Replication Methods for Abstract Data Types

by

Maurice Peter Herlihy

May, 1984

This research was supported in part by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-75-C-0661.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

Replication Methods for Abstract Data Types

by

Maurice Peter Herlihy

Abstract

Submitted to the
Department of Electrical Engineering and Computer Science
on 5 May, 1984 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy.

Replication can enhance the availability of data in a distributed system. This thesis introduces a new method for managing replicated data. We propose new techniques to address four problems associated with replication: (i) the representation and manipulation of replicated data, (ii) concurrency control, (iii) on-the-fly reconfiguration, and (iv) enhancing availability in the presence of partitions.

Unlike many methods that support replication only for uninterpreted files, our method makes use of 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. An analysis of the algebraic structure of the data type is used to derive a set of constraints on quorum intersections. Any choice of quorums that satisfies these constraints yields a correct implementation, and it can be shown that no smaller set of constraints guarantees correctness.

By taking advantage of type-specific properties in a general and systematic way, our method can realize a wider range of availability properties, more concurrency, more flexible reconfiguration, and better tolerance of partitions than existing replication methods.

Thesis Supervisor: Barbara H. Liskov

Title: Professor of Computer Science

Keywords: abstract data types, atomicity, availability, concurrency control, distributed systems, partitions, reconfiguration, reliability, replication.

Acknowledgments

I would like to thank my advisor, Barbara Liskov, and my readers, Dave Gifford and Dave Reed, for indispensable assistance in clarifying both my ideas and their presentation. I am grateful to Bill Weihl for many invaluable suggestions, to Sheng-Yang Chiu for many useful discussions, and to the other members of the Programming Methodology group for their companionship and moral support.

My term as a graduate student has been rendered easier by the generous financial support of the GenRad Foundation, the International Business Machines Corporation, and the Monday Night Poker Gang.

Finally, I would like to thank Ellen Laviana, for hardly ever doubting that I would finish, and my parents, for everything.

CONTENTS

FIGURES

Chapter One

Introduction

1.1 Motivation

A *distributed system* consists of multiple computers (called sites) that communicate through a network. One way in which distributed systems differ from more traditional centralized systems is in the nature of failures. The components of a centralized system typically depend on one another to the extent that the failure of a single component disables the entire system. The components of a distributed system, however, are autonomous computers, and the failure of an individual site need not disable the entire system. One advantage of distributed systems over centralized systems is that valuable data can be stored redundantly at multiple locations, a practice commonly called *replication*. 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. As a simple example, consider how one might manage a file containing extremely valuable information (perhaps a Ph.D. thesis on the verge of completion). Rather than keeping one copy of the file at a single site, it might be prudent to keep multiple copies at different sites. Replication increases the file's availability: if one copy is temporarily inaccessible, work might progress using a different copy. Replication also increases the file's reliability: if one copy is accidently destroyed, it might be possible to reconstruct it from the other copies. Finally, replication can enhance performance by permitting the file's owner to work on the copy that can be most easily accessed. Of course, care must be taken that the replicas of the file are managed properly: enhanced availability will be of little use if the file's owner erroneously edits or prints obsolete versions of the file.

This thesis introduces a new method for managing replicated data in a distributed system. We propose new techniques to address four problems associated with replication: (i) the representation and manipulation of replicated data, (ii) concurrency control, (iii) on-the-fly reconfiguration, and (iv) enhancing availability in the presence of partitions. Unlike many methods that support replication only for uninterpreted files, our method makes use of type-specific properties of objects (such as sets, queues, or directories) to provide more effective replication. Our method can realize a wider range of availability properties, more concurrency, more flexible reconfiguration, and better tolerance of partitions than existing replication methods. The techniques introduced in this thesis are primarily intended to enhance availability, but many can also be used to enhance reliability and performance.

1.2 Contributions

The first contribution of this thesis is the introduction of a new method for representing and managing replicated data. Each data object belongs to a *type*, which characterizes its behavior by defining a range of possible values and a set of primitive operations that provide the (only) means of manipulating objects of that type. For example, an object of type *File* provides operations to read and write its contents, and an object of type *Queue* provides operations to enqueue or dequeue items. In this thesis, we show how replication can be performed more effectively by taking into account the type of the data. Most existing methods support replication only for files, which are treated as containers for uninterpreted data. Programs that manipulate replicated data typically store each object in a replicated file, translating the operations that manipulate the item into read and write operations for the underlying file. In our method, by contrast, the properties of the type are used to derive the replication method. For example, a replicated implementation of a queue would differ from a replicated implementation of a file.

The basic replication method has two novel aspects: a new technique for representing and manipulating the replicated data, and a new method for deriving constraints on correct

implementations. Our method is both general and systematic. It is general because it is applicable to objects of arbitrary type, and it is systematic because the constraints are derived directly from the specification of the data type in question. The constraints derived in this way are both necessary and sufficient: any replicated implementation satisfying these constraints is correct, and no smaller set of constraints guaranteees correctness. Our method illuminates the relation between the structure of a data type and the kinds of availability properties that can be realized by replication.

Our second contribution is the investigation of concurrency control techniques for replicated data. Concurrency control for replicated data is more difficult than concurrency control for data residing at a single location because the information used to make scheduling decisions may be scattered among multiple sites, and care must be taken that it is managed properly. We have identified two alternative approaches to implementing concurrency control for replicated objects: a simple method employing a static classification of operations into potentially conflicting sets, and a more complex method that employs state information to decide when operations conflict. The more complex method supports a higher level of concurrency, but it may require additional message traffic. For certain data types, there is a second trade-off between availability and concurrency: support for a high level of concurrency may impose additional constraints on availability, and certain kinds of availability properties may restrict the permissible level of concurrency.

Our third contribution is a general technique for on-the-fly *reconfiguration*: changing how existing data is replicated. Reconfiguration is useful for adapting to changes in requirements, improvements in hardware, and as a technique for masking failures. We propose a general method for organizing reconfigurable replicated data in which the ability to reconfigure can be provided in a way that imposes a one-time cost only when reconfiguration is actually carried out.

A partition occurs when sites are divided into disjoint sets such that functioning members of

different sets cannot communicate for a period of time. Our fourth contribution is a new technique for enhancing the availability of data in the presence of partitions. Our technique preserves serializability without introducing static transaction classes or complex protocols for detecting partitions.

1.3 Model of Computation

This section introduces our model of computation. We describe our assumptions about the hardware base, our assumptions about the effects of failures, as well as describing the logical organization of replicated data.

A distributed system consists of a collection of *sites* connected (only) by a communication 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 very high probability) and discarded. Transient communication failures may be hidden by lower level protocols, but longer-lived failures can result in situations where functioning sites are unable to communicate. One such situation is a network *partition*, in which the sites are divided into disjoint sets such that functioning members of different sets cannot communicate. Partitions are not the only such situations that can arise: for example, one site may be able to send messages to another, but not vice-versa.

A failures 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 indicate that the original message was lost, that the reply was lost, that the recipient has crashed, or simply that the recipient is slow to respond.

Our replication methods depend on the ability to preserve certain consistency constraints 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 state transitions are *atomic*: that is, indivisible and recoverable. By *indivisible*, we mean the execution of one state transition never appears to overlap (or contain) the execution of another, and by *recoverable*, we mean the overall effect of a state transition is all-or-nothing: it either succeeds completely, or it has no effect.

We assume that the system provides the ability to make activities atomic. Atomic activities are called *transactions*. A transaction 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 transactions is *serializable* [Eswaren 76, Papadimitriou 79]: their overall effect is as if they had been executed in some sequential order. We assume that failures that prevent a transaction from completing correctly are turned into aborts using techniques such as commit protocols [Eswaren 76, Skeen 82]. To simplify our discussion, we focus on single-level transaction systems, although we occasionally remark on how our techniques can be generalized to a system providing nested transactions [Reed 83, Moss 81].

Serializability is not the only interesting criterion for evaluating concurrency control mechanisms. As noted by Lamport [Lamport 78], Reed [Reed 78], Gifford [Gifford 82], and others, another important property is *external consistency*, which is informally defined to be the property that if events do not occur close together in time, then their serialization

ordering as imposed by the concurrency control mechanism agrees with their ordering as observed from outside the system. The importance of external consistency can be illustrated by a simple example. Consider a bank customer who deposits $100 on Monday, withdraws $50 on the following Friday, and is assessed a penalty when the deposit is serialized after the withdrawal. Our basic replication method preserves external consistency using approximately synchronized clocks, but our technique for coping with partitions sacrifices external consistency to increase availability.

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. Following [Bernstein 81], replicated objects are implemented by two kinds of modules: Data Managers (DM's) and Transaction Managers (TM's). The object's state is replicated among the DM's and managed by the TM's. DM's serve as long-term repositories for the object's state, while TM's execute the object's operations on behalf of the object's clients. To apply an operation to a replicated object, a client sends a request to a TM for the object. The TM reads the data from some collection of DM's, carries out a local computation, sends updates to some collection of DM's, and returns a response to the client. Each operation is executed atomically.

An operation's availability to a client depends on two factors: the client must first be able to locate a TM for the object, and the TM must in turn locate enough DM's to carry out the operation. Because the TM's do not interact directly with one another, new TM's can be created without affecting existing ones. Consequently, TM's can be replicated to an arbitrary extent, implying that the availability of the replicated object is determined by the availability of the DM's.

1.4 Related Work

The replication methods described in this thesis have been primarily influenced by Gifford's quorum consensus method [Gifford 79, Gifford 82]. Daniels and Spector [Daniels 83] have adapted Gifford's method to implement replicated directories. Each of these schemes can be viewed as a specially optimized case of the replication method proposed in this thesis. These proposals are discussed in more detail in Chapter Two.

Our replication method employs a combination of logs and timestamps to represent replicated data. We make use of timestamp generation schemes due to Lamport [Lamport 78], Dubourdieu [Dubourdieu 82], and Chan et al. [Chan 82]. Logs have been used to facilitate recovery in databases [Gray 78], and as a technique for achieving reliably coordinated updates [Lampson 79]. In the Swallow repository [Reed 80], an object is represented by a sequence of versions, a mechanism with many of the same properties as a log.

Replication methods for files have been proposed that relax the requirement that the value read from a file should be the last value written. Replication methods such as the ones proposed by Thomas [Thomas 78], Johnson and Thomas [Johnson 75], the Grapevine registry [Birrel 81], and the Clearinghouse name server [Oppen 81] implement a kind of non-deterministic file that provides only a probabilistic guarantee that the value read is the value most recently written. These schemes enhance performance and availability at the cost of more complex behavior on the part of the replicated file. By treating non-deterministic files as a distinct data type, our method can be applied to make the same trade-off between performance and availability on the one hand, and determinism on the other. These issues are discussed in Chapter Two.

Minoura *et al.* [Minoura 79] have proposed a scheme for fully replicated databases in which the result of merging information from any subset of sites defines a legal,

transaction-consistent state, i.e. one that reflects a subsequence of the sequence of transactions that actually occurred. This property is useful for recovering from catastrophic failures, because the surviving state will satisfy any invariants preserved by transactions. Our replication method can be made to satisfy a similar property. In the basic method, the result of merging information from a subset of sites defines a legal state (i.e. one satisfying the type specification) but not necessarily a transaction-consistent state. Our method can be extended to support transaction consistency for individual replicated objects (which are analogous to fully replicated databases).

In our discussion of concurrency control in Chapter Three, we make use of locking-based schemes similar to those proposed by Eswaren [Eswaren 76] and Moss [Moss 81], and Reed's multi-version timestamp scheme [Reed 83, Reed 78]. We also make use of a general model of atomicity for objects of arbitrary type due to Weihl [Weihl 83a, Weihl 84], which is used as a framework for evaluating our techniques.

Our replication method masks failures as long as enough sites remain available. An alternative technique for fault-tolerance is to employ active measures to detect and compensate for failures. This approach is taken in existing replication methods such as the *primary copy* method [Alsberg 76, Stonebreaker 79] and the *true-copy token scheme* [Minoura 79, Minoura 82]. These schemes are discussed in Chapter Four, where we show that they can be modeled by a simple extension to our method.

A formal model for concurrency control in replicated databases has been proposed by Bernstein and Goodman [Bernstein 83]. This model is exclusively concerned with replicated files, which are objects that provide only *Read* and *Write* operations. A basic result of this thesis is that replication methods for files place unnecessary restrictions on availability and concurrency when used to implement objects of other types. A disadvantage of the Bernstein and Goodman model is that its basic notions are defined directly in terms of the particular semantics of *Read* and *Write* operations, and it is not obvious how to generalize

the model to accommodate data types that provide an arbitrary set of operations.

Eager and Sevcik [Eager 83] have extended Gifford's replication method for files to provide increased availability in the presence of partitions. Transactions executing in distinct partitions are allowed to use slightly different replication methods for the same file. The range of permissible replication methods is constrained to ensure that all transactions executing in one partition can be serialized before any transaction executing in another. In Chapter Five, we introduce a related, but more powerful technique that applies to objects of arbitrary type, not just to files, and that provides more effective recovery when partitions are rejoined.

Both the SDD-1 distributed database [Hammer 80] and the *available copies* replication algorithm [Goodman 83] employ a kind of dynamic reconfiguration. In SDD-1, messages for a site that has crashed are redirected to a spooler for later delivery. In the available copies algorithm, a crashed site is configured out of the system after it crashes, and is reinitialized and configured back in when it recovers. These methods are not directly comparable to ours because they make very different assumptions about faults. While our methods tolerate both site crashes and communication failures, these methods tolerate only site crashes, and they do not function correctly in a model of computation that admits communication failures such as partitions.

The Locus operating system [Popek 81] provides replicated files and directories. If a partition occurs, inconsistencies between replicas in different partitions are allowed to develop, but application-dependent measures are used to reconcile them when the partitions are rejoined.

A very different approach from ours is taken by Lamport [Lamport 78a] and by Schlichting and Schneider [Schlichting 81], in which *Byzantine Agreement* [Lamport 82] is used to tolerate arbitrary failures. In Lamport's scheme, processes as well as data are replicated, and Byzantine Agreement is used for broadcasts. Schlichting and Schneider's scheme uses

a similar technique to implement *fail-stop processors*, which are processors that halt before taking an incorrect step. The correctness of these schemes depends on the accuracy of predictions about the system's real-time behavior, and each replicated process must be deterministic.

1.5 Overview of Thesis

Chapter Two introduces a new replication method called *General Quorum Consensus*. A novel aspect of this method is that the object is not represented by a collection of copies; instead, each site maintains a partial log of the operations that have been applied to the object. Associated with each operation 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 algebraic structure of the object's type is used to derive a set of constraints on quorum intersections. Any choice of quorums that satisfies these constraints yields a correct implementation, and it is shown that no weaker set of constraints can guarantee correctness. By providing direct support for objects of arbitrary abstract type, General Quorum Consensus imposes fewer constraints on quorum choice than existing replication methods based on files.

Chapter Three discusses concurrency control for replicated objects. The correctness of General Quorum Consensus depends on the preservation of certain invariants relating the data at multiple sites. To ensure that these invariants are preserved in the presence of concurrency and failures, the activities that manage replicated data must be *atomic*: each activity either succeeds completely or has no effect, and each activity appears to take place instantaneously. Two concurrency control mechanisms are proposed. The first is a mechanism in which scheduling decisions are made on the basis of a predefined set of conflicts between pairs of operations. This mechanism is efficient, and it is shown to be optimal for the amount of information it uses: no mechanism based entirely on predefined operation conflicts can provide a higher level of concurrency. The second concurrency

control mechanism we propose is one in which scheduling decisions may take the object's state into account. It can support a higher level of concurrency than the conflict-based method because it takes advantage of additional information. It can implement any concurrency control scheme that preserves serializability, but it requires more message traffic, and it may impose additional restrictions on quorums. By taking advantage of type-specific properties, both methods can achieve a higher level of concurrency than existing replication methods based on files.

Chapter Four addresses the problem of reconfiguration: how to change an object's quorums in response to changes in requirements or changes in hardware. We propose a reconfiguration technique in which objects employ multiple levels of indirection. Each level of indirection requires an additional round of messages. The final round manipulates the actual data of interest to the client, and each previous round is used to discover the quorum for its successor. Multiple rounds of logical messages can sometimes be carried out by a single exchange of physical messages. Existing replication methods for files such as the *primary copy* scheme [Alsberg 76, Stonebreaker 79] and the *true-copy token* scheme [Minoura 79, Minoura 82] can be viewed as specially optimized instances of the multi-round method.

Chapter Five shows how to extend General Quorum Consensus to provide increased availability in the presence of *partitions*, which occur when the sites are divided into two or more disjoint sets such that functioning members of distinct sets cannot communicate. Unlike some replication methods [Hammer 80, Goodman 83, Popek 81] our method preserves serializability in the presence of partitions, but a client executing in the wrong partition may be unable to execute certain operations. In Chapter Five, we propose an extension to the basic method in which a replicated object may provide a *range* of alternative quorums. Each transaction chooses the quorums that are best suited to the partition in which it is executing. Concurrent transactions executing in distinct partitions can each progress employing different quorums. Constraints on the quorums an object can

provide ensure that the effect of transactions executing in distinct partitions remains serializable.

Chapter Six summarizes the basic ideas of the thesis.

Chapter Two

General Quorum Consensus

2.1 Introduction

This chapter introduces General Quorum Consensus, a new replication method for objects of arbitrary type. Because a top-down presentation of a replication algorithm general enough for objects of arbitrary type is necessarily rather abstract, we have chosen to introduce our algorithm through a series of concrete examples. We first review a replication method for files due to Gifford. We then consider how this approach can be adapted to the problem of constructing a replicated FIFO queue. We observe that a direct application of the file replication method to queues is not entirely satisfactory, and we propose some changes to Gifford's method that provide better support for constructing a replicated queue. In the next step, we show how the revised replication method for queues can be generalized to construct replicated objects of arbitrary type. We end the chapter with a series of examples.

We use the model of computation described in Chapter One. A replicated object is implemented by a collection of *Data Manager* guardians (DM's) and a collection of *Transaction Manager* guardians (TM's). The DM's are repositories for the object's state information, while the TM's mediate between clients and the data replicated at the DM's.

In this chapter, we use the following criteria for evaluating replication methods.

- The range of availability properties that can be realized by the method. For example, a file replication method that allows the availability of *Read* and *Write* to be traded off is more flexible than one that requires each operation to be equally available.

- The number and size of the messages needed to execute an operation.
- The amount of data stored at each DM.

In this chapter, we consider replication methods built on top of a transaction system, making no assumptions about how transactions are implemented or about the level of concurrency the transaction system can support. In the next chapter, however, we show that a high level of concurrency can be achieved if the concurrency control method is designed in conjunction with the replication method.

2.2 Gifford's Replication Method

In this section, we describe a replicated method due to Gifford [Gifford 79, Gifford 82]. This method was influential in the development of our method, and it presents a convenient starting point for our discussion. We first describe how this method can be used to construct a replicated file, and we then apply the method to the construction of a replicated FIFO queue. We finish the section by pointing out some shortcomings of the queue implementation, which we proceed to remedy in the next section.

2.2.1 A Replicated File Implementation

A *quorum* for an operation is defined to be any set of DM's whose cooperation is sufficient to execute that operation. It is convenient to divide a quorum into two parts: a TM 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 DM's 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 2.6.

A *request* is an operation invocation together with its argument values, and a *response* is a termination condition and results. An *event* is a pair consisting of a request and its associated response. For example, *Read*() and *Write*(*x*) are requests, *Ok*(*x*) and *Ok*() are responses, and [*Read*();*Ok*(x)] and [*Write*(*x*);*Ok*()] are events.

In Gifford's scheme, each DM 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 TM 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 TM first reads the version numbers from an initial *Write* quorum of DM's. The TM 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:

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

The first constraint ensures that each *Read* request will observe the effects of the most recent *Write*, and the second constraint ensures that each new version created by a *Write* request 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 indicates that each final quorum for operations in the first class must intersect each initial quorum for operations in the second class. Very informally, the direction of the arrow can be considered the direction of information flow.

Figure 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 indicates that every final quorum for *Write* must intersect every initial quorum for *Read*, and the directed edge from the *Write* vertex to itself indicates that every pair of initial and final quorums for *Write* must intersect.

In Figure 1b we display the range of quorum choices for an object replicated among five identical DM's.[1] An entry of the form (m,n) indicates that any m DM's constitute an initial quorum for the request and any n DM's constitute a final quorum for the event. We define the quorum *size* to be $max(m,n)$, the smallest number of DM's that includes both an initial and a final quorum. In this example, and in subsequent examples, we restrict our attention to quorum choices whose sizes are minimal. The three columns correspond to the three minimal quorum choices. (Given n identical DM's, Gifford's replication method permits $\lceil n/2 \rceil$ minimal quorum choices.) For example, the first column corresponds to a quorum choice in which an initial quorum for *Read* consists of any one DM and a final quorum for *Write* consists of all five DM's.

As discussed in [Gifford 79], one convenient way to characterize quorums is to assign votes to DM's so that a collection of DM's is a quorum if and only if its votes exceed a chosen threshold value. Two quorums will intersect if the sum of their threshold values exceeds the total number of votes assigned to all DM's. Not all quorum choices can be captured by voting, however. Consider a file replicated among four DM's, where *Read* quorums must contain either DM1 and DM2 or DM3 and DM4, and *Write* quorums must contain either DM1 and DM3 or DM2 and DM4.

---

1. Neither Gifford's method nor the methods we introduce later requires that all DM's be treated identically, but it is convenient to do so for examples.

Fig. 1. File Replication using Gifford's Method

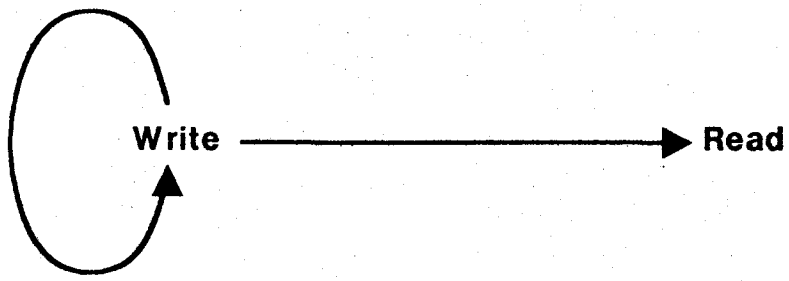

Figure 1a: the quorum intersection graph.

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

Figure 1b: Minimal quorum choices for five identical DM's.

There are three points about Gifford's method that deserve emphasis. First, it does not make sense to talk of the level of availability for the object as a whole, because executions of different operations can have different levels of availability. Instead, an object's fault-tolerance must be characterized by the availability of each of its events. Second, the set of quorums for an event completely determine its availability: an operation invocation will succeed if and only if an appropriate quorum is available. Third, the constraints on quorum choices completely determine the kinds of availability properties that can be 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 DM's, implying

that *Write* cannot be more highly available than *Read*.

