *Lookup* consists of a single node, then the final quorum for *Insert* must consist of all n repositories. In proposals for replicated tables found in the Grapevine name server [5], in the Clearinghouse name server [24], and in a more recent proposal by Fischer and Michael [11], such a trade-off is considered to be unacceptable. Instead, a key/item pair is added or deleted at a single repository, and the update is later propagated to the other repositories. These approaches have the advantage that updates complete more quickly and more likely to succeed, but they have the disadvantage that the behavior of the server becomes considerably more complex because clients may observe transient "inconsistencies" in the table, and concurrent conflicting updates must somehow be resolved.

There are two ways to view these "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. Although the first view seems to reflect the attitude of the designers of the replication methods under discussion, we prefer the second view for its economy of mechanism: we may still use serializability to characterize the properties upon which the programmer may rely, and we may use general quorum consensus to implement a replicated non-deterministic table having essentially the same properties.

The *SemiTable* is a map from keys to multisets of items. When the table is created, it contains (conceptually, at least) a binding between every key and a multiset containing only

the distinguished item *Absent*. An invocation of *Insert*(k,i) adds the item i to the multiset associated with k, and an invocation of *Lookup*(k) will return some item previously bound to k, or signal *Absent*. There is an additional probabilistic guarantee that any item returned is "probably" the most recently inserted one.

The minimal serial dependency relation induced by the *SemiTable* specification is degenerate: no event depends on any other event, and thus no quorums are required to intersect. The view constructed for *Lookup*(k) will include the most recent *Insert* or *Change* event for that key if the initial quorum for *Lookup* happens to intersect the final quorum for the *Insert* or *Change* event. The *probability* that *Lookup* will observe an item is thus the probability that the quorums will intersect. That probability is unity for the *Table* 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 *SemiTable*, 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.

7.5. The DoubleBuffer Type

Files, Queues, Tables, and SemiTables each have a unique minimal serial dependency relation. We now consider a data type that has two distinct minimal serial dependency relations, neither of which is a subset of the other. 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 an 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. The alternatives arise because *Produce* events affect later *Consume* events 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.

In the first relation, illustrated in Figure 7-5, *Consume* invocations depend on both *Transfer* and *Produce* events, an organization analogous to a mail program in which a request to deliver a message simply marks the message for later transmission, and data is not actually moved until it is requested by the recipient.

In the second relation, illustrated in Figure 7-7, *Consume* invocations depend on *Transfer* events, and *Transfer* invocations depend on *Produce* events, an organization analogous to a mail program in which a request to deliver a message results in immediate transmission.

It is interesting to note that neither serial dependency relation is strictly better than the other with respect to quorum size. For example, comparing the right-hand columns of Figures 7-6 and 7-8, we see that the first relation has a minimal quorum assignment in which *Consume*, *Transfer*, and *Produce* quorums respectively consist of any five, one, and one repositories, while the same quorums for the second relation consist of any five, five, and one repositories. In this instance, the first relation is clearly preferable. Comparing the left-hand columns, however, we see that the first relation has a minimal quorum assignment in which *Consume*, *Transfer*, and *Produce* quorums respectively consist of any one, five, and five repositories, while the same quorums for the second relation consist of any one, five, and one repositories. In this instance, the second relation is clearly preferable.

The *DoubleBuffer* type illustrates why it does not suffice simply to write out the new entry to a final quorum at the conclusion of each operation execution. Consider the relation in which *Consume* depends on *Transfer*, and *Transfer* depends on *Produce*, but *Consume* does not depend directly on *Produce*. Any view constructed for *Transfer* contains the most

Figure 7-5: First Serial Dependency Relation for DoubleBuffer

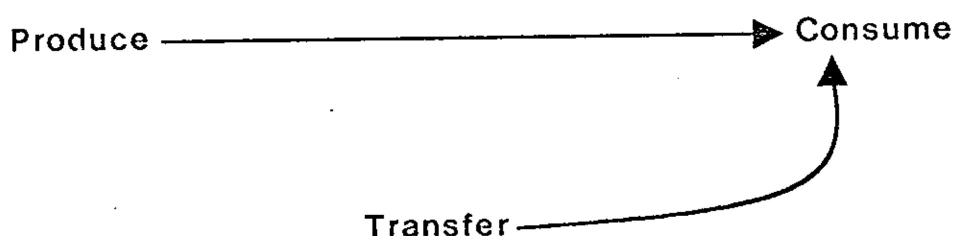

Figure 7-6: Quorum Assignments for Five Identical Repositories

Consume	(5,0)	(4,0)	(3,0)	(2,0)	(1,0)
Transfer	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
Produce	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)