2.2.2 A Replicated FIFO Queue Implementation

A replication method for files can be used to replicate any type of object simply by using a file to represent the object. For example, one could construct a replicated queue by encoding the queue's state in a bit-string, storing the bit-string in a file, and translating the *Enq* and *Deq* operations into *Read* and *Write* operations for the underlying file. In this section we examine a replicated queue implementation based on exactly this approach. The purpose of this example is to illustrate the benefits of an alternative replication method that takes advantage of type-specific properties of queues.

A *Queue* has two operations: *Enq* places an item in the queue, and *Deq* removes the least recently enqueued item, raising an exception if the queue is empty. A replicated queue might be used as a highly available spooler or as a highly available scheduler for off-line activities.

Let us consider the constraints on quorum choice imposed by the replication method described in the previous section. To dequeue an item, the TM reads the versions of the queue from an initial *Read* quorum of DM's. If the queue is non-empty, the appropriate item is removed, and the shortened queue is written out to a *Write* quorum. If the queue is empty, an exception is returned to the client, but the DM's need not be updated because the queue's state is unchanged. *Enq* is also implemented as a *Read* followed by a *Write*.

A *normal Deq* event is one that returns an item, while an *abnormal Deq* is one that signals *Empty*. The constraints on *Read* and *Write* quorums impose the following constraints on quorums for *Enq* events and normal *Deq* events. (Abnormal *Deq* events have empty final quorums.)

1. Every initial *Enq* quorum must intersect every final *Enq* quorum.
2. Every initial *Enq* quorum must intersect every final *Deq* quorum.
3. Every initial *Deq* quorum must intersect every final *Enq* quorum.
4. Every initial *Deq* quorum must intersect every final *Deq* quorum.

Figure 2a displays the required quorum intersections for *Enq* events, normal *Deq* events, and abnormal *Deq* events. If we consider only the quorums for *Enq* events and normal *Deq* events, Figure 2b illustrates the only minimal quorum choice for a queue replicated among five identical DM's[1] In fact, there is exactly one minimal quorum choice for *n* identical DM's, because both *Enq* and normal *Deq* quorums must encompass a majority of DM's.

These quorum choices appear to be 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 our method, but our method permits quorums that the file-based method does not. Consequently, the revised method can provide availability properties that the file-based method cannot.

2.3 A Revised Replication Method

In this section we introduce two technical changes to Gifford's method that support more flexible quorum choices for replicated queues.

- Timestamps are used in place of version numbers.

---

1. We are neglecting quorum choices such as *Enq* = (1,5), normal *Deq* = (1,5), and abnormal *Deq* = (1,0) because we feel that any choice favoring the availability of abnormal *Deq* events over normal *Deq* and *Enq* events is unlikely to be of practical interest.

Fig. 2. A File-based Representation for a Queue

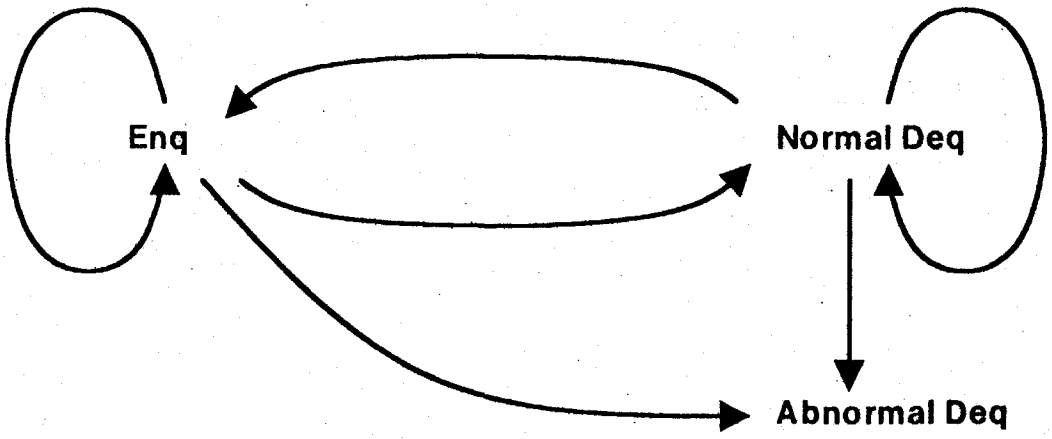

Figure 2a: The quorum intersection graph.

Enq	(3,3)
Normal Deq	(3,3)
Abnormal Deq	(3,0)

Figure 2b: Minimal quorum choices for five identical DM's.

- Logs are used instead of versions.

These techniques are described in the next sections.

2.3.1 Timestamps vs. Version Numbers

In this section, we show how to implement replicated files in a way that eliminates the constraint that *Write* quorums 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.

The basic idea behind the improved method is quite simple. We assume that it is possible to assign unique timestamps to transactions in such a way that:

1. Transactions are serializable in timestamp order.
2. Timestamp generation never fails.

The techniques used to assign timestamps to transactions depend on the way the transaction mechanism is implemented. Timestamp generation is relatively easy for transaction mechanisms already based on timestamps [Reed 83], and is slightly more difficult for a transaction mechanism based on two-phase locking [Dubourdieu 82, Chan 82]. We postpone discussion of the techniques used to assign timestamps to transactions to Chapter Three, where we discuss concurrency control.

The file replication method is modified as follows. Rather than reading the version numbers from an initial *Write* quorum, the TM generates a timestamp with the following fields:

1. The transaction's timestamp is used for the timestamp's high-order bits. This field is used to compare timestamps generated by distinct transactions.
2. A value from an internal transaction clock is used for the low-order bits. This field is used to compare timestamps generated by a single transaction.

The implementation of the *Read* operation is largely unchanged: rather than choosing the version with the latest version number, the TM chooses the version with the most recent timestamp. The implementation of the *Write* operation is more interesting: rather than reading and advancing the most recently generated version number, the TM simply generates a new timestamp and appends it to the new version of the file.

The timestamp-based scheme imposes fewer constraints on quorum choice. Because a TM executing a *Write* request does not need to observe previous version numbers, *Write* quorums are no longer required to intersect. The quorum intersection graph for a replicated file based on timestamps is shown in Figure 3a. Figure 3b shows the minimal quorum

choices for a file replicated among five identical DM's. The number of quorum choices for *n* identical DM's has effectively doubled, going from $\lceil n/2 \rceil$ to *n*.[1]

The ability to define non-intersecting *Write* quorums implies that the *Write* operation can be made more highly available than the *Read* operation. Such quorum choices facilitate "blind writes," in which a file is written without first being read. For example, if transactions **A** and **B** write to disjoint final quorums, then a later transaction 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* [Thomas 78].

Because files are typically read before they are written, we do not believe that quorum choices facilitating blind writes are of significant practical interest for files. Nevertheless,

---

Fig. 3. File Replication using Timestamps

Figure 3a: the quorum intersection graph.

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

Figure 3b: Minimal quorum choices for five identical DM's.

---

1. As before, we are neglecting quorum choices such as *Enq* = (0,5), normal *Deq* = (1,5), and abnormal *Deq* = (1,0) that favor the availability of abnormal *Deq* events over normal *Deq* and *Enq* events.

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*'s 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. Finally, the improved method is of theoretical interest, as it demonstrates that the requirement that *Write* quorums intersect is not really a fundamental constraint, but rather an artifact of a particular version numbering scheme.

2.3.2 Logs vs. Versions

Instead of replicating versions of the queue, consider the consequences of storing a *log* of events at each DM. 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 DM's, and the log at any particular DM may have arbitrary gaps.

An entry is denoted by:

```
t0: [op(args);term(results)]
```

where *t0* is a timestamp, *op* is the operation name, *args* are zero or more argument values, *term* is the termination condition name, and *results* are zero or more result values. We use *Ok* to denote the normal termination condition.

For queues, an entry consists of an *Enq* or *Deq* request, its arguments, termination condition, and results, together with a timestamp. To enqueue an item *x*, the TM creates a timestamped *Enq* entry naming the item to be enqueued:

```
t0: [Enq(x);Ok()]
```

This entry is appended to each log at a final *Enq* quorum of DM's.

A *Deq* is carried out in the following steps:

1. The TM reads the logs from an initial *Deq* quorum of DM's. These entries are merged in timestamp order, discarding duplicates. The resulting log is called a *view*.

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

3. If the queue is non-empty, the TM records the *Deq* event by appending a new entry to the view and sending the modified view to a final *Deq* quorum of DM's, where it is merged with the resident logs.

4. Finally, the response (the dequeued item or an exception) is returned to the client.

To show that this method works, it is necessary to demonstrate that the view defines a legal state for a queue, and that the view contains sufficient information to determine the correct response for the *Deq*.

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 actual 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. These optimizations are discussed in Section 2.6. 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 2.5 we present a systematic method for establishing constraints on quorum intersections, but for now we rely on informal arguments. If each initial quorum for a request 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 request. 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 request. To ascertain the item at the head of the queue, a TM executing a *Deq* must observe: (i) which items have previously been enqueued, and (ii) which of these items have subsequently been dequeued. The requirements translate into the following constraints on quorum intersection for *Enq* and normal *Deq* events:

1. Every initial *Deq* quorum must intersect every final *Enq* quorum.
2. Every initial *Deq* quorum must intersect every final *Deq* quorum.

The view for an *Enq* request need not include any prior events, because *Enq* returns no information about the queue's state. As a consequence, initial *Enq* quorums may be empty. As before, an abnormal *Deq* has an empty final quorum.

To illustrate this method, let us examine how a queue might be replicated among three DM's. In this example, initial quorums for *Enq* are empty, and final quorums for *Enq* consist of any two DM's. Initial quorums for *Deq* consist of any two DM's, as do final quorums for normal *Deq* events. Abnormal *Deq* events have empty final quorums.

Let us trace a short execution for the queue. The queue is initially empty. An item *x* is enqueued by appending a log entry with timestamp *t1* to the logs at two DM's, say DM1 and DM2:

DM1	DM2	DM3
t1: [Enq(x);Ok()]	t1: [Enq(x);Ok()]	

To dequeue *x*, a TM reads the logs from DM2 and DM3, merging the empty log from DM3 with the single-entry log at DM2. The resulting view indicates that *x* is the only item in the queue. The TM creates a *Deq* entry with timestamp *t2*, and writes out the entire log to DM2 and DM3:

DM1	DM2	DM3
t1: [Enq(x);Ok()]	t1: [Enq(x);Ok()]	t1: [Enq(x);Ok()]
	t2: [Deq();Ok(x)]	t2: [Deq();Ok(x)]

In a similar way, Item *y* is enqueued at DM1 and DM2 with timestamp *t3*, and *z* is enqueued at DM1 and DM3 with timestamp *t4*. For readability, a "missing" entry at a DM is shown as a blank space.

DM1	DM2	DM3
t1: [Enq(x);Ok()]	t1: [Enq(x);Ok()]	t1: [Enq(x);Ok()]
	t2: [Deq();Ok(x)]	t2: [Deq();Ok(x)]
t3: [Enq(y);Ok()]	t3: [Enq(y);Ok()]	
t4: [Enq(z);Ok()]		t4: [Enq(z);Ok()]

At this point, the log at each DM defines a legal queue, but no single DM contains all the entries that define the queue's state. Finally, a TM dequeues *y* by reading from and updating DM1 and DM3.

DM1	DM2	DM3
t1: [Enq(x);Ok()]	t1: [Enq(x);Ok()]	t1: [Enq(x);Ok()]
t2: [Deq();Ok(y)]	t2: [Deq();Ok(x)]	t2: [Deq();Ok(x)]
t3: [Enq(y);Ok()]	t3: [Enq(y);Ok()]	t3: [Enq(y);Ok()]
t4: [Enq(z);Ok()]		t4: [Enq(z);Ok()]
t5: [Deq();Ok(y)]		t5: [Deq();Ok(y)]

Figure 4a shows the quorum intersection graph for the log-based implementation of a replicated queue, and Figure 4b shows the quorum choices for a queue replicated among five identical DM's. As one can see by comparing Figures 2 and 4, the log-based implementation places fewer constraints on quorum intersection than the file-based implementation, and therefore permits a wider range of quorum choices and hence a wider range of availability trade-offs. Given *n* identical DM's, there is only one minimal quorum

choice for the file-based implementation (both *Enq* and *Deq* quorums encompass a majority of DM's) but there are $\lceil n/2 \rceil$ minimal quorum choices for the log-based implementation (only *Deq* quorums require a majority). In each of the new quorum choices, 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.

This example illustrates an important point: the revised method provides more freedom in

---

Fig. 4. A Log-Based Representation for a Queue

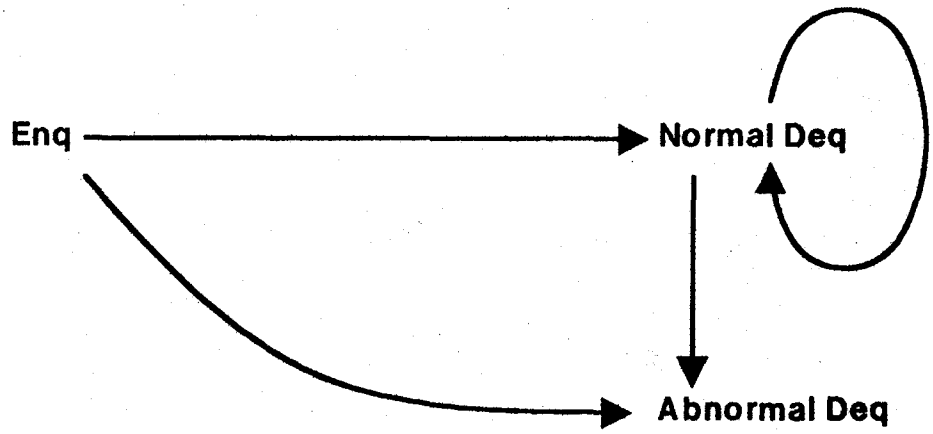

Figure 4a: The quorum intersection graph.

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

Figure 4b: Minimal quorum choices for five identical DM's.

trading off availability between the *Enq* and *Deq* operations than does the file-based method. This additional flexibility arises because logs and timestamps permit a finer grain of control over information flow. In the file-based method, a version of an object is either current or obsolete. If the version is current, then it contains the object's entire state, while if it is obsolete, it contains no information about the object's current state. 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 by merging their entries in timestamp order.

In summary, the additional flexibility of our method requires the use of both logs and timestamps. Although version numbers could be used to order log entries, the resulting quorum choices are more constrained. A TM executing an *Enq* would have to read the version numbers from enough DM's to ensure that it had observed the version number of the most recent prior *Enq*. Consequently, each initial quorum for *Enq* would have to intersect each final quorum; otherwise *Enq* entries might not be ordered correctly. Using timestamps, each *Enq* entry is ordered correctly without the need to observe previous *Enq*'s.

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 subsequent 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 DM can garbage collect its own log, or collections of DM's can cooperate to identify and discard out-of-date entries. Message sizes can also be reduced if TM's maintain a cache of log entries, requesting only the missing entries from DM's. These optimizations are type-specific, in the sense that an optimization that works for queues will not necessarily work for files. A number of optimizations are discussed in Section 2.6.

2.4 Replicating Objects of Arbitrary Type

We are now ready to address the main point of this chapter. In this section, we present an informal description of how the replication method described above for queues can be generalized to objects of arbitrary type. In the next section, we present a more formal description, together with correctness arguments.

The file and queue examples suggest the following approach to analyzing replication methods for types other than files. A quorum consensus method is characterized by a set of constraints on quorum intersections. These constraints determine the level of availability that can be provided for each individual operation, as well as the trade-offs among the levels of availability of various operations. Given a set of constraints, we need to determine whether it is correct and whether it is optimal. By *correct*, we mean that any quorum choice 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 this section, we propose a replication method that permits correct and optimal constraints on quorum intersections to be derived directly from the type specification for an arbitrary data type.

An object may be viewed as an automaton that accepts requests and returns responses. As before, a request consists of an operation name and argument values, a response is a termination condition and result values, and an event is a paired request and response. A computation is characterized by a *history* of the events that have occurred since the object's creation. Just as for queues, an object's state is given by a collection of logs residing at DM's. Log entries are partially replicated; each log entry might be stored at several DM's, and each DM might store only a fragment of the entire log. Each request is assigned a set of

initial quorums, and each event is assigned a set of final quorums. As discussed below, the object's set of legal histories is analyzed to derive a set of constraints on quorum intersections. An operation is executed in the following four steps.

1. When a TM receives a request from a client, it reads the logs from an initial quorum of DM's for that request. These logs are merged in timestamp order to construct a larger log called a *view*.

2. The TM chooses a response consistent with the object state as defined by the view.

3. The TM records the request and response by appending a new entry to the view, and sending the modified view to a final quorum of DM's for that event. Each DM in the final quorum merges the view with its resident log.

4. The response is returned to the client.

The first property desired of this method is the following *partial correctness* property:

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

This property ensures that every history generated by the replicated object is legal.

Just as for files and queues, correctness is ensured by constraining certain quorums to have non-empty intersections. Constraints on quorum intersection are derived from a *dependency relation* between requests and responses. A precise definition of dependency is given in the next section, but for now it suffices to say that a request depends on an event if omitting that event from a request's view might produce an incorrect response. For example, *Read* requests depend on *Write* events, because the correct response to *Read* requires knowledge of prior *Write*'s. *Write* requests do not depend upon prior *Read* or *Write* events, because the response to the *Write* is always the same. Similarly, *Deq* requests depend on prior *Enq* and normal *Deq* events, because knowledge of these events is needed to determine which item to dequeue. *Enq* requests do not depend any on prior events, because *Enq*, like *Write*, has only one possible response. In the next section, we show that

General Quorum Consensus is correct if and only if the view for each request is certain to include all prior events on which it depends. This property can be achieved if each initial quorum for the request is constrained to intersect each final quorum for the events. The resulting view does not necessarily contain the entire object state, but it contains enough information to choose a correct response.

Partial correctness is not the only property of interest. If the view assembled in Step One is not a legal history, then it would not be possible to choose a response in Step Two. To ensure this situation never arises, we also demonstrate the following *well-formedness* property:

Every view constructed for a request is a legal history.

In fact, we ensure a slightly stronger invariant: a history constructed by merging logs from any collection of DM's is legal. The invariant provides an incidental benefit in the form of *catastrophe insurance*: if some set of DM's becomes permanently inaccessible, the remainder still define a legal state that corresponds to a subhistory of the actual (lost) history. Catastrophe insurance might be useful for large databases or file systems. In addition to ensuring that the history reconstructed after a catastrophe is legal, it might also be desirable to ensure that it is *transaction-consistent*: that all or none of the entries for each transaction have survived. Transaction consistency can be guaranteed for an individual object if each transaction records the same set of entries at each DM for that object, but reconstructed histories for distinct objects may reflect different sets of transactions.

2.5 Correctness Arguments

This section presents correctness arguments for General Quorum Consensus. We first define the basic notions of specifications, objects, and logs, and then present a model for our replication method. We then define the notion of a *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 dependency relation for the object's type.

2.5.1 Specifications

Informally, an object's specification describes its externally observable behavior. Objects having the same specification are said to belong to the same *data type*. The specifications employed in this chapter 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 a transaction system. Our specifications actually model computations that have already been serialized by the underlying transaction system. In the next chapter, we discuss how a high level of concurrency can be achieved if the concurrency control method is integrated with the replication method.

A *specification* for an object is a tuple $\langle REQ, RES, LEGAL \rangle$, where *REQ* is a set of *requests*, *RES* is a set of *responses*, and *LEGAL* is a set of *legal histories* for that object. An *event* is a pair [*req*;*res*], where *req* is a request and *res* is a response.

$$EVENT = REQ \times RES$$

For example, "Enq(x)" is a request, "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* [Guttag 78]. In this thesis, we will rely on informally stated specifications.

2.5.2 Logs

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

$$l: TIMESTAMP \rightarrow EVENT$$

We use the notation $l(t) = \perp$ as shorthand to indicate 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:

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

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

If l is a log and e an event, we use the notation $l = m \circ e$ to indicate 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. We use the notation $l = l_1 \circ l_2$ to indicate 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.

Finally, we define a map H that associates with each log the history constructed by sorting its entries in timestamp order. Let $\{t_0,...,t_n\}$ be the set of timestamps for which l is defined, indexed in ascending order. The i-th event of H(l) is defined to be $l(t_i)$, for $0 \leq i \leq n$. A log l is said to be *legal* if H(l) is a legal history.

2.5.3 Dependency Relations

The correctness condition for General Quorum Consensus is based on the notion of a *dependency relation* between requests and events. Any relation $\succ$ between requests and events induces a corresponding relation between pairs of events, also denoted by $\succ$:

$[req;res] \succ [req';res']$ if and only if $req \succ [req';res']$.

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, 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 relation $\succ$ is a *dependency relation* if for all requests *req*, all legal logs l, all closed sublogs m of l containing the events e of l such that $req \succ e$, and for all responses *res*:

(2-1) $m \circ [req;res]$ is legal $\Rightarrow l \circ [req;res]$ is legal.

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

Perhaps the simplest example of a dependency relation is provided by the *File* type, for which *Read* requests depend on *Write* events, but *Write* requests 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* (i.e. *Ok*()) can be chosen from any closed sublog, including the empty log.

For deterministic types, in which each request has a single correct response, the implication goes in both directions, but for non-deterministic types, the implication goes in one direction only: there may be legal responses to *req* in l that are not evident from m. An example of such a non-deterministic type is given in the *SemiTable* example in Section 2.6.

A given type may have several dependency relations. For example, the relation in which every request depends on every event is a dependency relation, albeit not a very interesting one. More interesting dependency relations are the *minimal* relations, relations with the property that no smaller relation is a dependency relation. We remark that a type need not have a unique minimal dependency relation. An example of a data type with several distinct minimal dependency relations is given in the *Double_Buffer* example in Section 2.6.

We will see that 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 dependency relation.

2.5.4 Replication Schemes

A replication scheme is a tuple <S, *DM*, *INITIAL*, *FINAL*>, where S is a specification and *DM* is a set of DM's. We define the domain *QUORUM* to be the domain of sets of DM's:

$$QUORUM = 2^{DM}$$

INITIAL is a map that defines the correspondence between each request and its set of initial

quorums:

$$INITIAL: REQ \rightarrow 2^{QUORUM}.$$

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

$$FINAL: EVENT \rightarrow 2^{QUORUM}.$$

The quorum choices define a quorum intersection relation $\succ$ between requests and events:

(2-2) $req \succ [req';res']$ $\Leftrightarrow$ each initial quorum for the request *res* intersects each final quorum for the event $[req';res']$.

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

1. Choose an initial quorum I from *INITIAL*(*req*). Let v be the log constructed by merging the l_d for all *d* in I. We refer to v as the ***view***.
2. Choose a timestamp greater than any chosen in a previous step, and a result *res* such that $v \circ [req;res]$ is legal.
3. Choose a final quorum F from *FINAL*(*req*;*res*), and merge $v \circ [req;res]$ with the log at each *d* in F.
4. Output the response *res*.

In summary, after an object whose state is $\{l_d \mid d \in DM\}$ inputs a request *req*, it outputs a response *res*, and enters a new state given by $\{l'_d \mid d \in DM\}$, where:

$$l'_d = \text{if } d \in F \text{ then } l_d \cup v \circ [req;res] \text{ else } l_d$$

A computation involving a replicated object can be characterized by a *global log* containing

each event that has occurred since the object's creation. It is worth pointing out that an object's global log is not necessarily the log that would be constructed by merging the logs at all DM's at the end of the computation, because events having empty final quorums (like *Read* events) are contained in the global log but not in the log at any DM.

2.5.5 Well-Formedness and Correctness

We wish to demonstrate the following correctness property:

> 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 dependency relation. We argue by induction, demonstrating the invariance of certain properties relating an object's global log to the logs kept at the DM's. We begin with a few simple lemmas.

Lemma 2-3: The result of merging closed logs is a closed log.

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 2-4: If m is a closed sublog of l, then m is a closed sublog of $l \cdot e$.

Proof: Left to the reader. □

Lemma 2-5: 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 DM 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 demonstrate 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; our arguments demonstrate that the property is preserved by state transitions.

Theorem 2-6: Each DM's log is a closed sublog of the global log.

Proof: Let the global log be l, and let the states of the DM's be given by $\{l_d \mid d \in DM\}$. Now assume that the object undergoes a transition from l to $l' = l \circ e$, and let the new states of the DM's be given by $\{l'_d \mid d \in DM\}$. We show that if the l_d are closed in l, then the l'_d are closed in l'.

If d is not a member of the final quorum for e, then $l'_d = l_d$, so the result follows immediately from Lemma 2-4. Otherwise,

$$l'_d = l_d \cup v \circ e$$

where v is the view used to compute e. The view v is the result of merging logs from DM's, each of which is closed in l (induction hypothesis). It follows that v is closed in l (Lemma 2-3) and in l' (Lemma 2-4). The log $v \circ e$ is closed in l' because v is closed in l and contains every event on which e depends (Lemma 2-5). It follows that l'_d is closed because it is the result of merging the closed logs $v \circ e$ and l_d (Lemma 2-3). □

It follows that the result of merging logs from an arbitrary set of DM's is closed, and thus every view constructed for a request is closed.

Theorem 2-7: If the quorum intersection relation is a dependency relation, then the result of merging logs from any collection of DM's is legal.

Proof: Let S be an arbitrary set of DM's, and let l_S be the result of merging the logs from the DM's in S. let l'_S be the result of merging the logs from S following a transition from l to $l' = l \circ e$. 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. If S does intersect the final quorum for e, then the new event and its view have been merged with the log of one or more DM's in S, yielding:

$$l'_S = l_S \cup v \circ e$$

where v is the view used to compute e. The view v is the result of merging logs from DM's, each of which is closed in l by Theorem 2-6. It follows from Lemma 2-3 that v is closed in l. The log $(l_S \cup v)$ is legal by the induction hypothesis, since it is the result of merging logs from a collection of DM's. Because v is closed in l, it is also closed in the sublog $(l_S \cup v)$ of l, and it contains every event on which e depends (Lemma 2-5). The view v is also legal by the induction hypothesis. Because v is a closed legal sublog of $(l_S \cup v)$ that contains every event of $(l_S \cup v)$ on which e depends, the legality of $v \circ e$ implies the legality of $(l_S \cup v) \circ e$ by Property 2-1. But $(l_S \cup v) \circ e$ is the same as $l_S \cup v \circ e = l'_S$. □

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 result of merging the logs from any collection of DM's defines a legal sublog of the object's global log.

We now prove the correctness result.

Theorem 2-8: If the quorum intersection relation is a 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 $l \circ e$, the view v is closed in l (Theorem 2-6), legal (Theorem 2-7), and contains every event of l on which e depends (Lemma 2-5). We know that $v \circ e$ is legal by construction; thus from property 2-1 we deduce that $l \circ e$ is legal. □

We have just shown that General Quorum Consensus is correct if the quorum intersection relation is a dependency relation. We now show the converse: any quorum intersection relation that is not a dependency relation permits quorum choices that do not guarantee correctness.

Theorem 2-9: If the quorum intersection relation is not a dependency relation, then there exist quorum choices that generate illegal global logs.

Proof: We assume that the quorum intersection relation does not satisfy Property 2-1, and we construct a scenario in which an illegal global log is generated. If the quorum intersection relation is not a dependency relation, then there exists an event $e = [req;res]$, a log l, and a closed sublog m containing all events whose final quorums intersect the initial quorums for *req*, with the property that:

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

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

The logs are replicated at two DM's: DM1 and DM2. Each event in m chooses an initial quorum of DM1 and a final quorum of both DM1 and DM2. Each event in l but not in m chooses an initial quorum of DM1 and DM2, and a final quorum of DM2. As a result of these choices, each event in m observes only the prior events in m, and each event not in m observes all prior events.

We claim that these quorum choices must satisfy the object's constraints on quorum intersection. 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 TM assembles a view for *req*, it reads the log from DM1, which is exactly the sublog m. DM1 is an initial quorum for *req* because it intersects the final quorums for every event in m. The TM then chooses the response *res*, which by assumption is legal for m, but illegal for l. □

Theorem 2-9 is an optimality result. It shows that a minimal set of constraints on quorum intersection corresponds exactly to a minimal dependency relation for the type.

2.6 Examples

In this section we examine some examples of replicated objects. Our derivation of the constraints on quorum intersection is informal. A formal derivation would require introducing a formal specification method with proof rules strong enough to demonstrate

the properties discussed in the previous section. Working out the details of such a method lies beyond the scope of this thesis. These examples focus on a number of type-specific optimizations for reducing the sizes of logs and messages.

2.6.1 Files

Our previous discussions have employed a rather 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 *Paged_File* type provides the following operations:

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

overwrites the page with the given index, and

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

returns the current value of the page with a given index. Both *Read_Page* and *Write_Page* 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 (string)

The view for a *Read_Page* request must include the previous *Append* events, to detect whether the page exists, as well as all *Write_Page* events for that page, to determine the

page's current value. Consequently, a *Read_Page* request depends on *Append* events and on *Write_Page* events for that same page. The view for a *Write_Page* request need include only previous *Append* events. Consequently, a *Write_Page* request depends only on *Append* events. *Size* requests depend only on *Append* events because *Append* is the only operation that affects the size of the table.

Figure 5a 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 *Write_Page*(*i*,*) and *Read_Page*(*i*) respectively denote the class of normal *Write_Page* and *Read_Page* events for the page with index *i*. Figure 5b shows the range of quorum choices for a paged file replicated among five DM's based on the assumption that the quorums for *Read_Page* and *Write_Page* do not depend on the page affected. Note that *Append* always requires a majority of DM's, while *Size* requires only a minority. The view assembled to execute a *Read_Page* request need not contain the complete state of the file; it need only contain enough information to determine the current state of that page. Instead of keeping logs at DM's, an efficient implementation of the *Paged_File* type might keep a timestamped version of each page.

The *Paged_File* could be generalized to allow arbitrary sections of the file to be overwritten. 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.

2.6.2 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 DM that is part of a replicated queue implementation need retain only the

Fig. 5. Paged Files

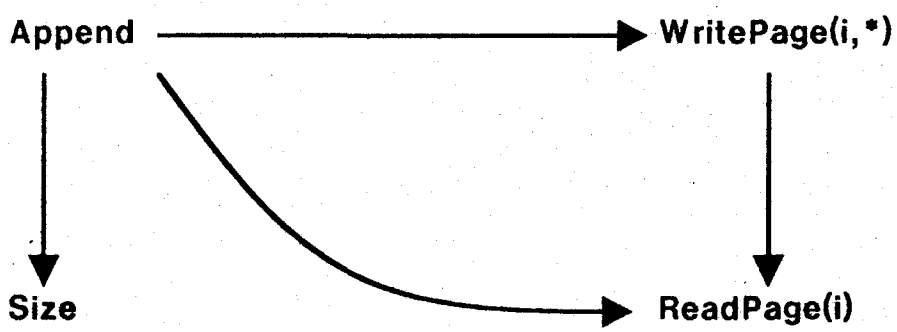

Figure 5a: Quorum Intersection Graph

Read_Page	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)
Write_Page	(1,5)	(2,4)	(3,3)	(2,2)	(1,1)
Append	(0,5)	(0,4)	(0,3)	(0,4)	(0,5)
Size	(1,0)	(2,0)	(3,0)	(2,0)	(1,0)

Figure 5b: Minimal quorum choices for five identical DM's.

---

following information:

- The *Enq* timestamp of the most recently dequeued item. This timestamp is called the DM's *horizon* timestamp.
- The *Enq* entries whose timestamps are later than the horizon timestamp.

The DM does not store *Deq* entries or *Enq* entries for items known to be dequeued.

An item is dequeued in the following way. The TM reads the horizon timestamps and the *Enq* entries from an initial *Deq* quorum. (The size of these messages may be reduced using a cache as described below). 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 indicates the item to be dequeued, and thus the new horizon time. The operation is complete when the TM installs the new horizon time at a final *Deq* quorum.

To illustrate the technique, we re-examine the three-site replicated queue example given in Chapter Two. As before, we start with a queue where *x* has been enqueued at two sites.

DM1	DM2	DM3
horizon: 0	horizon: 0	horizon: 0
t1: [Enq(x);Ok()]	t1: [Enq(x);Ok()]	

When *x* is dequeued, the horizon time for DM2 and DM3 is set to t1, and DM2 discards the *Enq* entry for *x*.

DM1	DM2	DM3
horizon: 0	horizon: t1	horizon: t1
t1: [Enq(x);Ok()]		

Item *y* is enqueued at DM1 and DM2, and *z* is enqueued at DM1 and DM3.

DM1	DM2	DM3
horizon: 0	horizon: t1	horizon: t1
t1: [Enq(x);Ok()]		
t2: [Enq(y);Ok()]	t2: [Enq(y);Ok()]	
t3: [Enq(z);Ok()]		t3: [Enq(z);Ok()]

When *y* is dequeued, the horizon time is set forward to t2 at DM1 and DM3, and DM1 discards the *Enq* entries for *x* and *y*.

DM1	DM2	DM3
horizon: t2	horizon: t1	horizon: t2
	t2: [Enq(y);Ok()]	
t3: [Enq(z);Ok()]		t3: [Enq(z);Ok()]

As a complementary compaction technique, DM's can periodically broadcast their horizon times. A DM 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 can also be reduced if each TM maintains a cache that is updated whenever it executes a *Deq*. To execute a *Deq*, the TM sends its horizon timestamp and the timestamp for the most recent encached *Enq* entry to each DM in the initial quorum. All DM's with earlier horizon times advance their own times, discarding *Enq* entries with earlier timestamps. Each DM responds with its own horizon time, and with all *Enq* entries whose timestamps are later than the *Enq* timestamp sent by the TM.

2.6.3 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* requests depend on prior *Insert* and *Delete* events. *Delete* requests also depend on prior *Insert* events and prior *Delete* events. Because *Change* signals if the key is absent from the table, *Change* requests 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 TM's 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 6a. The vertex marked *Insert*(k,*)/*Delete*(k) represents the class of normal *Insert* and *Delete* events for the key value *k*. The edge from *Insert*(*k*,*)/*Delete*(k) to *Lookup*(k) indicates that *Lookup* requests depend on prior *Insert* and *Delete* events for the same key value. To avoid

cluttering the diagram, events that terminate with exceptions are not shown.

The range of quorum choices for five identical DM's is shown in Figure 6b based on the assumption that quorums do not depend on key values. The dependency relation for *Table* is similar to the dependency relation for the *Paged_File* type.

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

---

Fig. 6. Tables

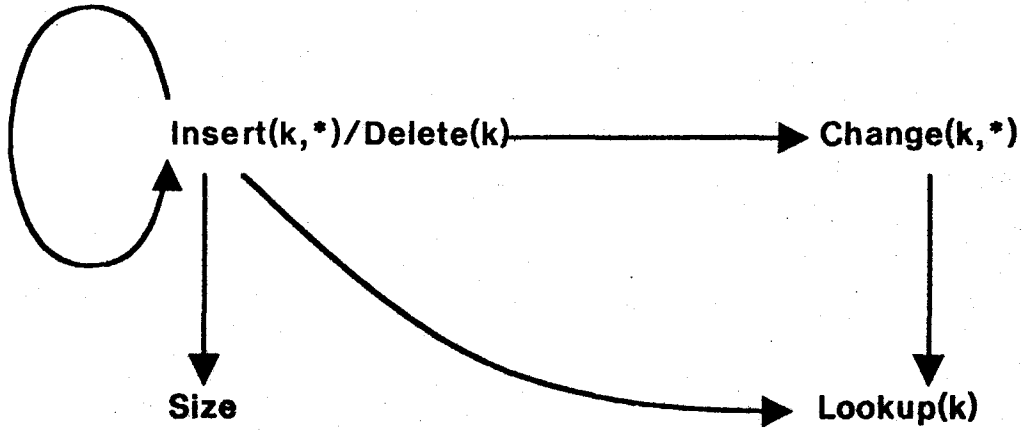

Figure 6a: Quorum Intersection Graph

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

Figure 6b: Minimal quorum choices for five identical DM's.

table at a DM 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 DM's table is implicitly bound to a *delete* entry with a timestamp of zero, which is older than any timestamp generated by a normal transaction.

Message sizes can be reduced if a TM executing an operation requests only the entry for a particular key. If each TM maintains a cache, it can include the timestamp for the key's cached entry with each request. Each DM 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 TM is later than the DM's timestamp, then the DM's entry is out of date and may be discarded. Out-of-date entries can also be detected if background tasks at DM's periodically broadcast the timestamp associated with a particular key. Once a DM has discarded an entry for a key, it responds to later requests 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 DM has an out-of-date *Insert* entry for that key. If a DM unilaterally discards an entry for a deleted key, a later *Lookup* request for that 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 DM's. One approach to this problem is a technique proposed by Daniels and Spector [Daniels 83] in which the table representation is rendered more compact by ordering the entries and merging adjacent entries for deleted bindings.

A complementary approach is to use long-running low-priority background transactions to

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

2.6.4 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 existing name servers that use replication, such as Grapevine [Birrel 81] and Clearinghouse [Oppen 81], this trade-off is considered to be unacceptable. Instead, a key/item pair is added or deleted at a single DM, and the update is subsequently propagated to the other DM's. This approach has the advantage that updates are faster and more likely to succeed, but it has the disadvantage that the behavior of the server becomes considerably more complex because clients may observe transient "inconsistencies" in the table, and concurrent conflicting updates must somehow be resolved.

There are two ways to view the "transient inconsistencies" that arise in systems such as Clearinghouse and Grapevine. One view is that this approach sacrifices atomicity 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 Clearinghouse and Grapevine, we prefer the second view for its economy of mechanism, as we may still use serializability to characterize the properties upon which the programmer may rely.

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*. A request of *Insert*(*k*,*i*) adds the item *i* to the multiset associated with *k*, and a request 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 interesting attribute of the *SemiTable* type is that no event depends on any other event; thus no quorums are required to intersect, and its quorum intersection graph has no edges. 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 DM's, 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* request 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. This is effectively the technique employed in Grapevine and Clearinghouse, as well as in an early replication proposal by Johnson and Thomas [Johnson 75].

2.6.5 The Double_Buffer Type

Files, Queues, Tables, and SemiTables each have a unique minimal dependency relation. We now consider a data type that has two distinct minimal dependency relations, neither of which is a subset of the other. An object of type *Double_Buffer* 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 *Double_Buffer* type provides three operations:

Produce = operation(item)

copies an item into the producer buffer, overwriting any previous contents,

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.

This type supports two distinct dependency relations. The alternatives arise because *Produce* events affect subsequent *Consume* events only if there has been an intermediate *Transfer*. Consequently, *Produce* entries can appear in the view constructed for a *Consume* request 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 dependency relation, and then one chooses the actual quorums subject to the constraints imposed by the dependency relation.

In the first relation, illustrated in Figure 7, *Consume* requests depend on both *Transfer* and *Produce* events. This situation is 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 8, *Consume* requests depend on *Transfer* events, and *Transfer* requests depend on *Produce* events. This situation is analogous to a mail program in which a request to deliver a message results in immediate transmission.

It is interesting to note that neither dependency relation is strictly better than the other from

Fig. 7. First Dependency Relation for Double_Buffer

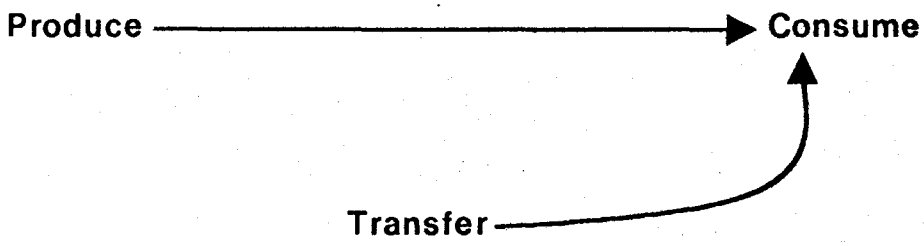

Figure 7a: the Quorum Intersection Graph.

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 7b: Minimal quorum choices for five identical DM's.

---

the point of view of quorum size. For example, comparing the right-hand columns of Figures 7b and 8b, we see that the first relation has a minimal quorum choice in which *Consume*, *Transfer*, and *Produce* quorums respectively consist of any five, one, and one DM's, while the same quorums for the second relation consist of any five, five, and one DM's. In this instance, the first dependency relation is clearly preferable. If we compare the left-hand columns of Figures 7b and 8b, however, we see that the first relation has a minimal quorum choice in which *Consume*, *Transfer*, and *Produce* quorums respectively consist of any one, five, and five DM's, while the same quorums for the second relation consist of any one, five, and one DM's. In this instance, the second dependency relation is clearly preferable.

The *Double_Buffer* type illustrates why our replication method requires that the entire view used to generate an entry must be written out with that entry to a final quorum. Consider the dependency relation in which *Consume* depends on *Transfer*, and *Transfer* depends on

Fig. 8. Second Dependency Relation for Double_Buffer

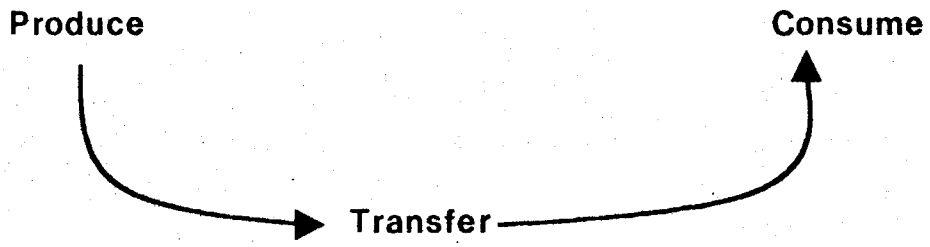

Figure 8a: the Quorum Intersection Graph.

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)

Figure 8b: Minimal quorum choices for five identical DM's.

Produce, but *Consume* does not depend directly on *Produce*. Any view constructed for *Transfer* contains the most recent *Produce* entry, and any view constructed for *Consume* contains the most recent *Transfer*. Because a TM executing a *Transfer* records both the *Transfer* entry and its view at a final quorum, the view for a later *Consume* request will also include the *Produce* entry needed to ascertain the item to be returned. If our replication method did not write out the entire view, then *Consume* would be forced to depend directly on *Produce*, unnecessarily restricting the range of quorum choices.

2.7 Multiple Levels of Replication

Some interesting sets of quorums can be realized by constructing replicated objects from other replicated objects. For example, by applying the quorum consensus method recursively, a DM for a replicated object could itself be a replicated object. One use for multiple levels of replication is an optimization we call *opportunistic* quorum choice. Because the constraints on quorum choice imposed by dependency relations reflect a worst-case analysis, it is sometimes possible to take advantage of fortuitously acquired information to recognize when a full quorum is not needed.

For example, consider a table used to store information about two classes of keys: say, East and West coast employees, where the quorums for operations involving distinct classes are disjoint. The table has an *Empty* operation that tests whether it is empty. Because *Empty* depends on all *Insert* events, each initial quorum for *Empty* must intersect the final quorums for *Insert* events for both classes of employees. This restriction is stronger than necessary, however, because a TM that observes the existence of an East coast employee need not attempt to observe the West coast employees, and vice-versa. If there are no East coast employees, however, then the TM must look for West Coast employees. In short, the TM must read from the entire initial quorum if the table actually is empty, but it may suffice to read from fewer DM's if the Table is not empty.

This behavior can be implemented by keeping East and West coast employees in subsidiary replicated tables. The TM executing the *Empty* event makes concurrent calls to each component. If one call returns *False*, then the table is empty, and the TM can return a result to the client whether or not the other call is successful. If one call returns *True*, the TM can return a result only if the other call succeeds. Although this protocol employs two message exchanges, one for each component, these exchanges can be executed concurrently.

2.8 Discussion

This chapter has introduced *General Quorum Consensus*, a systematic method for constructing replicated implementations of objects of arbitrary type. Previous work on quorum consensus methods has focused on particular types, such as files [Gifford 79], volumes [Gifford 82], and directories [Daniels 83]. By contrast, General Quorum Consensus is applicable to arbitrary types; the methods cited above may be considered specially optimized instances of our method.

We have shown that a necessary and sufficient condition for General Quorum Consensus to be correct is that the quorum intersection relation be a dependency relation. The problem of choosing quorums is thus reduced to the algebraic problem of identifying the minimal dependency relations for the type in question. The type's minimal dependency relations define fundamental limitations on the kinds of availability that can be achieved by our quorum consensus method. No amount of additional optimization can further reduce the set of required quorum intersections (although such optimizations can certainly affect other aspects of performance).

These results can be used to identify certain basic trade-offs. If one event depends on another, their degrees of availability must be inversely related. Because their quorums must intersect, the quorum for one can be made smaller only if the quorum for the other is made correspondingly larger. For example, in the replicated *Table*, if the attempts to look up items in the table are made faster and more likely to succeed, then attempts to modify the table must become slower and less likely to succeed.

Such trade-offs can be circumvented only if one can somehow change the rules. One way to change the rules is to choose a type with fewer dependencies, such as substituting the non-deterministic *SemiTable* type discussed above in place of the deterministic *Table* type. A comparison of the *Table* and *SemiTable* types illustrates a second basic trade-off between

non-determinism and flexibility in quorum choice. The less constrained an object's behavior, the less constrained its quorum choices. (A similar trade-off is noted in [Weihl 83] between non-determinism and concurrency.) Of course, a non-deterministic type is only appropriate for certain applications. A benefit of General Quorum Consensus is that it supports non-deterministic types, making this trade-off explicit. (Another way to change the rules is described in Chapter Four, where we discuss reconfiguration.)