Figure 7-7: Second Serial Dependency Relation for DoubleBuffer

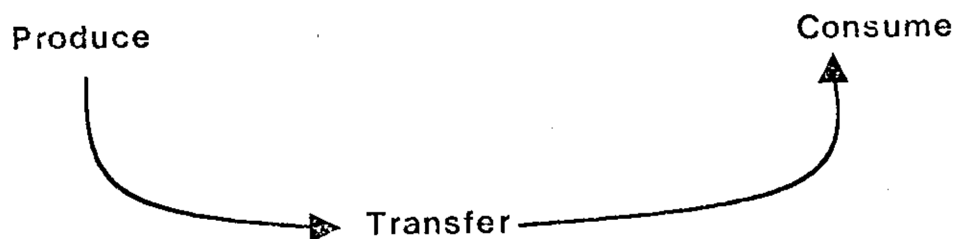

Figure 7-8: Quorum Assignments for Five Identical Repositories

Consume	(5,0)	(4,0)	(3,0)	(2,0)	(1,0)
Transfer	(5,1)	(4,2)	(3,3)	(4,4)	(5,5)
Produce	(0,1)	(0,2)	(0,3)	(0,2)	(0,1)

recent *Produce* entry, and any view constructed for *Consume* contains the most recent *Transfer*. Because a front-end executing a *Transfer* records both the *Transfer* entry and its view at a final quorum, the view for a later *Consume* invocation will also include the *Produce* entry needed to ascertain the item to be returned. If our replication method were to write out only the new *Consume* entry, then *Consume* would be forced to depend directly on *Produce*, unnecessarily restricting the range of quorum assignments.

8. 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 described here is that reconfiguration 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 [12]. 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 component objects, called the resource state and the configuration state. The *resource state* contains the information of interest to clients, while the *configuration state* contains the quorum information for the resource state. For example, a reconfigurable queue's resource state identifies the items in the queue, while its configuration state identifies the quorums for the *Enq* and *Deq* operations. Both the configuration and resource states are themselves replicated. The configuration state provides *GetQuorum* and *SetQuorum* operations.

An object is reconfigured in the following steps:

- Construct a view of the old resource state by merging the logs from a set of repositories that includes an old initial quorum for each operation.

- Initialize the new resource state by writing out the view at a set of repositories that constitutes a new final quorum for each operation.

- Record the new resource state quorums at a *SetQuorum* quorum for the new configuration state.

- Mark the old configuration state as *obsolete* and record the new configuration state's quorums at a *SetQuorum* quorum for the old configuration state.

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 obsolete configuration states before locating the current resource state.

The number of messages needed for this protocol can be kept to a minimum by a sensible choice of quorums. Quorums should be assigned so that 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 containing an obsolete cache timestamp, 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 information 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. If the old configuration states are discarded before that front-end's cache is brought up to date, then that front-end will be unable to locate the new resource state. 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.

The state of an object of type *RefCount* is given by 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 object's current value:

Value = operation() returns(int)

The quorum intersection graph for the *RefCount* type is shown in Figure 8-1. together with the range of quorum assignments for five identical repositories. *Value* invocations depend on both *Inc* and *Dec* events, but *Inc* and *Dec* invocations do not depend on any prior events because neither returns any information.

Each object's configuration state is modified to include a *RefCount* component. When a front-end first records the configuration state in its cache, it records *Inc* entries at a quorum of the configuration state's repositories. When a front-end observes that the configuration state has been rendered obsolete, it records *Dec* entries at a quorum of repositories. Once a configuration state has been rendered obsolete, the object's maintainers may merge the *Inc* and *Dec* entries from a quorum for the *Value* operation to ascertain the number of 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

Figure 8-1: Quorum Intersection Graph for Reference Counter

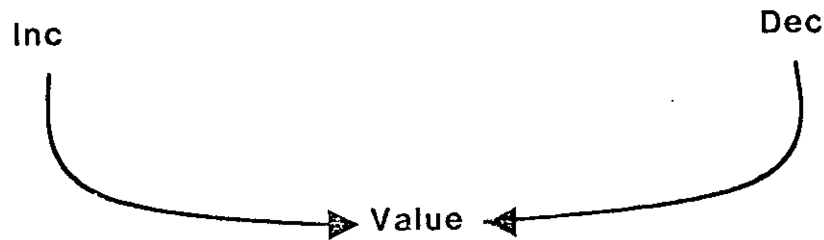

Figure 8-2: Quorum Assignments for Five Identical Repositories

Inc	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
Dec	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
Value	(5,0)	(4,0)	(3,0)	(2,0)	(1,0)

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 Current	1:00 Current	1:00 Current
{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 is marked as *current* with 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. No single repository has a record of all the increments that have occurred. 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.
- Mark the old configuration state as *obsolete* and 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 Current	2:00 Obsolete	2:00 Obsolete
{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 Current	2:15 Current	2:15 Current
{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 attempts to enqueue or dequeue an item using the quorums in its out-of-date cache. 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 Current	2:00 Obsolete	2:00 Obsolete
{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 Current	2:15 Current	2:15 Current
{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()		

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.