An important step in the development of General Quorum Consensus was the realization that quorum choice and efficient storage management should be treated independently. A replicated object should be designed in two steps: quorums are chosen on the basis of a straightforward log-based scheme, and log compaction techniques are then applied to reduce storage consumption. Related replication methods are complicated by the fact that fault-tolerance and storage management are considered together, and it is not always easy to tell which purpose a particular feature serves. One result of our work is the conclusion that many aspects of these methods are best understood as optimizations that reduce storage consumption without affecting fault-tolerance. We have not tried to present a general log compaction method, because effective log compaction techniques seem to be quite type-specific, and because there is typically a range of techniques available for any given type, ranging from obvious to subtle, and from cheap to expensive.

Our method employs two technical innovations: the use of logs instead of versions, and timestamps instead of version numbers as the basis for replication. Gifford's original scheme uses quorum intersection for two distinct purposes: to compute responses to requests, and to order events. The introduction of timestamps eliminates the need to use quorum intersection to order events, resulting in fewer constraints on quorum choices. In the replicated queue example, pairs of *Enq* quorums must intersect if version numbers are used, but not if timestamps are used. The suggestion that timestamps could be used in place of version numbers is originally due to Gifford [Gifford 82]; we have worked out the consequences of this suggestion in detail.

We have argued that logs rather than versions should be the basic unit of replication. A technical problem with versions is that one cannot update a version unless it is already up to date. In the replicated queue implementation using versions, we saw that the *Enq* operation must locate and read an up-to-date version of the queue, create a new version that includes the new item, and write out the modified version. By contrast, in the implementation based on logs, the *Enq* operation does not need to observe the queue's current state; it suffices to send a timestamped entry containing the new item to a final quorum of DM's. The difference is that partially correct logs can be merged, while there is no analogous way to reconcile partially up-to-date versions. Because a version of an object represents a prefix of its current history, while a log represents an arbitrary subhistory, logs can be viewed as generalizing the notion of versions.

Like Gifford's method, General Quorum Consensus requires an underlying transaction mechanism, but it is independent of how transactions are implemented. Of course, the level of concurrency provided by a replicated object does depend on the underlying transaction system. In the next chapter, we discuss techniques for implementing highly concurrent replicated objects.

We have shown how to establish a set of correct and optimal constraints on quorum choice for objects of arbitrary type, but we have not addressed the question of deciding which quorum choice best suits the needs of a particular application. In fact, a strength of our method is that probabilistic analyses of availability can be conducted independently of the derivation of quorum constraints. The availability of an object's DM's can be modeled as a *stochastic process* [Cox 65] in which transitions between available and unavailable states are governed by probabilistic laws. One can use quorum information together with a stochastic model of the underlying hardware to compute *figures of merit*, such as the mean time until a particular operation becomes unavailable, or the likelihood that a particular operation will remain available for a fixed duration.

Chapter Three

Concurrency Control

3.1 Introduction

In the previous chapter we were able to derive constraints on quorum choice assuming only that transactions are atomic and serializable in timestamp order. The advantage of considering replication and concurrency control independently is that the basic replication method is easier to understand. The corresponding disadvantage is that it may be difficult to provide adequate concurrency for certain applications. In this chapter, we investigate concurrency control techniques for replicated objects. Concurrency control for replicated objects is more difficult than concurrency control for objects that reside at a single location because the information used to make scheduling decisions is itself replicated, and care must be taken that it is managed properly.

We use the following criteria as a basis for evaluating concurrency control mechanisms.

1. The level of concurrency supported.
2. The level of availability supported (i.e. the constraints imposed on quorum choice).
3. The amount of message traffic required.
4. The complexity of the local computations needed to make scheduling decisions.

Perhaps the simplest way to implement atomicity is to treat logs as files having *Read* and *Write* operations synchronized either by a locking scheme [Eswaren 76, Moss 81] or by a multi-version timestamp scheme [Reed 78, Reed 83]. In this chapter we propose two alternative approaches that provide a higher level of concurrency by integrating the concurrency control method with the replication method.

The simpler approach is called the *local scheme*, because scheduling decisions are made by individual DM's on the basis of local information. When a TM executing an operation attempts to observe or update the log at a particular DM, that DM may decide to accept, postpone, or refuse the request. The operation execution proceeds as soon as a quorum of DM's has accepted the request. The DM makes scheduling decisions on the basis of a predefined set of *conflicts* between pairs of events, accepting a request on behalf of a transaction only if it has not already accepted any conflicting requests from other uncommitted transactions. Because scheduling decisions are made by consulting a local table of conflicts, the local scheme requires no message traffic and hence does not reduce availability. The local scheme is optimal in the following sense: no other scheme in which decisions are based entirely on predefined conflicts can provide a higher level of concurrency. Nevertheless, the local scheme has some disadvantages. Because the log at each DM typically represents only a fragment of the object's current history, any scheme based on local information cannot take full advantage of information about the object's state, and is therefore inherently limited in the level of concurrency it can support [Bloom 75]. For the same reason, the local scheme does not provide adequate support for partial operations, which are operations that cannot be scheduled in certain states.

The shortcomings of the local scheme are remedied by a more complex scheme, called the *non-local* scheme, in which scheduling decisions are made by TM's on the basis of information collected from multiple DM's. When a TM attempts to execute an operation, it first collects information from multiple DM's, and then decides whether the operation can be scheduled. If not, the TM must try again after the object's state has changed. Each TM makes its own scheduling decisions, and short-term mutual exclusion locks are used to synchronize decision-making by different TM's. The non-local scheme is potentially more expensive in terms of message traffic and reduced availability, but it can support a higher level of concurrency than the local scheme because scheduling decisions take advantage of more information. The non-local scheme also provides better support for partial operations.

The non-local scheme can implement any concurrency control scheme that preserves serializability.

Section 3.2 describes our model of computation, Section 3.3 and 3.4 discuss the local and non-local schemes for a transaction system in which serialization ordering is determined dynamically, and Section 3.5 discusses concurrency control for schemes in which serialization ordering is statically predetermined. Section 3.6 contains a discussion of our results, and Section 3.7 discusses some related work.

3.2 Atomicity

Objects that support atomicity are referred to as *atomic objects*. From the point of view of a single transaction, the state of an atomic object reflects the effects of a sequence of prior transactions. From the same point of view, a *system* consists of a collection of atomic objects whose states all reflect the effects of the same sequence of prior transactions. The implementor of an atomic object is free to permit concurrency among transactions as long as serializability is preserved globally. The level of concurrency to support is an engineering consideration; the implementor must balance the need for concurrency against the need to keep the object's implementation simple and efficient. Our model for atomic objects is based on that of Weihl [Weihl 83a, Weihl 84]; our treatment is less formal, and we have changed details to facilitate the discussion of replicated objects.

We first discuss a model for individual atomic objects. Following [Weihl 84], an atomic object is defined by two specifications: its *serial specification* describes its behavior in the absence of concurrency and failures, while its *behavioral specification* describes the level of concurrency it supports.

A serial event is a paired request and response, a serial history consists of a sequence of serial events, and a serial specification defines a set of *legal* serial histories. The notion of

specification employed in the previous chapter corresponds to the notion of *serial* specification as used in this chapter. We will use lower-case letters to denote serial histories (g, h).

Like its serial specification, an object's behavioral specification consists of a set of histories that correspond to legal computations. We will use upper-case letters to denote behavioral histories (G, H). Behavioral histories encompass two kinds of events:

1. *Operation executions:* The event `[req; res; A]` denotes the execution of request *req* followed by response *res* on behalf of transaction **A**.
2. *Transaction events:* The event `[Commit A]` denotes the successful completion of transaction **A**, and `[Abort A]` its unsuccessful completion.

The intuitive interpretation of a behavioral history is that the ordering of operation executions reflects the order in which the object chooses the responses to invocations (but not necessarily the order in which the object received the invocations).

We assume that all behavioral histories satisfy the following well-formedness properties.

1. No transaction executes an operation at an object once it has committed.
2. No transaction both commits and aborts.

We say that a transaction **A** has committed *with respect to* transaction **B** if either **A** has committed or **A** is the same as **B**.

For example, the following is a behavioral history for an atomic FIFO queue, where the events of transaction **B** are indented to improve readability.

```
[Enq(x);Ok();A]
                                    [Enq(y);Ok();B]
                                    [Deq();Ok(y);B]
                                    [Commit B]
[Commit A]
```

Transaction **A** enqueues item *x*, **B** enqueues item *y* and then dequeues *y*, **B** commits and **A** commits. Note that events associated with distinct transactions are interleaved.

The atomic object should appear as if the committed transactions that manipulated it were executed in some serial order. Let > be a total order on committed and uncompleted transactions. Let *H* be a behavioral history for an atomic object. The *serialization* of *H* in the order > is the serial history *h* constructed in the following steps:

- Discard all events associated with aborted transactions.
- Reorder the events so that if **A** > **B** then all events of **A** follow all events of **B**, but the ordering of events within a single transaction is unaffected.
- Discard all *Commit* events, as well as all transaction identifiers in events.

The result is a serial history *h* consisting entirely of operation executions, with no mention of transactions. The history *H* is said to be *serializable in the order* > if the serialization *h* is a legal serial history (i.e. is included in the object's serial specification). The history *H* is *serializable* if there exists an ordering > such that *H* is serializable in the order >. The history *H* is *atomic* if the subhistory of *H* consisting of events associated with committed transactions is serializable.

For example, consider the behavioral history given above. If **A** > **B**, then the resulting serialization is:

```
[Enq(y);Ok()]
[Deq();Ok(y)]
[Enq(x);Ok()]
```

which is a legal serial history for the queue. If **B** > **A**, however, then the resulting serialization is:

```
[Enq(x);Ok()]
[Enq(y);Ok()]
[Deq();Ok(y)]
```

which is not a legal serial history because the wrong item has been dequeued.

Our analysis is concerned with pessimistic rather than optimistic methods for concurrency control. Under *optimistic* methods, transactions are allowed to violate serializability, but potential violations are detected when the transaction attempts to commit [Kung 81]. Under *pessimistic* methods, transactions are not allowed to proceed until it can be guaranteed that they will not violate serializability.

We make the additional assumption that the concurrency control mechanism cannot make scheduling decisions on the basis of assumptions about the transaction's future behavior. In particular, a transaction can always commit after executing an event. An *outcome* of a behavioral history *H* is the behavioral history constructed by committing some subset of the uncommitted transactions in *H*. A behavioral history *H* is *on-line atomic* if all its outcomes are atomic. For example, the following behavioral history is not on-line atomic, because the history that results if transaction **A** commits is not atomic.

```
                                        [Enq(y);Ok();B]
                                        [Deq();Ok(y);B]
                                        [Commit B]
[Deq();Ok(y);A]
```

So far we have discussed behavioral histories for individual objects. A behavioral history for a *system* consists of interleaved operation executions for multiple objects. For example, the following history involves transactions **A** and **B**, and queues *q1* and *q2*.

```
q1: [Enq(x);Ok();A]
                                        q2: [Enq(y);Ok();B]
                                        q2: [Commit B]
q2: [Deq();Ok(y);A]
q1: [Commit A]
q2: [Commit A]
```

In this history, **A** enqueues *x* at *q1*, **B** enqueues *y* at *q2* and commits, and **A** dequeues *y* from *q2* and commits. A behavioral history for a system is *serializable in the order* > if the history of each component object is serializable in the order >. It is *serializable* if there exists an order > such that each object's history is serializable in the order >; and it is *atomic* if the subhistory associated with committed transactions is serializable. The history shown above is serializable (only) in the order **A** > **B**, and is thus serializable and atomic.

Note that for a system to be atomic, it is not enough to require that each atomic object's history be atomic, because histories for different objects may be serializable in incompatible orders. For example, consider the following history:

```
q1: [Enq(x);Ok();A]
                                    q2: [Enq(y);Ok();B]
q2: [Deq();Ok(y);A]
                                    q1: [Deq();Ok(x);B]
q1: [Commit A]
                                    q1: [Commit B]
q2: [Commit A]
                                    q2: [Commit B]
```

Here, *q1*'s history is serializable only if **A** > **B**, while *q2*'s history is serializable only if **B** > **A**. Although each individual object's history is atomic, the history for the system as a whole is not atomic. To remedy this problem, it is necessary to ensure that all atomic objects are serializable with respect to the same order. Weihl [Weihl 83a, Weihl 84] has identified constraints on histories that ensure atomicity and that are both *local* and *optimal*. A constraint is *local* if the behavior of the system as a whole is atomic given that each object's histories satisfy the constraint, and a constraint is *optimal* if no weaker local constraint ensures atomicity for the system. In this chapter, we consider two of constraints discussed by Weihl: hybrid atomicity and static atomicity. For both of these constraints, transactions are issued timestamps, and are constrained to be serializable in timestamp order.

Under *hybrid atomicity*, uncommitted transactions synchronize dynamically on the basis of conflicts that arise over shared data. Transactions are issued timestamps as they commit. The synchronization method and the timestamp generation method must be compatible in the sense that each object's history must be serializable in timestamp order. In this chapter, we will consider synchronization methods in which transactions choose timestamps using *Lamport clocks* [Lamport 78]. Each site keeps a logical clock whose value is incremented whenever a timestamp is generated. Every message contains the sender's current clock value, and when a site receives a message containing a later clock value, it advances its own clock to a value later than the value it has observed.

This technique for choosing timestamps interacts with scheduling decisions in the following way: if a conflict arising over shared data causes one transaction to be delayed until another commits, then the delayed transaction will choose a later timestamp. For example, if transactions synchronize through two-phase locking [Eswaren 76, Moss 81], transactions will be given a timestamp order consistent with the order of lock transfers. The technique of assigning timestamps to transactions as they commit has been employed for database synchronization [Dubourdieu 82, Chan 82].

The scheme we have described so far does not guarantee external consistency because an observer outside the system may notice that ordering of events imposed by transaction timestamps may differ from the ordering observed in "real time." For example, one client of a FIFO queue may enqueue an item *x* and commit, and some time later another client may enqueue an item *y* and commit, yet if the first client is given a later timestamp than the second, *y* will be dequeued before *x*. The effects of these two transactions are serializable, but their serialization order may be unexpected. External consistency for events that occur sufficiently far apart in time can be guaranteed if timestamps are generated using approximately synchronized clocks [Lamport 78].

An alternative to hybrid atomicity is *static atomicity*, in which each transaction is issued a timestamp when it is created. Computations are constrained to ensure that transactions remain serializable in timestamp order. This approach encompasses Reed's multi-version timestamp scheme [Reed 78, Reed 83].

In Section 3.3 and Section 3.4 we will discuss concurrency control methods for replicated objects based on hybrid atomicity. Static atomicity is discussed in Section 3.5.

3.3 Local Concurrency Control

In this section, we present a local concurrency control method for replicated objects employing hybrid atomicity. All scheduling is done at DM's on the basis of local information, and decisions are made on the basis of a predefined set of conflicts between events. As before, a TM executes an operation in four steps: (i) it merges the logs from an initial quorum of DM's, (ii) it conducts a local computation, (iii) it records the results at a final quorum of DM's, and (iv) it returns the results to the client. When a DM receives a request from a TM, it compares the new request with the requests it has already accepted from other uncommitted transactions. If the new request conflicts with an accepted request, the new request is postponed until the transactions that have executed conflicting requests have committed or aborted. The replication method ensures that an operation may be scheduled safely whenever a quorum of DM's have agreed to accept the request.

The local scheme is subject to deadlock. Some deadlocks can be avoided if TM's communicate with DM's in a canonical order. This approach will avoid certain deadlocks arising within a replicated object, but it cannot avoid deadlocks that span multiple objects. Deadlock avoidance also increases delay, because messages must be sent sequentially rather than concurrently. Another approach is to detect deadlocks after they have arisen, aborting one of the deadlocked transactions to permit the others to proceed. In the scheme described in this section, there is no difficulty constructing a "waits-for" graph employed by many distributed deadlock detection algorithms (see [Kohler 81] for a survey of such algorithms). A third approach is simply to restart transactions that do not appear to be making progress.

3.3.1 Organization

Each DM stores a log of entries, which are timestamped records of events. Unlike the logs considered in the previous chapter, these logs represent behavioral histories, and therefore they include interleaved events of different transactions as well as *Commit* and *Abort* entries. (As a simple optimization, DM's can discard all entries associated with aborted transactions.) As before, timestamps are used to order log entries, permitting logs at different DM's to be merged. Lamport clocks are used to generate timestamps.

Note that we are employing two kinds of timestamps here. The concurrency control method assigns a timestamp to each *transaction* as it commits, and our implementation assigns a timestamp to each *entry* that records an event. It is convenient to identify the timestamp assigned to a transaction with the timestamp assigned to the entry for its *Commit* event. Consequently, the serialization ordering of committed transactions can be reconstructed from the timestamp ordering of their *Commit* entries.

3.3.2 The Basic Local Method

Transactions synchronize by acquiring locks at DM's. Each transaction holds its locks until it commits or aborts. A transaction's locks may be broken in the event of a deadlock, crash, or communication failure, but that transaction must be aborted. There are two kinds of locks: an initial lock is associated with each request, and a final lock is associated with each event.

Before a DM sends its resident log to a TM, it must grant the transaction an *initial lock* for the request. The transaction is delayed until it acquires initial locks at an initial quorum of DM's for that request. Once the quorum of initial locks have been acquired, the TM chooses a response, appends an entry for the new event to the view, and sends the modified view to a final quorum of DM's. Before a DM merges the modified view with its resident log, it must

grant the transaction a *final lock* for the event. The transaction is delayed until it acquires final locks at a final quorum of *DM*'s for that event. A DM will grant a lock to a transaction only if no other transaction holds a *conflicting* lock at that DM. The correctness condition for choosing lock conflicts is given below.

The full replication method works as follows.

1. When a TM receives a request from a client, it forwards the request to an initial quorum of DM's.

2. As each DM grants initial lock for the request, it responds with the entries for transactions that have committed with respect to the requesting transaction.

3. When the TM has received the replies from an initial quorum, it merges the logs, constructs a view, and chooses a response. The TM generates a new timestamp, appends the new entry to the view, and sends the view to a final quorum of DM's.

4. As each DM grants a final lock for the event, it merges the updated view with its resident log. As soon as a final quorum of DM's has acknowledged the update, the TM returns the response to the client.

In this scheme, all scheduling decisions are made at DM's on the basis of locally resident information. Note that because the view consists entirely of events that have committed with respect to the transaction executing the operation, the view can be treated as a serial history instead of a behavioral history, and consequently TM's may be structured exactly as they were in the previous chapter.

All lock conflicts are between initial and final locks. Lock conflicts are defined by a *lock conflict relation* between requests and events. The correctness condition for a lock conflict relation is the following:

> *The intersection of the lock conflict and quorum intersection relations must be dependency relation.*

Recall that an object's dependency relations are defined by its serial specification. We

remark that it is necessary but not sufficient for both relations to be dependency relations, because the intersection of two dependency relations is not necessarily a dependency relation (e.g. the *Double_Buffer* data type in Chapter Two). In practice, however, both the lock conflict and quorum intersection relations will typically correspond to the same minimal dependency relation.

We now present an informal correctness argument for the local method. A more precise argument is given in the next section. We first observe that the correctness of the method is determined only by conflicts between requests and events whose respective initial and final quorums are required to intersect. If one transaction's initial quorum for the request does not intersect another transaction's final quorum for the event, then a lock conflict will not delay either transaction. On the other hand, if the quorums are required to intersect, then the transaction that acquires the first set of locks will delay the other, because the later transaction will encounter at least one lock conflict in every quorum. As discussed above, deadlocks are possible.

The initial locks guarantee that the response chosen by a transaction cannot be invalidated by another transaction. A transaction may acquire a quorum of initial locks for a request only when all other transactions that have executed events on which the request depends have committed. Once a quorum of initial locks has been acquired, any transaction attempting to execute an event on which the request depends will be unable to acquire a quorum of final locks until the conflicting initial locks are released.

The final locks guarantee that the transaction will not invalidate a response already chosen by another transaction. A transaction may acquire a quorum of final locks for an event only when all other transactions that have executed requests that depend on the event have committed. Once a quorum of final locks has been acquired, any transaction attempting to execute a request that depends on the event will be unable to acquire a quorum of initial locks until the conflicting final locks are released.

So far, we have described a scheme that works for single-level transactions. This scheme can be generalized to nested transactions using lock inheritance rules similar to those of Moss [Moss 81].

3.3.3 Discussion

It is instructive to compare the local scheme to a scheme in which access to logs is synchronized by two-phase locking [Eswaren 76, Moss 81]. The log at each DM has an associated lock which may be held in shared or exclusive mode. A transaction must acquire a shared lock before reading a log, and an exclusive lock before updating it. A transaction cannot acquire a shared or exclusive lock if another transaction has an exclusive lock. A transaction's locks are released when it commits or aborts. We will refer to this scheme as the *simple* scheme.

The two schemes are equivalent with respect to availability, message traffic, and the complexity of the local computations needed to make scheduling decisions. Constraints on availability are determined by the constraints on quorum intersection, and both schemes require only that the quorum intersection relation be a dependency relation. In both schemes, scheduling decisions do not require any message traffic because they are based on information local to each DM. In both schemes, the computations required for scheduling decisions can be carried simply by consulting a table of lock conflicts. On the other hand, these schemes are not equivalent with respect to the level of concurrency they support, as shown by the following examples.

For the file type, the simple locking scheme permits transactions to execute concurrent *Write* events only if they can locate disjoint quorums. Otherwise, the exclusive lock acquired by one transaction will delay the other. The local scheme permits transactions to execute concurrent *Write* events regardless of their quorums.

For the *Queue* type, the simple scheme permits concurrent transactions to execute *Enq*

events only if they can locate disjoint quorums, while the local scheme permits concurrent *Enq* events regardless of quorums.

For the *Table* type, the simple scheme allows concurrent transactions to modify the table only if they can locate disjoint quorums, while the local scheme permits concurrent modifications that involve distinct keys regardless of quorum choice. The *SemiTable* type imposes even fewer constraints, because locks events involving the same key need not conflict. (Nevertheless, it might be convenient to have *Insert*(*k*,*v*) conflict with *Lookup*(*k*) to increase the likelihood that *Lookup* will return the most recent binding for the key.)

In summary, the local scheme imposes the same costs as the simple scheme with respect to availability, message traffic, and local computation, but it provides more concurrency. In the next section, we show that the local scheme is an optimal conflict-based strategy: no concurrency control scheme based entirely on predefined conflicts can provide more concurrency.

One shortcoming of the local scheme is that it is inherently limited in the level of concurrency it can support. Although the local scheme is optimal for schemes based only on predefined conflicts, concurrency control methods that take advantage of state information can provide a higher level of concurrency. For example, consider a printer spooler implemented by a replicated FIFO queue. Files are enqueued by clients and dequeued by a printer controller. Suppose that a transaction **A** has enqueued a file *x* and committed, and that *x* has tentatively been dequeued by the printer controller transaction **B**.

```
[Enq(x);Ok();A]
[Commit A]
                                    [Deq();Ok(x);B]
```

Under the local scheme, a transaction **C** attempting to enqueue file *y* would be delayed until the printer controller's transaction commits, because the initial lock for *Deq* conflicts with

the final lock for *Enq*. Note, however, that it is not really necessary to delay **C**, because the serial history that results if **B** is serialized before **C** is:

```
[Enq(x);Ok()]
[Deq();Ok(x)]
[Enq(y);Ok()]
```

and if **C** is serialized before **B**:

```
[Enq(x);Ok()]
[Enq(y);Ok()]
[Deq();Ok(x)]
```

and both of these are legal histories. In the printer spooler application, it would be unsatisfactory to delay clients while the printer controller dequeues files. To recognize that the *Enq* may proceed in this instance, the concurrency control method must take the state of the queue into account, implying that scheduling decisions must employ state information collected from multiple DM's.

A second shortcoming of the local scheme is that it does not provide adequate support for *partial operations*, which are operations that cannot be executed in certain states. For example, consider a FIFO queue in which *Deq* invocations are legal only when the queue is non-empty. Partial operations do not seem to be very useful in the absence of concurrency; a partial *Deq* applied to an empty queue would never return. In the presence of concurrency, however, partial operations can be quite useful; a partial *Deq* operation applied to an empty queue would not be scheduled until an item had been placed in the queue. Such behavior might be useful for printer spooler application where the printer controller is willing to wait for files to appear in the queue.

Unfortunately, no local method can postpone *Deq* invocations when the queue is empty.

The problem is that individual DM's cannot ascertain the size of the queue on the basis of local information; thus it is impossible to recognize when a *Deq* invocation should be delayed. For example, consider a queue replicated among three DM's with *Enq* quorums of (0,2) and *Deq* quorums of (2,2). In the following situation DM1 observes the queue to be empty when it is not:

DM1	DM2	DM3
	t1: [Enq(x);Ok();A]	t1: [Enq(x);Ok();A]
	t2: [Commit A]	t2: [Commit A]

but in the following situation DM1 observes the queue to be non-empty when it is actually empty:

DM1	DM2	DM3
t1: [Enq(x);Ok();A]	t1: [Enq(x);Ok();A]	t1: [Enq(x);Ok();A]
t2: [Commit A]	t2: [Commit A]	t2: [Commit A]
	t3: [Deq();Ok(x);B]	t3: [Deq();Ok(x);B]
	t3: [Commit B]	t3: [Commit B]

The only way to determine whether the queue is empty is to observe the logs at an initial *Deq* quorum of DM's. Consequently, support for partial operations requires that at least some scheduling decisions be made in a non-local manner using information collected from multiple DM's.

3.3.4 Correctness Arguments

As in Chapter Two, our replication method uses logs to represent histories. A *serial log* is a map from timestamps to serial events, and a *behavioral log* is a map from timestamps to behavioral events. We use lower-case letters to denote serial logs (l, m), and upper-case letters to denote behavioral logs (L, M). A log is said to be hybrid atomic (resp. static atomic) if its associated history is hybrid atomic (resp. static atomic).

A replicated object is an automaton whose state is given by the locks and logs at each DM. The following constitute the object's atomic state transitions:

1. A DM grants an initial lock and sends its local log to a TM.
2. A DM grants a final lock and merges the view received from a TM with its local log.
3. A DM releases a transaction's initial and final locks, and appends a *Commit* or *Abort* entry to its local log.

Each transaction follows the protocol described above, but transitions associated with distinct transactions may be interleaved.

The *global log* associated with a computation contains all the events for which a complete set of initial and final locks have been acquired, as well as *Commit* and *Abort* events. The correctness condition we wish to show is that the global log is guaranteed to be a legal behavioral log if and only if the intersection of the lock conflict relation and the quorum intersection relation is a dependency relation.

We begin with some lemmas relating lock conflicts and serialization orders. Let $\succ$ denote the intersection of the lock conflict and quorum intersection relations, and let L be a global log for an object. Let e and e' be events of L respectively executed by transactions **A** and **B**. We say that a transaction is *active* if it has not committed or aborted.

Lemma 3-1: If **A** is active and $e \succ e'$, then **B** has committed (or aborted) with respect to **A**.

Proof: Otherwise if **B** is active, some DM in the intersection of the initial quorum for e and the final quorum for e' has granted conflicting locks. □

Lemma 3-2: If **A** is active and $e' \succ e$, then **B** has committed (or aborted) with respect to **A**.

Proof: Same as the previous lemma. □

We are now ready to demonstrate the basic correctness property.

Theorem 3-3: Suppose **A** acquires quorums of initial and final locks for e, causing the object's global log to undergo a transition from L to L'. We claim that if L is on-line hybrid atomic, so is L'.

Proof: Let the serial log $l_1 \circ e \circ l_2$ be the serialization in timestamp order of any outcome of L'. Because L is on-line hybrid atomic, $l_1 \circ l_2$ and hence l_1 are legal serial logs. Let v be the serialization of the view used to choose e. Because all events of v are committed with respect to **A**, the timestamp generation protocol ensures that v is a sublog of l_1. In fact, v is a closed sublog of l_1 containing the events on which e depends (Lemma 3-1), therefore $l_1 \circ e$ is legal by the definition of dependency. No events in l_2 depend on e (Lemma 3-2), and therefore $l_1 \circ e \circ l_2$ is legal. □

We have shown that atomicity is preserved if the intersection of the lock conflict and quorum intersection relations is a dependency relation. We now demonstrate the converse: the local scheme preserves legality only if the intersection of the lock conflict and quorum intersection relations is a dependency relation. We start with a lemma.

Lemma 3-4: Let $\succ$ be a relation between requests and events that is *not* a dependency relation; i.e. there exists an event e, and legal serial logs l and m such that m is a closed sublog of l containing all events e' such that $e \succ e'$, and $m \circ e$ is legal but $l \circ e$ is illegal. We claim that there exist m and l such that m is missing exactly one event of l.

Proof: Let $|l|$ denote the number of timestamps for which l is defined. Choose m and l so that $|l|-|m|$ is minimal. Consider the log m' constructed by restoring the earliest missing element e' of m. We first show that m' is legal. If $m' = m_1 \circ e' \circ m_2$, then $m_1 \circ e'$ is legal since it is a prefix of l. Because m is closed, no event in m_2 depends on e', thus $m_1 \circ e' \circ m_2$ is legal. We now claim that $m' = l$. Otherwise, if $m \circ e$ is legal, then $|l|-|m'| < |l|-|m|$, and if $m \circ e$ is illegal, then $|m'|-|m| < |l|-|m|$, and either case contradicts the assumption that $|l|-|m|$ is minimal. □

We are now ready to show that the correctness condition given above completely characterizes the set of correct conflict-based concurrency control methods.

Theorem 3-5: A lock conflict relation preserves atomicity only if its intersection with the quorum intersection relation is a dependency relation.

Proof: Let $\succ$ be the intersection of the lock conflict and quorum intersection relations. We assume that $\succ$ is not a dependency relation, and we construct a scenario in which an illegal log is generated. Let l, m, and e satisfy the properties of the previous theorem, and let e' be the event of l missing from m. Consider the following scenario. Transaction **A** executes the events of m_1 and commits. Transaction **B** executes e', and **C** executes m_2, followed by e. The event e' does not appear in any view constructed for **C**, because **B** has not committed. **C** is not delayed, because the locks for e' do not conflict with the locks for any event of **C**. If **B** and **C** commit, and **C** is given the later timestamp, the resulting log will be serialized as $m_1 \circ e' \circ m_2 \circ e$, which by assumption is not a legal serial log. □

The last theorem is an optimality result. It states that any minimal set of lock conflicts corresponds exactly to a minimal dependency relation. In a more general context, it states that the local scheme encompasses any concurrency control method in which predefined conflicts are used to delay operations as they attempt to observe or modify the object's state.

3.4 Non-Local Concurrency Control

We have seen that concurrency control methods based on predefined conflicts, although efficient, have two drawbacks: they are inherently limited in the level of concurrency they can support, and they provide inadequate support for partial operations. In this section we propose an alternative concurrency control method, called the *non-local scheme*, in which these drawbacks are remedied by allowing scheduling decisions to take advantage of state information. Because each DM typically contains only a fragment of the object's state, scheduling decisions are made by TM's on the basis of information collected from multiple DM's. The object's behavioral specification is used to derive a set of constraints on quorum

intersection. Although these constraints appear quite similar to those derived from the serial specification, the formal similarity hides some interesting differences that arise from the need to ensure that the TM observes enough of the object's state to make scheduling decisions as well as to choose a response. As before, we focus on concurrency control methods based on hybrid atomicity.

The non-local scheme introduces some new problems.

- *Low-level synchronization:* For example, how can we ensure that two TM's for an atomic FIFO queue do not simultaneously dequeue the same item?
- *Delay:* A TM may collect state information from an initial quorum only to discover that the client's request cannot be carried out until a conflicting transaction commits, aborts, or executes additional operations. How can the TM minimize the expense of (i) noticing when the desired state change occurs, and (ii) collecting new state information?
- *Quorum choice:* How do scheduling and quorum choice interact? We will see that deciding whether a request can be scheduled may require more information than simply deciding what the response should be. Consequently, atomic objects that support a high level of concurrency may place more constraints on quorum choice than the atomic objects that support a lower level of concurrency.

3.4.1 The Basic Method

We start with a summary description of how an operation invocation is carried out, and we then describe each step in detail.

1. When a TM receives a request from a client, the TM requests the logs from a collection of DM's. Each DM that responds grants the transaction a short-term *request lock* for that request. The logs are merged in timestamp order to construct a *view*.
2. If the view does not permit a response consistent with the object's behavioral specification, the TM releases its request locks at the DM's, waits for the object's state to change, and restarts the entire protocol. Otherwise a response is chosen.

3. The TM generates a new timestamp, appends the new entry to the view, and sends the updated view to a final quorum of DM's. Each DM accepts the view as soon as it has no conflicting request locks, merges the view with its resident logs, and returns an acknowledgment. As soon as the TM receives acknowledgments from a final quorum, all request locks are released.

4. The response is returned to the client.

We remark that concurrency control for centralized (non-replicated) objects can be viewed as a degenerate instance of this method in which all quorums intersect. Consequently, this method can be used to implemented any concurrency control strategy consistent with hybrid atomicity, although some strategies may impose additional constraints on quorum intersection.

3.4.2 Low-Level Synchronization

The purpose of low-level synchronization is to ensure that certain events are physically serialized. One approach is to associate short-term mutual exclusion locks with each DM; while one transaction holds the lock, no other transaction can observe or update the DM's log. This section describes an alternative scheme that supports more concurrency.

Low-level synchronization may be accomplished through short-term *request locks* associated with each DM. When a transaction reads a DM's log for a request *req*, the transaction is granted a request lock for *req* at that DM. While that transaction holds the lock, that DM will not accept any updates on behalf of an event on which *req* depends. For example, a DM for a queue will not accept requests to append *Enq* entries while another transaction holds a *Deq* request lock.

Unlike the locks in the local scheme, request locks are not held until a transaction commits. Instead, a request lock is held until either (i) a response has been chosen and recorded at a final quorum, or (ii) the TM has decided that no response is possible in the current state. While a transaction holds a request lock at an initial quorum, no other transaction can append an event on which the request depends; thus request locks have the effect of

serializing events that depend on one another. Request locks are subject to deadlock, and deadlock resolution methods such as avoidance, detection, and time-outs may be used. Because request locks are held for a brief duration, it is to be hoped that deadlocks arise infrequently. Request locks can be broken in the event of deadlocks, crashes, or communication failures, but the transaction holding the lock must be aborted.

3.4.3 Partial Operations

After a TM has constructed a view for a request, it may discover that there is no legal response to the request in the object's current state. This situation might arise because the underlying sequential type is partial (e.g. a queue with a partial *Deq*), or because the request must await the outcome of a concurrent transaction (e.g. attempting to read a file that has been written by an uncommitted transaction). In either case, the TM cannot return a response until the object's state changes.

A naive implementation of a partial operation, say a partial *Deq*, might employ a kind of busy-waiting. If a TM merges the logs from an initial quorum for a partial *Deq* only to discover the queue is empty, then it releases its request locks, waits for some duration, and tries again. If this approach is to provide adequate performance, some means must be found to minimize the resulting message traffic.

Message frequency can be reduced if DM's notify waiting TM's of the commit or abort of transactions that have executed events on which the blocked request depends. In the partial *Deq* example, if a TM is unable to dequeue an item, it can request each DM in its initial quorum to notify it when a transaction that has enqueued or dequeued an item commits or aborts. The quorum intersection relation ensures that the TM will be notified as soon as there is an item to be dequeued. Notification cannot rule out all unnecessary message traffic, but it can ensure that message traffic occurs only as a result of a state change. (The TM may still have to poll occasionally to ensure that it is not waiting for

notification from a failed DM).

Message sizes can be kept small if the TM maintains a cache between invocations, requesting only incremental changes from the DM's. When the TM releases the locks at its initial quorum, it can ask the DM to keep track of the events it subsequently receives. When the TM next attempts to construct a view, the DM's need only respond with the changes that have occurred in the meantime.

Because partial operations require more message traffic than total operations, this discussion might suggest that data types with total operations are inherently more efficient. Such a conclusion is misleading, however, as illustrated by the replicated printer spooler example. If the *Deq* operation called by the printer controller is partial, then the TM will poll the DM's for new items to be dequeued. If the *Deq* operation is total, then the printer controller would have to poll the queue; thus the level of message traffic is likely to be the same in both cases.

3.4.4 Quorum Choice

Just as for sequential objects, constraints on quorum choice for atomic objects are governed by a *dependency relation*. The formal definition of dependency for behavioral logs is the same as for serial logs. A relation $\succ$ is a dependency relation if whenever M is a legal sublog of L closed under dependency such that M contains all events of L on which an event e depends, then:

$$M \circ e \text{ is legal} \Rightarrow L \circ e \text{ is legal.}$$

The only difference between this definition and the one given in the previous chapter is that here we are dealing with behavioral logs, while before we were dealing with serial logs. Because we are dealing with behavioral logs, the view constructed for an invocation may include the events for uncommitted and aborted transactions.

Since it is always legal for an active transaction to commit or abort, *Commit* and *Abort* requests do not depend on any prior events. When does an operation invocation depend on a *Commit* or *Abort* event? We make the conservative assumption that if an event *e* of transaction **A** depends on an event of transaction **B**, then *e* depends on the commit or abort of **B**. It is clear that the converse holds: if no event of **A** depends on any event of **B**, then no event of **A** depends on the commit or abort of **B**. To ensure that every event that depends on an event of **B** observes whether **B** has committed or aborted, the system distributes *Commit* or *Abort* entries to each DM visited by **B**.

The interaction of concurrency and dependency can be illustrated by an informal analysis of an implementation of a highly concurrent replicated FIFO queue. An item in the queue is said to be *committed* with respect to a transaction **A** if the transaction that enqueued it has committed with respect to **A**. Transaction **A** may execute an *Enq* if and only if all items that have been dequeued are committed with respect to **A**. To see why this restriction holds, consider the following example. Suppose transaction **A** has enqueued and dequeued item *x*:

```
[Enq(x);Ok();A]
[Deq();Ok(x);A]
```

If **B** is allowed to enqueue an item *y*, then **B** might be serialized before **A**, resulting in the illegal serial history:

```
[Enq(y);Ok()]
[Enq(x);Ok()]
[Deq();Ok(x)]
```

By contrast, if all dequeued items have committed with respect to a transaction, then all *Enq* events executed by that transaction will appear after the *Enq* events for the dequeued items, preserving the queue's FIFO discipline. For example, consider the following behavioral

history:

```
[Enq(x);Ok();A]
[Commit A]
                                    [Deq();Ok(x);B]
```

If a third transaction **C** attempts to enqueue an item, it may proceed, because the (tentatively) dequeued item *x* is committed with respect to **C**, and both serialization orderings of **B** and **C**:

```
[Enq(x);Ok()]
[Enq(y);Ok()]
[Deq();Ok(x)]
```

and

```
[Enq(x);Ok()]
[Deq();Ok(x)]
[Enq(y);Ok()]
```

are legal serial histories.

How does this implementation compare to the local scheme? The local scheme provides poor support for a printer-spooler application because clients are delayed whenever the printer controller dequeues a file for printing, and the printer controller is delayed whenever a client enqueues a new file. In an implementation of the printer spooler based on the non-local scheme, clients are never delayed by the printer controller because the printer controller dequeues only files enqueued by committed client transactions, and the printer controller is delayed only when there is no committed file to dequeue.

What are the constraints on quorum intersections for the non-local implementation of the

queue? The important point here is that concurrency introduces new dependencies between events. For a serial queue, *Enq* events are always legal and always return the same response, thus *Enq* events need not depend on other events. For an atomic queue, however, we have seen that *Enq* events are not legal if an uncommitted item has been dequeued. Our (informal) analysis shows that *Enq* events for atomic queues depend on prior *Deq* events, but not on other *Enq* events. Consequently, each initial quorum for *Enq* must intersect each final quorum for *Deq*, a constraint that did not exist in the local scheme.

The quorum intersection graph for the atomic queue is shown in Figure 9a, and the minimal quorum choices for five DM's are shown in Figure 9b. Note that although the initial quorums for *Enq* events are larger for the non-local scheme than for the local scheme, the availability of *Enq* events is the same in both schemes, because a TM executing an *Enq* can use the same set of DM's for its initial and final quorums.

Fig. 9. An Atomic FIFO Queue

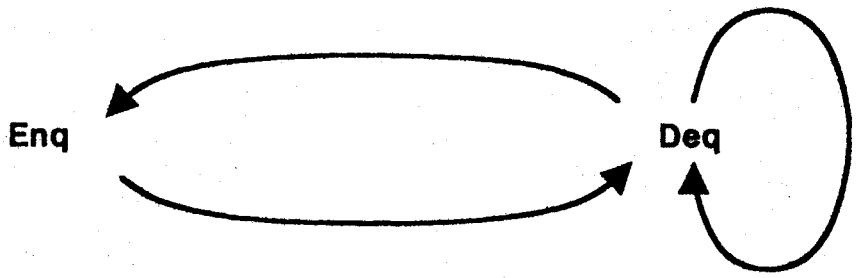

Figure 9a: The quorum intersection graph.

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

Figure 9b: Minimal quorum choices for five identical DM's.

As an example of a type for which an increase in concurrency does affect availability, let us consider three implementations of a data type representing a bank account. The *Account* type provides two operations: *Deposit*(*n*) increments the account balance by *n* dollars, and *Withdraw*(*n*) decrements the account balance by *n* dollars if the balance is greater than *n*, otherwise it signals an exception. We first consider an implementation of a bank account in which synchronization is based on the local scheme. We then consider a non-local scheme that increases concurrency without increasing overall quorum sizes. We finally consider another non-local scheme that further increases concurrency at the cost of imposing additional constraints on quorum choices.

The quorum intersection graph for a bank account based on the local scheme is shown in Figure 10a. *Withdraw* requests depend on *Deposit* events and on earlier *Withdraw* events, but *Deposit* requests do not depend on any other events because they return no information about the current account balance. The minimal quorum choices for five identical DM's are given in Figure 10b.

Now consider a situation where transaction **A** has successfully withdrawn ten dollars from an account, and transaction **B** wishes to deposit ten dollars. Using the local scheme, **B** would not be allowed to proceed, because the final lock for *Withdraw* conflicts with the initial lock for *Deposit*. In this situation however, **B** could be allowed to proceed, because every serialization of the following is a legal history:

```
[Deposit(10);Ok();A]
[Withdraw(10);Ok();A]
                                [Deposit(10);Ok();B]
```

By contrast, **B** could not be allowed to proceed if **A** had observed that the account balance was less than twenty dollars, because the history that results if **B** is serialized before **A** is illegal:

Fig. 10. The Conflict-Based Bank Account

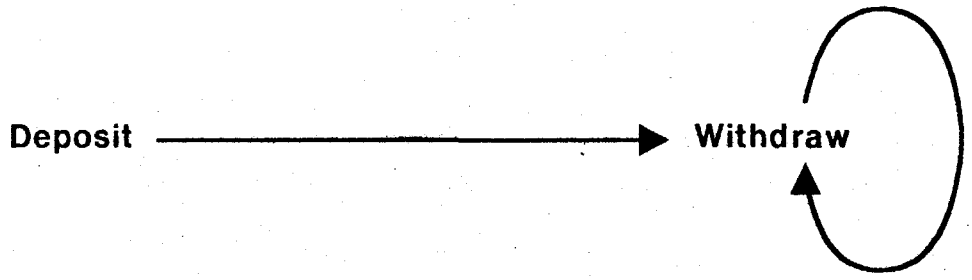

Figure 10a: The quorum intersection graph.

Deposit	(0,1)	(0,2)	(0,3)
Withdraw	(5,1)	(4,2)	(3,3)

Figure 10b: Minimal Quorum choices for five identical DM's.

```
[Deposit(10);Ok();A]
[Withdraw(20);Overdrawn();A]
                                [Deposit(10);Ok();B]
```

If B is allowed to execute a deposit in the first situation but not the second, then all *Withdraw* entries must appear in the view for *Deposit*, yielding the quorum intersection graph shown in Figure 11a.

As illustrated by Figure 11b, this additional constraint does not change the availability of any operations because the overall quorum sizes have not changed (each event can use the same set of DM's for both initial and final quorums). The additional constraint does affect performance, however, because *Deposit* now has non-empty initial quorums, and *Withdraw* has larger final quorums.

Fig. 11. First State-Based Bank Account

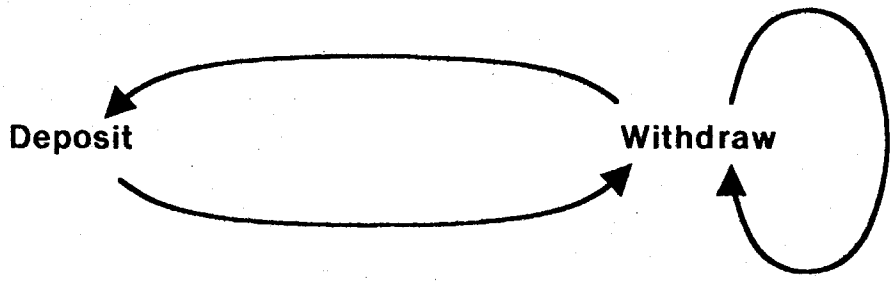

Figure 11a: The quorum intersection graph.

Deposit	(1,1)	(2,2)	(3,3)
Withdraw	(5,5)	(4,4)	(3,3)

Figure 11b: Minimal quorum choices for five identical DM's.