9. Discussion

This paper has introduced *general quorum consensus*: a new method for representing and managing replicated data. The model of computation employed admits two kinds of failures: communication interruptions and site crashes. Both categories of failures can be detected, but not necessarily distinguished from one another. Underlying support for atomic actions is needed to preserve invariant properties of the replicated data.

General quorum consensus is type-specific. Associated with each operation of the data type is a set of quorums, which are collections of sites whose logs must be observed or updated to execute the operation. An analysis of the data type's algebraic structure is used to derive a set of constraints on quorum intersections. These constraints are correct and optimal; any quorum assignment that satisfies these constraints yields a correct implementation, and it is shown that no weaker set of constraints guarantees correctness. Such constraints can be established for arbitrary types, including types having non-deterministic operations.

Constraints on quorum intersection define the range of realizable availability properties. For example, if quorums for two operations are required to intersect, then the operations' levels of availability must be inversely related. To characterize these constraints, we introduce the notion of a *serial dependency* relation for a data type. The problem of characterizing a data type's range of realizable availability properties is transformed into the algebraic problem of identifying its set of minimal serial dependency relations.

Our method employs two technical innovations: the use of timestamps in place of version numbers, and the use of logs in place of versions. Gifford's original scheme uses quorum intersection for two distinct purposes: to compute responses to invocations (*Read/Write* intersection), and to order events (*Write/Write* intersection). The introduction of timestamps eliminates the need to use quorum intersection to order events, permitting operations that do not commute to be given non-intersecting quorums. The use of logs in place of versions reduces the constraints on quorum assignment for operations that alter the object's state. A difficulty with versions is that any operation that modifies the object's state must locate and modify a current version, because versions that have diverged cannot be reconciled. By contrast, divergent logs can be reconciled simply by merging their entries in timestamp

order.

As a practical matter, a log compaction mechanism is needed to prevent log and message sizes from growing without bound. Effective log compaction techniques are type-specific; as illustrated by the examples in Section 7, there is typically a range of techniques available for any given type, ranging from obvious to subtle, and from cheap to expensive. As an aside, we remark that an important step in the development of general quorum consensus was the realization that quorum assignment and storage compaction should be treated as distinct issues.

General quorum consensus supports on-the-fly reconfiguration. A novel replicated reference counting scheme is introduced to facilitate moving an object from one set of sites to another. Reconfiguration incurs a cost in reduced availability and increased message traffic only when it actually occurs.

The major contribution of this paper is the presentation of a systematic and general technique for exploiting type-specific properties of data to achieve more effective replication. The traditional approach to replication has been to treat the data as an uninterpreted file whose contents may only be read or written. While this approach is capable of representing data of arbitrary type, our type-specific method can realize a wider range of availability properties and more flexible reconfiguration.

Acknowledgments

I would like to thank Joshua Bloch and Dean Daniels for their careful reading 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., 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.

[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] Daniels, D., and Spector, A.
An Algorithm for Replicated Directories.
In *Proceedings of the 2nd Annual Symposium on Principles of Distributed Computing*. August, 1983.

[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] 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] 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.

[12] Gifford, D. K.
Weighted Voting for Replicated Data.
In *Proceedings of the Seventh Symposium on Operating Systems Principles.* ACM SIGOPS, December, 1979.

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

[14] 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.

[15] Guttag, J. V., and Horning, J. J.
The Algebraic Specification of Abstract Data Types.
Acta Informatica (10):27-52, 1978.

[16] Hammer, M. M., and Shipman D. W.
Reliability Mechanisms in SDD-1, a System for Distributed Databases.
ACM Transactions on Database Systems 5(4):431-466, dec, 1980.

[17] 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.

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

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

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

[21] 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.

[22] B. Liskov, M. Herlihy, P. R. Johnson, G. Leavens, R. Scheifler, and W. Weihl.
Preliminary argus reference manual.
Technical Report 39, M.I.T. L.C.S. Programming Methodology Group, October, 1983.

[23] 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.

[24] 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.

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

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

[27] 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.

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