Even more concurrency is possible. Suppose that the account balance is zero, transaction **A** has observed that the account balance is less than twenty dollars, and transaction **B** attempts to deposit ten dollars. Transaction **B** could be allowed to proceed, because both serializations of the following history are legal:

```
[Withdraw(20);Overdrawn();A]
                              [Deposit(10);Ok();B]
```

On the other hand, if transaction **C** had previously deposited ten dollars, then **B** must be delayed, because the following history has an illegal serialization, i.e. if **B** and **C** are serialized before **A**.

```
[Withdraw(20);Overdrawn();A]
                        [Deposit(10);Ok();B]
                                              [Deposit(10);Ok();C]
```

Consequently, if the deposit is to take place in the first situation but not the second, then all prior *Deposit* entries must appear in the view for *Deposit*, yielding the quorum intersection graph shown in Figure 12a. This additional constraint affects both quorums and performance. As illustrated in Figure 12b, every pair of quorums for *Deposit* must now intersect.

Fig. 12. Second State-Based Bank Account

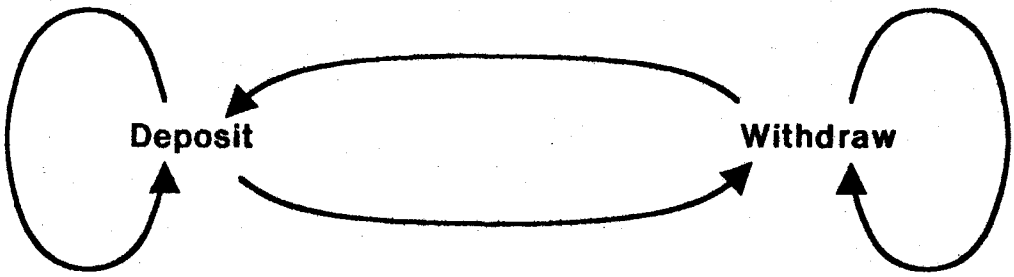

Figure 12a: The quorum intersection graph.

Deposit	(3,3)
Withdraw	(3,3)

Figure 12b: Minimal Quorum choices for five identical DM's.

3.5 Static Atomicity

Our discussion of local concurrency control methods has assumed that uncommitted transactions are serialized dynamically. In this section we discuss the implementation of replicated atomic objects in a model of computation in which transactions are serialized in a predetermined order. Although static and hybrid atomicity support quite different kinds of concurrency, they impose the same kinds of trade-offs between concurrency and efficiency and between concurrency and availability.

In the atomicity mechanism proposed by Reed [Reed 83, Reed 78], each transaction is issued a *pseudotime*, which is a timestamp used to establish the transaction's ordering with respect to other transactions. We do not discuss methods for choosing pseudotimes because these are discussed in detail by Reed. It is convenient to record the timestamp assigned to a transaction by placing a *Begin* entry in the log of each DM visited by a transaction, using the timestamp for the begin entry as the timestamp for the transaction.

For dynamically serialized objects, scheduling is accomplished by delaying particular transactions until other transactions commit or abort. For objects employing static atomicity, scheduling is accomplished by a combination of delays and refusals. For example, to execute a *Read* event with a particular pseudotime, it is necessary to observe the *Write* event with the most recent pseudotime preceding that of the reader. If necessary, the reader must wait until the transaction that executed the *Write* commits or aborts. To execute a *Write*, it is necessary to ensure that no transaction with a later pseudotime has read the value that is being overwritten. If the value has been read, the *Write* attempt must be refused. Because each transaction waits only for transactions with earlier pseudotimes, this scheme is not subject to deadlock. By contrast, under hybrid atomicity requests are never refused, but deadlocks can occur.

Local concurrency control for static atomicity is based on the notion of a *possibility lock*.

When a new event is appended to a DM's log, that transaction is granted a *possibility lock* for the event, which is released when the transaction commits. While the transaction holds the lock, no transaction executing a conflicting request with a later pseudotime may read the DM's log. As before, the intersection of the quorum intersection and lock conflict relations must be a dependency relation. The full replication method works as follows.

1. When a TM receives a request from a client, it forwards the request to an initial quorum of DM's.

2. Each DM that receives the request checks whether a transaction with an earlier pseudotime holds a conflicting possibility lock. If so, the request is postponed until the conflicting transaction commits or aborts. If there are no conflicts, the DM responds with the entries for committed transactions having earlier pseudotimes than the requesting transaction.

3. When the TM has received replies from an initial quorum, it merges the logs, constructs a view, and chooses a response. The TM appends the new entry to the view, and sends the view to a final quorum of DM's.

4. Each DM checks whether it has an event executed by a transaction with a later pseudotime that depends on the new event. If so, the update is refused. Otherwise, the DM grants a possibility lock for the event, and merges the updated view with its resident log. As soon as a final quorum of DM's has returned acknowledgments, the TM returns the response to the client.

Step Two ensures that all transactions that can affect the response have committed. Step Four ensures that the new response cannot invalidate existing events. This technique is not subject to deadlock, because a transaction never waits for a transaction with a later pseudotime, and so cycles cannot arise. Some additional optimizations are possible; under certain circumstances a DM may refuse a request in Step Two, if it can ascertain that any response to the request will conflict with an event having a later pseudotime.

Like the local scheme for hybrid atomic objects, this scheme does not permit scheduling decisions to take advantage of state information, and consequently it is limited in the level of concurrency it can support. For the FIFO queue, a transaction attempting to dequeue an item may be blocked unnecessarily because a transaction with an earlier pseudotime has

enqueued an item, or an attempt to enqueue an item may be refused unnecessarily because a transaction with a later pseudotime has dequeued an item.

Following our earlier analysis for hybrid atomic objects, these problems can be alleviated only if concurrency control employs non-local state information. The following is a summary description of a non-local concurrency control method based on static atomicity.

1. When a TM receives a request from a client, the TM requests the logs from a collection of DM's. Each DM that responds grants the transaction a short-term *request lock* for that request. The logs are merged in timestamp order to construct a *view*.

2. If the view does not permit a response consistent with the object's behavioral specification, the TM releases the locks at the DM's. If a legal response may become possible in the future (e.g. the client is attempting to read a value written by an earlier uncommitted transaction), the TM waits for the object's state to change and then restarts the protocol. If no legal response will ever be possible (e.g. the client is attempting to overwrite a value already read by a later transaction), then the TM returns a refusal to the client. Otherwise a response is chosen.

3. The TM generates a new timestamp, appends the new entry to the view, and sends the updated view to a final quorum of DM's. Each DM accepts the view as soon as it has no conflicting request locks, merges the view with its resident logs, and returns an acknowledgment. As soon as the TM receives acknowledgments from a final quorum, all request locks are released.

4. The response is returned to the client.

This protocol differs from the non-local protocol under hybrid atomicity only in Step Two, where a response is chosen on the basis of a static atomic behavioral specification. Like the earlier method, this non-local method is also fully general: it can realize any level of concurrency consistent with static atomicity.

The bank account data type given in an earlier example can be used to illustrate that highly concurrent implementations may impose additional constraints on quorum choices for static atomicity as well as for hybrid atomicity. The constraints on quorum choice are the same for

both local schemes: *Withdraw* requests depend on *Deposit* and *Withdraw* events, but *Deposit* requests do not depend on any other events. Consider a situation in which the initial balance of the account is zero, and transaction **A** has a later pseudotime than transaction **B** (indicated here by the relative ordering of their *Begin* events). **A** has observed that the account balance is less than twenty dollars, and transaction **B** attempts to deposit ten dollars. Transaction **B** could be allowed to proceed, because all outcomes of the following history are static atomic:

```
                                [Begin B]
[Begin A]
[Withdraw(20);Overdrawn();A]
                                [Deposit(10);Ok();B]
```

On the other hand, if transaction **C** had previously deposited ten dollars, then **B** must be refused, because the following history is not static atomic:

```
                                                [Begin C]
                                                [Deposit(10);Ok();C]
                                                [Commit C]
                        [Begin B]
[Begin A]
[Withdraw(20);Overdrawn();A]
                        [Deposit(10);Ok();B]
```

If **B** and **C** both commit, these transactions will be serialized in the order **A** follows **B** follows **C**, yielding the following illegal serial history:

```
[Deposit(10);Ok()]
[Deposit(10);Ok()]
[Withdraw(20);Overdrawn()]
```

Consequently, if **B** is to proceed in the first situation but not the second, then all prior *Deposit* events must appear in the view for *Deposit*, resulting in a stronger dependency relation and an additional constraint on quorum choice.

3.6 Discussion

We have identified two basic trade-offs in the choice of concurrency control methods for replicated objects: the level of concurrency is inversely related to the level of message traffic, and perhaps more surprisingly, the level of concurrency is inversely related to flexibility in quorum choice.

We first described a simple and efficient *local scheme* based on predefined conflicts. Because each DM makes its own scheduling decisions on the basis of locally resident information, the local scheme introduces no message traffic and hence does not affect the object's availability. With respect to concurrency, the local scheme is optimal for the amount of information it uses: no other scheme in which scheduling decisions are made exclusively on the basis of predefined conflicts can provide a higher level of concurrency. Because the local scheme does not take state information into account, it has two shortcomings: it is limited in the level of concurrency it can support, and it cannot provide adequate support for partial operations.

We next described an alternative concurrency control method, called the *non-local* scheme, in which scheduling decisions can take advantage of state information. The non-local scheme can realize a higher level of concurrency than the local method; in fact it can be used to implement any concurrency control scheme that preserves atomicity. The non-local scheme also supports partial operations. The additional power comes at a cost: the non-local scheme may require more complex local computations and more message traffic. If a high level of concurrency is desired, the non-local scheme may also place additional constraints on quorum choice. As one increases the concurrency provided by an atomic object, one increases the information needed to make scheduling decisions, and one thus

reduces one's freedom in choosing quorums.

For both hybrid and static atomicity, the same trade-offs arise between concurrency and efficiency, and between concurrency and availability. In both local methods, scheduling decisions can be made efficiently, the number of messages needed to execute an operation invocation is fixed, and the constraints on quorum choice are determined by the serial specification. Both non-local methods support a higher level of concurrency, but may require arbitrarily complex local computations, additional message traffic, and additional constraints on quorum choices.

We have shown that the notion of a dependency relation is central to both quorum choice and concurrency control. Dependency relations appeared in the definitions of quorum intersection relations, conflict relations for local locks and short-term locks, as well as notification techniques. This common structure should not be surprising, because techniques for replication and concurrency control are both based on notions of events that affect one another. Very informally, if one event does not affect the other, then those events can have non-intersecting quorums, and they can execute concurrently. The notion of a dependency relation provides a precise characterization of what it means for one event to affect another.

3.7 Related Work

The model of computation employed in this chapter is essentially the model of computation provided by the Argus programming language [Liskov 82]. Argus provides nested transactions and built-in atomic objects employing strict two-phase locking, as well as the ability for programmers to define their own atomic objects. It is instructive to compare our approach to constructing replicated atomic objects to the approach to constructing single-site atomic objects that is supported by Argus [Weihl 83]. Aside from the obvious difference that our approach encompasses replication, the two approaches have a number of similarities. Both achieve a high degree of concurrency by employing a non-atomic

representation protected by short-term synchronization mechanisms. Both employ representations in which periodic garbage collection of obsolete information may be used to enhance performance, and both share the shortcoming that scheduling is performed using a form of busy-waiting.

Although our model of atomic objects is based on a model proposed by Weihl [Weihl 83a, Weihl 84], it differs from Weihl's model in several respects. Weihl's model treats requests and responses as distinct events; the ordering of requests and responses models the order in which requests are received and responses chosen. Our model treats a request and its associated response as a single unit; the ordering of request/response pairs models the order in which responses are chosen. The difference is minor, but our model permits us to use the same formal definition of dependency for both serial and behavioral histories, while Weihl's model would have required different formulations in each case.

Perhaps a more important difference is that Weihl's model is primarily intended to capture the behavior of objects that reside at a single location, while ours is explicitly concerned with objects whose representations are distributed. A computation in a distributed system can be viewed as a partially ordered set of events [Lamport 78], where events occurring at a single location are totally ordered, but events occurring at distinct locations might not be ordered. An *observation* of a computation is a total ordering of its events compatible with the partial ordering. If an object resides at a single location, then every observation will place the same ordering on its events, while if an object resides at multiple locations, different observations might order the events differently. In Weihl's scheme, constraints on timestamp generation for hybrid atomicity are expressed in terms of an observation of the events affecting individual objects. These constraints need to be generalized to deal with replicated objects, where different observations may order those events differently. Although this generalization appears straightforward, a formal model of atomicity for replicated objects has yet to be worked out in detail.

The local scheme presented in this chapter may be viewed as a generalization of the locking scheme proposed by Moss [Moss 81], which is itself a generalization of two-phase locking [Eswaren 76]. Moss's scheme supports non-replicated objects having *Read* and *Write* operations. The local scheme generalizes Moss's scheme in two ways: first, by providing a systematic method for defining lock conflicts for arbitrary operations in a system based on hybrid atomicity, and second, by extending the locking scheme to replicated objects.

It is instructive to compare the local scheme to other proposals for conflict-based synchronization [Bernstein 81, Korth 81, Weihl 84]. These proposals are based on the notion of commutativity of operations. The local scheme described in this chapter is less restrictive than the commutativity-based schemes in the following sense: it can be shown that every lock conflict relation induced by the commutativity-based rules is a correct lock conflict relation for the local scheme, but the local scheme may permit lock conflict relations not permitted by the commutativity-based rules. For example, the commutativity-based schemes require locks for *Enq* events to conflict, while the local scheme does not. The additional power of the local scheme is due to the use of timestamps to provide a globally known serialization ordering for transactions. This comparison, together with the optimality result given above, suggest that that the local scheme is of interest even for objects that are not replicated.

We have not considered *optimistic* scheduling methods [Kung 81], in which transactions execute with no synchronization whatsoever. Each transaction is allowed to commit only after *certifying* that the resulting history remains serializable. Because transactions are never delayed by synchronization conflicts, optimistic methods promise better performance if synchronization conflicts are rare and transactions are almost always allowed to commit. Just as for the other concurrency control methods we have discussed, the challenge in adapting the optimistic approach to replicated data lies in managing the (replicated) information needed to make scheduling decisions, in this case certifying that the result of committing a transaction preserves serializability.

Chapter Four

Reconfiguration

4.1 Introduction

In the replication methods considered in previous chapters, the quorums for events have been fixed. In this chapter, we extend General Quorum Consensus to support *reconfiguration*: the ability to change the quorums for existing objects. One use for reconfiguration is to adapt to changes in requirements. 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 can also facilitate changes in hardware by permitting an object to be moved from one set of sites to another. Perhaps the most interesting use for reconfiguration is as a technique for masking failures. The quorum structure is an object's first line of defense: as long as a quorum for an operation remains available, the operation's availability is unaffected by failures. As failures occur, however, there is an increasing likelihood that subsequent failures will cause all quorums to be lost. After a certain number of failures have occurred, it becomes desirable to reconfigure the object, perhaps by redistributing quorums among the remaining DM's, perhaps by initializing new DM's to replaced the failed ones.

In Section Two, we review a reconfiguration method for replicated files proposed by Gifford. A benefit of this method is that the ability to reconfigure does not impose a cost unless it is actually used. Reconfiguring a file results in a temporary period of decreased availability and increased message traffic as TM's learn of the new configuration. In Section Three, we propose a reconfiguration method that generalizes Gifford's technique. Reconfiguration is accomplished by introducing levels of indirection in the representation of reconfigurable replicated objects. The use of indirection to implement reconfiguration is not new; our contribution is to have worked out a general model that takes into account the interaction of

reconfiguration with replication. Like Gifford's scheme, our scheme has the property that the ability to reconfigure incurs a cost only when reconfiguration actually occurs. We also propose a replicated reference counting scheme to permit old configuration information to be garbage-collected when it is no longer needed. The reference counting scheme does not decrease the object's availability.

Quorum consensus can be used as a building block from which one can construct a variety of replication methods. In Section Four, we show that existing replication methods for files such as the *primary copy* scheme [Alsberg 76, Stonebreaker 79] and the *true-copy token* scheme [Minoura 79, Minoura 82] can be viewed as specially optimized instances of the multi-round quorum consensus algorithm. Section Five contains a discussion of related work and a summary of our results.

4.2 Gifford's Reconfiguration Method

In Gifford's scheme [Gifford 79], a file's current set of quorums is recorded as part of its representation. We refer to the file's quorum information as its *configuration state*. Each DM keeps a copy of the the configuration state together with a version number, called its *generation number*. The file's configuration state can be read or written, and the quorums for reading or writing the configuration state are the same as the quorums for reading or writing the file itself.

A file is reconfigured as follows. The current version of the file is recorded at a new final *Write* quorum. A new generation number is then chosen by incrementing the highest generation number observed among an old initial *Write* quorum. The new configuration state is recorded (with the newer generation number) at both an old and a new final *Write* quorum.

Each TM keeps a cache containing the most recent configuration state and generation number it has observed. When a TM reads from an initial *Read* or *Write* quorum, it reads

both the file versions and the configuration states. If the TM observes a generation number later than the one it has cached, it discards the file versions, and starts over using the more recent set of quorums.

If timestamps are used in place of version numbers, this scheme requires minor changes because the *Write* operation may have an empty initial quorum. Each *Read* or *Write* quorum for the file must include a *Read* quorum for the current configuration state. When a TM sends a request to a DM, it includes the timestamp for its cached configuration state. If the DM's configuration state has a later timestamp, the DM rejects the request, and notifies the TM that its configuration state is out-of-date. A TM attempting to write to a file would not be notified of an out-of-date configuration state until it attempts to update a final *Write* quorum, but the effect is the same.

The principal merit of this method is that the ability to reconfigure does not incur a cost in increased message traffic and reduced availability until the file is actually reconfigured. Because every initial *Read* or *Write* quorum is certain to include the most recent generation number, the currency of the TM's cached configuration state can be checked without communicating with additional DM's and without sending extra messages. The cost of reconfiguration is incurred just once per TM. A TM that has not yet learned of the most recent reconfiguration must locate both an old and a new quorum for the operation it is attempting to execute, using one round of messages for each. A TM that has missed several reconfigurations must employ a round of messages to locate a quorum for each successive configuration state.

A shortcoming of this method is that it does not provide a way to discard old configuration states when all TM's have updated their caches. Care must be taken that the old configuration state is not discarded as long as some TM has not updated its cache, because otherwise that TM may be unable to locate the current resource state. Consequently, this method can be used to redistribute quorums among a fixed set of DM's, but it is not

appropriate for moving a file from one set of DM's to another.

In the next section, we present a model of reconfiguration that generalizes Gifford's scheme. Our model permits reconfiguration for objects of arbitrary type, and we identify constraints on quorum choice that ensure that reconfiguration incurs no expense when it is not actually used. We also propose a replicated reference counting scheme that allows obsolete quorum information to be discarded when it is no longer needed.

4.3 Reconfigurable Objects

4.3.1 A Very Simple Method

We first describe a simple reconfiguration method that supports a limited form of reconfiguration. We then introduce successive refinements that increase the flexibility of the reconfiguration method while reducing its cost.

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. In subsequent discussions we will assume that an object's configuration state provides *Read* and *Write* operations, although more sophisticated data types, such as tables, would do equally well.

Each TM is initialized with the *Read* and *Write* quorums for the configuration state. To apply an operation to the object, a TM carries out the following protocol:

1. The TM reads from the configuration state the current initial and final quorums for the resource state operation.

2. The TM applies the operation to the resource state and returns the results to the client.

This protocol involves two rounds of messages: one to ascertain the quorum for the resource state and one to carry out the operation. Each quorum for the operation has two parts: (i) a quorum for reading the configuration state and (ii) a quorum for operating on the resource state.

To describe the quorum for reconfiguration, it is convenient to introduce some new terminology. An initial (or final) *coquorum* for an event is defined to be any set of DM's that intersects every initial (or final) quorum for that event. For example, each initial quorum for the file *Read* operation is a final coquorum for *Write*. To change the quorums for an event, it is necessary to merge the logs from an old initial quorum, and to write out the resulting view to a new initial coquorum.

The full protocol for changing the quorums for a resource state operation is given by the following steps. We remark that because reconfiguration must preserve quorum intersections, it typically involves the simultaneous alteration of quorums for multiple events.

1. The TM reads from the configuration state to ascertain the current quorums for the resource state.

2. The TM constructs a view by merging the logs from an initial quorum for each resource state operation being reconfigured.

3. The TM initializes the new resource state by writing out the view to a new initial coquorum for each resource state operation being reconfigured.

4. The TM records the new sets of quorums at the configuration state.

This method supports only a limited form of reconfiguration: the quorums for the resource state can be changed, but not the quorums for the configuration state.

To illustrate this method, let us examine the implementation of a reconfigurable queue. We use the notation (m,n) to indicate that an event has initial quorums consisting of any m DM's

from a particular set, and final quorums consisting of any *n* DM's from that same set. In this example, the queue's configuration state is replicated among DM1, DM2, and DM3, with *Read* quorums of (1,0) and *Write* quorums of (0,3). When the queue is created, the resource state is replicated among DM4, DM5, and DM6, with *Enq* quorums of (0,1), and *Deq* quorums of (3,1). The configuration state includes a timestamp (here t0), the identities of the DM's storing the resource state, and the quorums for *Enq* and *Deq*.

```
DM1                    DM2                    DM3
t0: {DM4,DM5,DM6}      t0: {DM4,DM5,DM6}      t0: {DM4,DM5,DM6}
    Enq = (0,1)            Enq = (0,1)            Enq = (0,1)
    Deq = (3,1)            Deq = (3,1)            Deq = (3,1)
```

A TM enqueues an item *x* by first reading the configuration state, say from DM1, and then by recording an *Enq* entry at a final *Enq* quorum (DM4).

```
DM1                    DM2                    DM3
t0: {DM4,DM5,DM6}      t0: {DM4,DM5,DM6}      t0: {DM4,DM5,DM6}
    Enq = (0,1)            Enq = (0,1)            Enq = (0,1)
    Deq = (3,1)            Deq = (3,1)            Deq = (3,1)

DM4                    DM5                    DM6
t1: [Enq(x);Ok()]
```

Now let us reconfigure the resource state, changing *Enq* quorums to (0,2) and *Deq* quorums to (2,2). The TM conducting the reconfiguration reads the current configuration state, and constructs a view by merging the logs from an initial quorum for *Enq* (which is empty) and *Deq* (which consists of all three DM's). The TM initializes the new resource state by writing out the view to a new initial coquorum for both *Enq* and *Deq* (DM4 and DM5). Finally, the TM records the new quorums by updating the configuration state at a final *Write* quorum (DM1, DM2, and DM3).

```
DM1                    DM2                    DM3
t2: {DM4,DM5,DM6}      t2: {DM4,DM5,DM6}      t2: {DM4,DM5,DM6}
    Enq = (0,2)            Enq = (0,2)            Enq = (0,2)
    Deq = (2,2)            Deq = (2,2)            Deq = (2,2)

DM4                    DM5                    DM6
t1: [Enq(x);Ok()]      t1: [Enq(x);Ok()]
```

Now suppose another TM attempts to enqueue an item *y*. After reading the configuration state, it enqueues the item at a new final *Enq* quorum (DM5 and DM6).

```
DM1                    DM2                    DM3
t2: {DM4,DM5,DM6}      t2: {DM4,DM5,DM6}      t2: {DM4,DM5,DM6}
    Enq = (0,2)            Enq = (0,2)            Enq = (0,2)
    Deq = (2,2)            Deq = (2,2)            Deq = (2,2)

DM4                    DM6                    DM6
t1: [Enq(x);Ok()]      t1: [Enq(x);Ok()]
                       t3: [Enq(y);Ok()]      t3: [Enq(y);Ok()]
```

This example illustrates the costs associated with reading the configuration state before executing each operation.

1. Each operation invocation requires an additional round of messages to read the configuration state.
2. Each operation invocation requires locating a quorum for both the configuration and the resource states, potentially reducing availability.

In addition, this method supports only a very restricted form of reconfiguration: the quorums for the resource state can be changed, but the quorums for the configuration state are fixed. In the next sections, we address each of the problems in turn, introducing refinements to alleviate these costs and to permit the quorums for the configuration state itself to change.

4.3.2 Reconfiguring the Resource State

Message delays can be reduced if each TM maintains a cache containing the value of the most recently observed configuration state. To execute an operation, the TM carries out the following steps concurrently, perhaps executing each as a nested transaction:

- It reads the current configuration state to determine whether its cache is current.
- It applies the operation directly to the resource state using the quorums from its cached configuration state.

If the TM's cache is found to be current, then the operation has been executed in a single round of messages. If the cache is found to be obsolete, then the TM must abort the second step, update its local cache, and start over. Using this scheme, each TM must use an additional round of messages immediately after each reconfiguration, but a single round of messages suffices between reconfigurations.

Rather than choosing the quorums for the configuration and resource states independently, reconfiguration can be rendered more highly available if the quorums for the two components satisfy the following *quorum inclusion* property:

> Every quorum for a resource state operation includes a quorum for reading the configuration state.

(For the moment, let us assume that this property is preserved across reconfigurations; this assumption is relaxed in the next section.) TM's execute operations using the following protocol. A TM first attempts to apply the operation directly to the resource state, using the quorums from its cached configuration state. With each request to read or update the DM's log, the TM includes the timestamp for its cached configuration state. Each DM compares the TM's cached timestamp with the timestamp from its own configuration state. If the TM's timestamp is out of date, the request is refused, and the TM is notified of the new

configuration state. The TM aborts the invocation, updates its cache, and starts over. The quorum inclusion property ensures that if the TM can locate a quorum of DM's willing to execute the operation, then the TM's configuration state is current.

To illustrate this technique, let us review the queue example once again. Both the configuration and resource states are stored at the same three DM's, and the TM's caches initially indicate that the quorums for *Enq* and *Deq* are (0,1) and (3,1) respectively.

```
DM1                    DM2                    DM3
t0: {DM1,DM2,DM3}      t0: {DM1,DM2,DM3}      t0: {DM1,DM2,DM3}
    Enq = (0,1)            Enq = (0,1)            Enq = (0,1)
    Deq = (3,1)            Deq = (3,1)            Deq = (3,1)
```

To enqueue an item *x*, a TM sends an *Enq* entry together with its encached configuration state timestamp (t0) to a final *Enq* quorum, say DM1. Because the timestamp in the message matches the timestamp for its own configuration state, DM1 accepts the request.

```
DM1                    DM2                    DM3
t0: {DM1,DM2,DM3}      t0: {DM1,DM2,DM3}      t0: {DM1,DM2,DM3}
    Enq = (0,1)            Enq = (0,1)            Enq = (0,1)
    Deq = (3,1)            Deq = (3,1)            Deq = (3,1)

t1: [Enq(x);Ok()]
```

As before, the configuration state is then reconfigured, changing *Enq* quorums to (0,2) and *Deq* quorums to (2,2). Note that this reconfiguration preserves the quorum inclusion property: each quorum for the resource state (any two DM's) includes a *Read* quorum for the configuration state (any single DM).

```
DM1                    DM2                    DM3
t2: {DM1,DM2,DM3}      t2: {DM1,DM2,DM3}      t2: {DM1,DM2,DM3}
    Enq = (0,2)            Enq = (0,2)            Enq = (0,2)
    Deq = (2,2)            Deq = (2,2)            Deq = (2,2)

t1: [Enq(x);Ok()]      t1: [Enq(x);Ok()]
```

When another TM attempts to enqueue an item *y*, it sends the *Enq* entry to DM2 (which it believes to be a final *Enq* quorum), together with the timestamp t0 from its cached configuration state. DM2 observes that the timestamp in the message is less than its own timestamp, and it refuses the request, responding with copy of the new configuration state. The TM updates its cache and sends the *Enq* entry to a new final quorum, say DM2 and DM3, which both accept the request because the timestamps match.

```
DM1                    DM2                    DM3
t2: {DM1,DM2,DM3}      t2: {DM1,DM2,DM3}      t2: {DM1,DM2,DM3}
    Enq = (0,2)            Enq = (0,2)            Enq = (0,2)
    Deq = (2,2)            Deq = (2,2)            Deq = (2,2)

t1: [Enq(x);Ok()]      t1: [Enq(x);Ok()]
                       t3: [Enq(y);Ok()]      t3: [Enq(y);Ok()]
```

The quorum inclusion property provides two benefits:

- *Enhanced availability:* Each quorum for operating on the resource state alone is a quorum for the operation as a whole. Thus the the availability of the operation as a whole is unaffected by the need to check the currency of the configuration state.

- *Fewer messages:* In the absence of reconfiguration, a TM can use the same physical messages to carry out two tasks: to establish the currency of its cached configuration state, and to apply the operation to the resource state.

As long as the quorum inclusion property is satisfied, the ability to reconfigure does not

incur a cost in reduced availability or increased message traffic until the object is actually reconfigured. Nevertheless, this reconfiguration method is still not really satisfactory, because certain kinds of reconfiguration will not preserve the quorum inclusion property. For example, it would not be possible to move the queue to a different set of DM's because there is no provision for changing the quorums for the configuration state. This problem is addressed in the next section.

4.3.3 Reconfiguring the Configuration State

We now describe a more flexible scheme, in which an object has a chain (or linked list) of configuration states. Each configuration state has two possible values:

1. *Current:* it contains quorum information for the resource state.
2. *Obsolete:* it contains quorum information for the next configuration state in the chain.

An object has a single (current) configuration state when it is initialized. When the configuration state is reconfigured, the chain is extended: a new configuration state is created to hold the quorum information for the new resource state, and the quorum information for the new configuration state is stored in the old configuration state, which is then marked *obsolete*. Once a configuration state has been marked obsolete, it not subsequently modified. The current configuration state always satisfies the quorum inclusion property: each quorum for a resource operation includes a quorum for reading the current configuration state.

Operations are carried out just as before. Each TM includes the timestamp from its cached configuration state with each message, and each DM refuses requests with obsolete timestamps. If a TM discovers its cache is obsolete, it must follow the chain, locating a quorum for each configuration state in turn, until the current quorums for the resource state are discovered. The operation can be applied directly to the resource state once its

quorums are known. The chain must be followed only once by each TM: when a new configuration state is observed, it is recorded in the TM's cache.

Reconfiguration is accomplished by the following protocol:

1. Locate the current configuration state and read the quorums for the resource state.
2. Merge the logs from an initial quorum for the old resource state and write out the view to an initial coquorum for the new resource state.
3. Initialize the new configuration state with the quorum information for the new resource state.
4. Update the old configuration state to point to the new configuration state.

To illustrate this technique, we present a more complicated example of reconfiguration. The queue initially resides at DM1, DM2, and DM3. As before, the quorums for *Enq*, *Deq*, *Read*, and *Write* are (0,1), (3,1), (1,0), and (0,3) respectively. Following reconfiguration, the queue resides at DM4, DM5, and DM6, and the quorums for *Enq*, *Deq*, *Read*, and *Write* are (0,2), (2,2), (2,0), and (0,2) respectively.

As before, an item *x* is enqueued before the object is reconfigured.

```
DM1                    DM2                    DM3
t0: Current            t0: Current            t0: Current
    {DM1,DM2,DM3}          {DM1,DM2,DM3}          {DM1,DM2,DM3}
    Enq = (0,1)            Enq = (0,1)            Enq = (0,1)
    Deq = (3,1)            Deq = (3,1)            Deq = (3,1)

t1: [Enq(x);Ok()]
```

To reconfigure the queue, the TM conducting the reconfiguration reads the current configuration state, say at DM1, and then constructs a view by merging the logs from an initial quorum for *Enq* (which is empty) and *Deq* (which consists of all three DM's). The TM

initializes the new resource state by writing out the view to a new initial coquorum for both *Enq* and *Deq* (DM4 and DM5). It records the new configuration state at a new *Write* quorum (DM4 and DM5), marks the old configuration state as obsolete, and records at the old configuration state the *Read* and *Write* quorums for the new configuration state.

```
DM1                    DM2                    DM3
t0: Obsolete           t0: Obsolete           t0: Obsolete
    {DM4,DM5,DM6}          {DM4,DM5,DM6}          {DM4,DM5,DM6}
    Read = (2,0)           Read = (2,0)           Read = (2,0)
    Write = (0,2)          Write = (0,2)          Write = (0,2)

DM4                    DM5                    DM6
t2: Current            t2: Current
    {DM4,DM5,DM6}          {DM4,DM5,DM6}
    Enq = (0,2)            Enq = (0,2)
    Deq = (2,2)            Deq = (2,2)

t1: [Enq(x);Ok()]      t1: [Enq(x);Ok()]
```

The quorum for a subsequent reconfiguration consists of any two out of DM3, DM4, or DM5.

When another TM attempts to enqueue an item *y*, it sends the *Enq* entry to DM2 (which it believes to be a final *Enq* quorum), together with the timestamp t0 from its cached configuration state. DM2 refuses the request, and sends the quorums for the new configuration state back to the TM. The TM updates its cache and sends the the new timestamp and *Enq* entry to a new final *Enq* quorum (DM5 and DM6). This request is accepted because neither DM has configuration state with a later timestamp.

```
DM1                    DM2                    DM3
t0: Obsolete           t0: Obsolete           t0: Obsolete
    {DM4,DM5,DM6}          {DM4,DM5,DM6}          {DM4,DM5,DM6}
    Read = (2,0)           Read = (2,0)           Read = (2,0)
    Write = (0,2)          Write = (0,2)          Write = (0,2)

DM4                    DM5                    DM6
t2: Current            t2: Current
    {DM4,DM5,DM6}          {DM4,DM5,DM6}
    Enq = (0,2)            Enq = (0,2)
    Deq = (2,2)            Deq = (2,2)

t1: [Enq(x);Ok()]      t1: [Enq(x);Ok()]
                       t3: [Enq(y);Ok()]      t3: [Enq(y);Ok()]
```

Once a TM has replaced an obsolete cache, it has effectively reconfigured its own configuration state. Reconfiguration imposes a penalty on a TM whose cache is out of date, because it must conduct an additional exchange of messages the next time it attempts to carry out an operation. To reduce this cost, a TM that reconfigures the configuration state might attempt to notify as many other TM's as possible. As a further optimization, if the chain of obsolete configuration states becomes longer than two, early obsolete configuration states could be modified to point directly to the current configuration state. A TM with an out-of-date cache that is unable to follow the chain could query DM's and other TM's, asking for "hints" about the location of the current configuration state. If reconfiguration is infrequent, then it is unlikely that caches will fall far behind.

4.3.4 Reference Counts

A drawback of all the reconfiguration schemes we have considered 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 DM's to another, the configuration states at the old set of DM's cannot be discarded as long as there is a possibility that some TM's cache contains the old configuration state. If the old configuration states are discarded before the TM's cache is brought up to date, then the TM will be unable to locate the new resource state. This section proposes a simple reference counting scheme that enables the object's maintainers to detect when all TM's 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 *Ref_Count* date type provides three operations. The state of the object 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 *Ref_Count* type is shown in Figure 13, together with the range of quorum choices for five identical DM's. *Value* requests depend on both *Inc* and *Dec* events, but *Inc* and *Dec* requests do not depend on any prior events because neither returns any information.

Each configuration state is modified to include a reference count. The reference count is incremented each time a TM first records that configuration state in its cache. The reference count is decremented each time a TM observes that the configuration state has been rendered obsolete. Once a configuration state has been rendered obsolete, the object's maintainers may use the *Value* operation to detect when the configuration state may be garbage collected. 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 TM's and DM's, an unneeded configuration state will always have a reference count of zero.)

By choosing the quorums for a configuration state's *Inc* and *Dec* operations to be the same as the quorums for its *Read* operation, reference counting can be accomplished without affecting the object's availability. A TM must locate an initial *Read* 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 TM can use the new configuration state, it must check its currency by locating an initial *Read* quorum. That same quorum can be used to increment the new configuration state's reference counter.

To illustrate the use of replicated reference counts, let us review the previous example one last time. Initially, the queue resides at DM1, DM2, and DM3. The quorums for *Read* and *Write* are (1,0) and (0,3), the quorums for *Enq* and *Deq* are (0,1) and (3,1), and the quorums for *Inc*, *Dec*, and *Value* are (0,1), (0,1), and (3,0). The item *x* is enqueued at DM1 as before. There are two TM's: TM1 and TM2, thus the reference count is initialized to two by recording two *Inc* entries at DM1.

Fig. 13. The Reference Counter Type

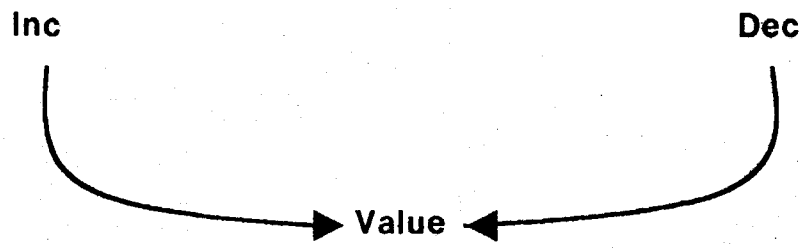

Figure 13a: The quorum intersection graph.

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)

Figure 13b: Minimal quorum choices for five identical DM's.

```
DM1                    DM2                    DM3
t0: Current            t0: Current            t0: Current
    {DM1,DM2,DM3}          {DM1,DM2,DM3}          {DM1,DM2,DM3}
    Enq = (0,1)            Enq = (0,1)            Enq = (0,1)
    Deq = (3,1)            Deq = (3,1)            Deq = (3,1)

t1: [Inc(); Ok()]
t2: [Inc(); Ok()]

t3: [Enq(x);Ok()]
```

As before, the queue is reconfigured to reside at DM4, DM5, and DM6, with *Read* and *Write* quorums of (2,0) and (0,2), *Enq* and *Deq* quorums of (0,2) and (2,2), and *Inc*, *Dec*, and *Value*

quorums of (0,2), (0,2), and (2,0) respectively. Note that each quorum for either *Enq* or *Deq* is also a quorum for the configuration state's *Read* operation, as well as the reference count's *Inc* and *Dec* operations. When TM1 reconfigures the object, it updates its cache, decrements the counter at the old configuration state (by recording a *Dec* entry at DM2), and increments the counter at the new configuration state (by recording an *Inc* entry at DM5 and DM6).

```
DM1                    DM2                    DM3
t0: Obsolete           t0: Obsolete           t0: Obsolete
    {DM4,DM5,DM6}          {DM4,DM5,DM6}          {DM4,DM5,DM6}
    Read = (2,0)           Read = (2,0)           Read = (2,0)
    Write = (0,2)          Write = (0,2)          Write = (0,2)

t1: [Inc(); Ok()]
t2: [Inc(); Ok()]
                       t4: [Dec(); Ok()]

DM4                    DM5                    DM6
t6: Current            t6: Current
    {DM4,DM5,DM6}          {DM4,DM5,DM6}
    Enq = (0,2)            Enq = (0,2)
    Deq = (2,2)            Deq = (2,2)

t5: [Inc(); Ok()]       [Inc(); Ok()]

t3: [Enq(x);Ok()]      t3: [Enq(x);Ok()]
```

At a later point, TM1 checks the value of the old configuration state's reference count by merging the *Inc* and *Dec* entries from DM1, DM2, and DM3. The resulting view indicates that the reference count still has the value one, and so the old configuration state cannot be discarded. As before, when TM2 attempts to enqueue *y* at DM3, it detects that its configuration state is obsolete. The TM updates its cache, records a *Dec* entry at DM3, and sends *Inc* and *Enq* entries to a final quorum for both *Inc* and *Enq* (DM5 and DM6).

```
DM1                    DM2                    DM3
t0: Obsolete           t0: Obsolete           t0: Obsolete
    {DM4,DM5,DM6}          {DM4,DM5,DM6}          {DM4,DM5,DM6}
    Read = (2,0)           Read = (2,0)           Read = (2,0)
    Write = (0,2)          Write = (0,2)          Write = (0,2)

t1: [Inc(); Ok()]
t2: [Inc(); Ok()]
                       t4: [Dec(); Ok()]
                                              t7: [Dec(); Ok()]

DM4                    DM5                    DM6
t6: Current            t6: Current
    {DM4,DM5,DM6}          {DM4,DM5,DM6}
    Enq = (0,2)            Enq = (0,2)
    Deq = (2,2)            Deq = (2,2)

t5: [Inc(); Ok()]      t5: [Inc(); Ok()]
                       t8: [Inc(); Ok()]      t8: [Inc(); Ok()]

t3: [Enq(x);Ok()]      t3: [Enq(x);Ok()]
                       t9: [Enq(y);Ok()]      t9: [Enq(y);Ok()]
```

When TM1 next checks the old configuration state's reference count, it observes that the counter value has reached zero, and the obsolete configuration state can now be discarded.

4.3.5 Summary

This section first introduced a simple technique that employs a level of indirection to support a limited ability to reconfigure. Using this technique, each operation execution requires two rounds of messages instead of one, and there is a potential decrease in availability resulting from the need to communicate with additional sites.

A combination of cache management, careful quorum choice, multiple levels of indirection,

and a replicated reference counting scheme are used to transform the simple technique into a more powerful and flexible scheme, in which the ability to reconfigure incurs a cost only when reconfiguration actually takes place.

4.4 Examples

In this section, we show how two replication methods from the literature can be modeled by the reconfiguration method described in the previous section. We examine the *primary copy* scheme [Alsberg 76, Stonebreaker 79] and the *true-copy token* scheme [Minoura 79, Minoura 82]. The purpose of this section is both to illustrate the flexibility of multiple-round quorum consensus and to provide additional insight into the other replication methods.

4.4.1 The Primary Copy Scheme

We first consider a variant of the primary copy scheme [Alsberg 76, Stonebreaker 79]. The scheme described here is not exactly the same as the one originally proposed by Alsberg and Day because we make use of an underlying transaction system.

A replicated file is said to be *n-site resilient* if the file continues to be available for both reading and writing after the simultaneous failure of $n-1$ DM's. The following scheme achieves n-site resiliency. Timestamped versions of the file are kept at d DM's. One DM is designated as the *primary*, and the rest are *backups*. A current version of the file always resides at the primary DM, and at $n-1$ backups. The resource state is the current version of the file, and the configuration state is the identity of the primary DM. A *Read* quorum for the resource state consists of the primary, and a *Write* quorum consists of the primary and any $n-1$ backups. The configuration state is represented at each DM by a timestamped record indicating the identity of the primary DM. A read quorum for the configuration state consists of any n DM's, and a *Write* quorum consists of any $(d-n)+1$ DM's.

To read the file, a TM must locate the current primary (read the configuration state) and read

its version (read the resource state). As described in the previous section, each TM keeps a cache containing the identity of the most recently observed primary. To execute the *Read*, the TM sends a *Read* request including the timestamp for its cached configuration state to the cached primary and to *n-1* backups. If any of these DM's has a configuration state with a later timestamp, it refuses the request and informs the TM of the new primary's identity. Otherwise the primary returns the current version of the file, and the backups return a confirmation that the TM's configuration state is current. We emphasize that it is not enough to read directly from a DM that considers itself to be the primary, because a new primary may have been chosen by a quorum of backups executing in a distinct partition. An example where this problem arises is shown below.

Write operations are executed in a similar manner. The TM locates the current primary by sending a *Write* request to the cached primary and to *n-1* backups, and each DM either installs the new version or informs the TM that its cached configuration state is obsolete. To choose a new primary, the TM conducting the reconfiguration reads the file versions from $(d\text{-}n) + 1$ backups, writes the version with the latest timestamp to the new primary and to *n-1* backups, and then records the identity of the new primary at $(d\text{-}n) + 1$ DM's.

The following example illustrates a two-site resilient file replicated among three DM's. The quorum for locating the primary consists of any two DM's. Once the primary is known, it suffices to read from the primary and to write to the primary and one backup. Initially, DM1 is the primary and the value *x* has been written to DM1 and DM2.

DM1	DM2	DM3
t0: Primary: DM1	t0: Primary: DM1	t0: Primary: DM1
t1: Version: x	t1: Version: x	

If a partition isolates DM1, then the file may be reconfigured by reading the versions from DM2 and DM3, writing the version with the later timestamp to both DM2 and DM3, and

recording the identity of the new primary (here DM2) at both DM's. The resulting state is:

DM1	DM2	DM3
t0: Primary: DM1	t2: Primary: DM2	t2: Primary: DM2
t1: Version: x	t1: Version: x	t1: Version: x

Now suppose the partition is rejoined, and a TM with an obsolete configuration state `t0: Primary: DM1` attempts to read the file. The TM sends a *Read* request with its configuration timestamp to DM1 (which it believes to be the primary) and to DM3. DM1 accepts the *Read* request, believing itself to be the primary, but DM3 informs the TM that it has a more recent configuration state indicating that DM2 is the primary. The TM must then abort the previous attempt, update its cached configuration state, and try again. This example illustrates why it is not sufficient to read directly from a DM that believes itself to be the primary.

The availability properties of the primary copy scheme can be analyzed in terms of its quorum structure. A weak point in this scheme is that a new primary can be chosen only as long as fewer than *n* DM's have become unavailable. Once *n* DM's have failed, the file can no longer be reconfigured, and the file will become unavailable if the primary fails, resulting in a level of availability equivalent to that of a single-site file.[1] The problem is that the scheme we have described is a single-level scheme: although one can change the quorums for the resource state by electing a new primary, one cannot change the quorums for the configuration state by changing how the primary is elected.

A more robust version of the primary copy scheme can be defined as follows. The configuration state is extended to include the identities of the backups as well as the identity of the primary. To execute an operation, a TM must communicate with *n* current backup's to

---

1. The availability is actually marginally worse than a single-site file, because the failure of all remaining *d*-2 backups will also render the file unavailable.

verify that its configuration state is current. The TM either reads the version from the primary DM, or writes new versions at the primary and at *n-1* current backups. As before, if a TM's cache is current, both steps can be carried out in a single exchange of messages by including in each request the timestamp for the TM's cached configuration state. This scheme is more robust than the single-level scheme because failed DM's can be configured out completely. As long as failed DM's can be reconfigured out quickly enough to ensure that fewer than *n* have failed at any time, the file remains *n*-site resilient as long as *n* or more DM's are available. The file will be even more robust if the resiliency factor *n* can be adjusted in response to failures and repairs.

4.4.2 The True-Copy Token Scheme

We now consider the *true-copy token* scheme of Minoura, Owicki, and Wiederhold [Minoura 79, Minoura 82]. A replicated file is represented by a collection of versions stored at DM's. 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. The exclusive true copy can be reconfigured into multiple shared copies, and the set of shared copies can be reconfigured into a single exclusive copy. A true copy is indicated by the presence of a *true-copy token*, which also indicates whether the copy is shared or exclusive.

Minoura and Wiederhold have described techniques for regenerating true-copy tokens that have been destroyed by site failures [Minoura 82]. Nevertheless, under our assumption that site failures cannot be distinguished from partitions, the failure of a DM containing a true-copy token will limit further reconfiguration. If an exclusive copy token becomes unavailable, the 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 serves to enhance performance rather than availability.

The true-copy token scheme can be viewed as an instance of multi-round quorum

consensus. The resource state is just the set of true copies. If there exists a unique exclusive copy, the *Read* and *Write* quorums each consist of the DM at which the exclusive copy resides. If there exist multiple shared copies, a *Read* quorum consists of any DM at which a shared true-copy token resides, but there are no *Write* quorums; the file must be reconfigured before it can be written.

The current configuration state is the set of true copy tokens. Reconfiguration can take any of the following forms:

1. A exclusive copy token can be moved from one DM to another.
2. An exclusive copy token can become a shared copy token.
3. A shared copy token can be copied or moved from one DM to another.
4. The entire set of shared copy tokens can be replaced by a single exclusive copy token.

Although the authors do not describe in detail how to locate the current set of true-copy tokens, it is not difficult to adapt the configuration state chain scheme for this purpose. If each DM makes a record whenever a token is copied or moved, a TM attempting to locate the set of true-copy tokens can follow the chain of records until it locates the token(s). As described above, reference counts can be used to discard these records when they are no longer needed.

As an example of reconfiguration, let us consider how to transfer an exclusive copy from one DM to another. We note in parentheses how each step corresponds to a step in multi-round General Quorum Consensus.

1. Find the DM holding the exclusive copy token (locate the current configuration state).

2. Copy the file version from the source DM to the target DM (reconfigure the resource state).

3. Create a new exclusive copy token (initialize a new configuration state).

4. Replace the old token with a pointer to the DM containing the new token (update the old configuration state to point to the new configuration state).

Note that we assume that the underlying atomicity mechanism ensures that the transfer either succeeds completely or has no effect, and the resulting state is therefore consistent.

4.5 Discussion

This chapter has shown how to extend General Quorum Consensus to permit reconfiguration by introducing levels of indirection in an object's representation. Operations on such objects may require multiple rounds of messages, although we have shown that message traffic can be reduced by an appropriate choice of quorums. A replicated reference counting scheme is proposed to permit obsolete quorum information to be discarded. Some existing replication methods for files that employ reconfiguration can be viewed as special cases of multi-round quorum consensus: the primary copy scheme [Alsberg 76, Stonebreaker 79], the true-copy token scheme [Minoura 79, Minoura 82], and Gifford's reconfiguration method [Gifford 79].

Not all replication methods can be modeled by quorum consensus. The *available copies* algorithm for databases [Goodman 83] uses a level of indirection to support a kind of reconfiguration. The database is divided into two parts: each file is represented by a replicated set of copies analogous to the resource state, and the set of copies in current use is indicated by a replicated set of directories analogous to the configuration state. The available copies algorithm differs from ours in an important respect: as noted by its authors, it fails to preserve consistency in the presence of partitions. SDD-1 [Hammer 80] employs a similar replication scheme in which sites that have crashed are configured out of the system. It, too, does not tolerate communication failures or partitions. These examples are

discussed further in the next chapter, where we propose a method for enhancing availability in the presence of partitions.

Chapter Five

Partitions

5.1 Introduction

A *partition* occurs when an object's DM's are divided into two or more disjoint sets such that functioning members of distinct sets cannot communicate with one another. This chapter shows how to extend General Quorum Consensus to provide increased availability in the presence of partitions. Unlike some replication methods [Hammer 80, Goodman 83, Popek 81], quorum consensus preserves serializability in the presence of partitions. A TM can execute an operation if and only if it can locate an appropriate quorum within its partition. For example, if a file has a *Read* quorum of one and a *Write* quorum of *n*, then *Read* operations can be executed in distinct partitions, but *Write* operations cannot. The availability provided by General Quorum Consensus is limited by the constraint that operations whose quorums are required to intersect cannot be executed concurrently in distinct partitions. For example, one cannot execute a *Read* in one partition and a *Write* in another, or a *Deq* in one partition and an *Enq* or a *Deq* in the other.

This chapter introduces a reconfiguration technique that permits operations whose quorums are otherwise required to intersect to be executed in distinct partitions. Instead of providing a single set of quorums for each operation, an object may provide a *range* of alternative quorum sets. Each transaction chooses the quorum set that is best suited to the partition in which it is executing. Transactions in distinct partitions can each make progress by employing different quorum sets. Constraints on the range of quorums an object can provide ensure that transactions executing in distinct partitions remain serializable. Our scheme takes an approach similar to the one taken by Eager and Sevcik [Eager 83]. Our scheme makes two improvements: it is applicable to objects of arbitrary type, not just to files, and it provides a more practical mechanism for restoring normal operation when the

partition has been repaired.

5.2 The Missing Write Scheme

Eager and Sevcik [Eager 83] have proposed a replication method for files that allows *Read*'s and *Write*'s to take place concurrently in distinct partitions. In this section we describe their method in several stages, starting with a very simple description, and successively introducing complications. We outline correctness arguments for some of the techniques described; the reader is referred to [Eager 83] for more complete arguments.

Transactions execute by reading from and writing to replicated files. Each transaction executes in one of two *modes*: either normal or partitioned. In *normal mode*, a transaction writes to all copies of a file and reads from a single copy. In *partitioned mode*, the transaction reads and writes from a majority of copies, using Gifford's quorum consensus method [Gifford 79, Gifford 82]. The *Read* operation is more available for normal-mode transactions, and the *Write* operation is more available for partitioned-mode transactions.

Let us assume that all transactions serialized before a certain point execute in normal mode, and that all transactions serialized after that point execute in partitioned mode. We claim that each replicated file continues to satisfy the file specification: the value read is always the value most recently written. Suppose the file has been written by transaction **A**, and is subsequently read by transaction **B**. There are four cases to consider. If both **A** and **B** executed in normal mode, then all copies reflect **A**'s value, and the result is immediate. The same property holds if **A** executed in normal mode but **B** executed in partitioned mode. If both **A** and **B** executed in partitioned mode, then **B** is certain to observe at least one copy containing **A**'s value, and that copy will have the most recent version number. Finally, it is not possible for **A** to have executed in partitioned mode and **B** in normal mode, because we have assumed that normal mode transactions are never serialized after partitioned mode transactions.

We now describe a simple technique that ensures that normal mode transactions are never serialized after partitioned mode transactions. When a transaction in partitioned mode reads or writes a copy of the file, it leaves a *missing write token* with that copy. When a transaction in normal mode attempts to read from or write to a copy with a missing write token, the transaction is aborted and restarted in partitioned mode. This mechanism ensures that if a transaction in normal mode executes to completion, then it will not have read any values written by partitioned-mode transactions, nor will it have overwritten any values read by partitioned-mode transactions, and therefore it may be serialized before any partitioned-mode transactions.

What do these modes have to do with partitions? If a transaction in normal mode cannot update all the copies of a file, the transaction might still be able to make progress by restarting in partitioned mode and updating only a majority of copies. If a partition occurs, a normal-mode transaction in a partition containing a minority of copies can execute *Read*'s but not *Write*'s, while a partitioned-mode transaction in a majority partition can execute both *Read*'s and *Write*'s. Consequently, Eager and Sevcik's method permits *Read* and *Write* operations to be executed concurrently in distinct partitions by ensuring that partitioned-mode transactions are serialized after normal-mode transactions.

One shortcoming of this method is that it works only for files. It cannot be used to provide increased availability for, say, FIFO queues, because both *Enq* and *Deq* must be treated as *Write*'s and therefore cannot execute in distinct partitions. We will show how the approach taken by Eager and Sevcik can be combined with the mechanisms we have developed for General Quorum Consensus to provide increased availability for objects of arbitrary type.

A second shortcoming of Eager and Sevcik's method is that once a transaction has entered partitioned mode, it may be quite difficult to restore the system to normal mode. Once missing write tokens have been created, transactions that encounter such tokens will be restarted in partitioned mode, causing the tokens to propagate throughout the system.

Eventually, the likelihood that a normal-mode transaction will be able to execute to completion may become quite small. Once the partition is rejoined, however, it is desirable to allow transactions to run in normal mode.

Two steps are needed to restore the system to normal mode:

1. The "missing writes" must be carried out; all updates executed in partitioned mode must be propagated to all copies of the file.

2. The missing write tokens must be located and discarded.

In their paper, Eager and Sevcik describe the bookkeeping necessary to carry out these steps. A transaction's missing write tokens cannot be discarded until all of the missing writes associated with that transaction and with all previously serialized transactions have been carried out. As long as any missing write tokens continue to exist, they are likely to propagate, and therefore the system will have been restored to normal operation only when all missing write tokens have been located and discarded. Eager and Sevcik propose that a systematic network traversal be used for this purpose.

The problem with this technique is that restoration is not a local process. Because tokens created by partitioned-mode transaction can be propagated to an arbitrary extent, the number of sites whose cooperation is needed both to carry out the missing writes and to discard the tokens cannot be bounded in advance. If this technique is viewed as a race between a restoration transaction and token propagation by other transactions, then it is clear that token propagation has a formidable advantage. A transaction may propagate a token by communicating with only a few sites, but the restoration transaction may require the cooperation of every site in the system to make progress. A restoration transaction based on a systematic network traversal might be feasible in a small, administratively centralized database, in which it is possible to shut down the system for periodic maintenance, but it is not feasible for larger, administratively decentralized distributed systems such as one might find in a campus or in a corporation.

In this chapter we propose an alternative technique for restoring the use of normal quorums after a partition has been repaired. Unlike the method of Eager and Sevcik, our method is *local*; each replicated object acts independently to restore its own normal quorums. Not only does local restoration simplify the bookkeeping, it makes restoration more likely to succeed, and probability of success can be analyzed and predicted in advance.

5.3 The General Method

We begin with a few observations on some basic limitations governing the level of service that can be provided in the presence of partitions. We adopt Guttag's terminology [Guttag 78] classifying operations as mutators and observers. A *mutator* is an operation that changes the object's state, such as *Write* for files, and both *Enq* and *Deq* for queues. An *observer* is an operation that returns information about the object's state, such as *Read* for files, and *Deq* for queues. Observers have non-trivial initial quorums, and mutators have non-trivial final quorums. An operation may be both an observer and a mutator (e.g. *Deq*).

If all initial quorums for an observer operation have become unavailable, the ability to execute that operation cannot be restored, because it is not possible to reconstruct enough information to choose subsequent responses. For example, once it is no longer possible to locate an initial *Deq* quorum, then it is no longer possible to reconstruct the items in the queue, and therefore it is not possible to restore the ability to execute *Deq* events. Consequently, the best we can hope for is to facilitate the execution of mutator operations by reducing the sizes of final quorums, possibly by increasing the sizes of initial quorums. In Eager and Sevcik's scheme, if the unanimous *Write* quorum cannot be located, a transaction can still write to a file only a majority *Write* quorum, but the transaction cannot read from the file if a *Read* quorum becomes unavailable, because there is simply not enough information to ascertain the file's current state. Applying this observation to the FIFO queue example, we might be able to restore the ability to execute an *Enq*, but it cannot restore the ability to execute a *Deq*.

5.3.1 A Simple Method

We initially describe a very simple scheme, and successively introduce complications. In this section we discuss a scheme that supports increased availability for arbitrary data types, but that permits transitions in one direction only: from normal to partitioned mode. We then present a more general scheme employing an arbitrary number of modes, and in the final section, we consider how to restore the system to normal operation after a partition is repaired.

Each replicated object provides two sets of quorums: the *normal* quorums and the *partitioned* quorums. These quorums are related by the following constraints:

1. The quorum intersection relations of both sets of quorums must be compatible with the same dependency relation.
2. Every partitioned initial quorum for an event must contain a normal initial quorum for that event.

The important point about these constraints is that although partitioned initial quorums are larger than normal initial quorums, partitioned final quorums may be smaller. Because pure mutators such as *Write* or *Enq* have empty initial quorums, their partitioned quorums are more available than their normal quorums.

The quorums for files in Eager and Sevcik's scheme satisfy these constraints. A file replicated among n DM's has a normal *Read* quorum of one, and a normal *Write* quorum of n. The partitioned quorums for both operations each consist of a majority. Each partitioned initial quorum for *Read* (a majority) contains a normal initial quorum (an individual DM), but the partitioned final quorum for *Write* (a majority) is smaller than the normal final quorum (all DM's).

Transactions operate in one of two *modes*: normal or partitioned. A transaction in normal mode uses the normal quorums for each object on which it operates, while a transaction in

partitioned mode uses the partitioned quorums. Just as for Eager and Sevcik's method, we claim that if all normal-mode transactions are serialized before all partitioned-mode transactions, then all replicated objects continue to satisfy their specifications. The correctness argument is the same. Each entry that has been recorded at a normal final quorum has also been recorded at a partitioned final quorum, and therefore any view constructed by a transaction executing in partitioned mode is certain to include all earlier entries on which it depends.

The requirement that every transaction in normal mode must be serialized before every transaction in partitioned mode is enforced by the following techniques. A flag is associated with each DM; the flag is initially set to *normal*, but it is reset to *partitioned* when the DM's log is read or updated by a transaction in partitioned mode. (Techniques for restoring DM's to normal mode are discussed in a later section.) Transactions observe the following rules:

1. A DM whose flag has been set to *partitioned* rejects all requests from transactions in normal mode.

2. Every timestamp generated by a transaction in partitioned mode is greater than any timestamp generated by a transaction in normal mode.

The timestamp property may be implemented using the high-order bit of each timestamp as a "mode bit" which is set to zero for transactions in normal mode, and to one for transactions in partitioned mode. These rules ensure that transactions running in partitioned mode are serialized after transactions running in normal mode. The first rule ensures that a transaction in normal mode never observes an entry generated by a transaction in partitioned mode, and the second rule ensues that entries generated by partitioned-mode transactions always appear to follow entries generated by normal-mode transactions.

As before, if a normal-mode transaction discovers that it cannot locate a quorum for an operation, the transaction might still be able to make progress by by aborting and restarting

in partitioned mode. As illustrated in the example given below, an object's availability can be enhanced by allowing transactions to choose whether to run in normal or partitioned mode.

An informal correctness argument for replicated objects implemented on top of a transaction system can be summarized as follows. We wish to show that an object's global log is always legal, consisting of a prefix consisting of events executed by normal-mode transactions, followed by a sequence of events executed by partitioned-mode transactions. We argue by induction. Assume that the object state is given by the log $n \circ p$, where n consists entirely of events executed by normal-mode transactions, and p consists entirely of events executed by partitioned-mode transactions. If an event e is executed in partitioned mode, the resulting state is $n \circ p \circ e$. The initial quorum for e intersects the final quorum for every event in both n and p on which it depends, thus $n \circ p \circ e$ is legal. If e is executed in normal mode, the resulting state is $n \circ e \circ p$, because e is given a timestamp earlier than any event in p. Because the initial quorum for e intersects the final quorum for every event in n on which it depends, $n \circ e$ is legal. No event in p can depend on e, because otherwise e could not have located a quorum of normal-mode DM's, thus $n \circ e \circ p$ is legal.

This technique is independent of the object's concurrency control methods. When a transaction attempts to execute an event in normal mode, the concurrency control mechanism schedules it on the basis of the other events executed by uncommitted transactions in normal mode. When a transaction attempts to execute an event in partitioned mode, the concurrency control mechanism schedules it on the basis of the other events executed by uncommitted transactions in both modes.

We remark that when this method is applied to files, the resulting scheme is slightly more flexible than that of Eager and Sevcik, because the use of timestamps supports minority *Write* quorums. In Eager and Sevcik's scheme, transactions in the majority partition may read and write the file, but transactions in a minority partition may read the file but not write it. Our scheme supports an alternative configuration in which the transactions in a minority

partition may write the file, but may not read it.

To illustrate this method, let us examine the implementation of a FIFO queue replicated among three DM's. We use the notation (*m*,*n*) to indicate that an operation has initial quorums consisting of any *m* DM's from a particular set, and final quorums consisting of any *n* DM's from that same set. In normal mode, *Enq* has quorums of (0,2), and *Deq* has quorums of (2,2). In partitioned mode, *Enq* has quorums of (0,1), and *Deq* has quorums of (3,1).

This choice ensures that the *Enq* operation will remain available in both majority and minority partitions, although *Deq* will remain available only in the majority partition. This scheme might be useful in the implementation of highly available spooler, permitting clients to spool files for printing even though the files will not actually be printed until the partitions are rejoined.

The status of the mode flag for each DM is shown at the top of each column. The queue is initially empty, and an item *a* is enqueued at DM1 and DM2 in normal mode with timestamp t1.

DM1	DM2	DM3
Mode: Normal	Mode: Normal	Mode: Normal
t1: [Enq(a);Ok()]	t1: [Enq(a);Ok()]	

The DM's are partitioned into two groups: one consisting of DM1, and the other consisting of DM2 and DM3. A transaction in DM1's partition attempts to enqueue item *b*, discovers it cannot locate a quorum, and enqueues the item at DM1 after restarting in partitioned mode.

Timestamps generated in normal mode are written in lower-case, and those in partitioned mode are written in upper-case.

```
Partitions: {DM1},{DM2,DM3}.

DM1                          DM2                  DM3
Mode: Partitioned            Mode: Normal         Mode: Normal

t1: [Enq(a);Ok()]            t1: [Enq(a);Ok()]
T1: [Enq(b);Ok()]
```

In the other partition transactions operate using the normal quorums. Here, a transaction has enqueued an item *c* and dequeued *a*.

```
Partitions: {DM1},{DM2,DM3}.

DM1                          DM2                  DM3
Mode: Partitioned            Mode: Normal         Mode: Normal

t1: [Enq(a);Ok()]            t1: [Enq(a);Ok()]    t1: [Enq(a);Ok()]
                             t2: [Enq(c);Ok()]    t2: [Enq(c);Ok()]
                             t3: [Deq();Ok(a)]    t3: [Deq();Ok(a)]
T1: [Enq(b);Ok()]
```

Note that the timestamp T1 generated in partitioned mode is later than the timestamp t2 generated in normal mode. After the partitions are rejoined, a transaction in partitioned mode executes a *Deq* by merging the logs from all three DM's, choosing the item to be dequeued, and writing out the resulting view to DM1. Because the partitioned-mode transaction has read the logs at all three DM's, their mode flags have been set to *partitioned*.

```
Partitions: {DM1,DM2,DM3}.

DM1                    DM2                    DM3
Mode: Partitioned      Mode: Partitioned      Mode: Partitioned

t1: [Enq(a);Ok()]      t1: [Enq(a);Ok()]      t1: [Enq(a);Ok()]
t2: [Enq(c);Ok()]      t2: [Enq(c);Ok()]      t2: [Enq(c);Ok()]
t3: [Deq();Ok(a)]      t3: [Deq();Ok(a)]      t3: [Deq();Ok(a)]
T1: [Enq(b);Ok()]
T2: [Deq();Ok(c)]
```

5.3.2 Multiple Levels

In this section we generalize the method described above from two sets of quorums to an an arbitrary number of different quorum sets. These generalizations further enhance availability, and they are needed for the restoration technique to be introduced in the next section. Each object provides a *sequence* of quorum sets, one for each non-negative integer. The quorum set associated with different levels need not (and indeed cannot) be distinct. The integer associated with each quorum set is called its *level*. We adopt the convention that level zero is initially used for normal unpartitioned activity, and that higher level modes are used in the presence of partitions.

Quorum sets are subject to the following constraints, which are direct generalizations of the constraints imposed by the two-level technique.

1. The quorum intersection relation at each level must be compatible with the same dependency relation.

2. Every initial quorum for an event at level $n > 0$ must contain an initial quorum for that event at level $n-1$.

These constraints imply that if a log entry is recorded at a final quorum for level n, then it has also been recorded at a final quorum for all levels $N > n$, but not vice-versa.

Each transaction operates at a fixed level, using the quorums for that level for each object on which it operates. Just as before, we claim that if all level *n-1* transactions are serialized before all level *n* transactions, then all replicated objects continue to satisfy their specifications. Transactions observe the following rules:

1. Each DM has a counter indicating the highest level transaction that has read or updated its log. A DM whose counter has been set to *n* will reject all requests from transactions in lesser modes.

2. Every timestamp generated by a transaction in mode *n* is greater than any timestamp generated by a transaction in mode *n-1*. The timestamp property is implemented by reserving enough high-order bits in each timestamp to indicate the level of the associated transaction.

These rules ensure that transactions running in mode *n* are serialized after transactions running in mode *n-1*. If a transaction cannot locate a quorum for a desired operation, it may restart at a higher level.

For example, consider a file replicated among three DM's that supports the following sequence of quorums.

	Read Quorum	Write Quorum
Level 0	1	3
Level 1	2	2
Level 2	3	1
Level 3	3	1
.	.	.
.	.	.
.	.	.

In this example, levels two and higher are bound to the same quorums. Following a partition, transactions executing in the majority partition can execute *Read* and *Write* operations at level one. A transaction executing in the minority partition has a choice: if it

executes at level zero, it can read the file, but cannot write it, while if it executes at level two or higher, it can write the file, but cannot read it.

5.3.3 Restoring Normal Mode

The basic idea underlying our restoration scheme is the following. Each object has its own binding between levels and quorum sets. Initially, transactions execute at level zero, which is bound to the normal set of quorums. When a partition occurs, some transactions may choose to execute at a higher level, say level *n*, which is bound to a different set of quorums. When the partition is rejoined, the use of the normal set of quorums is restored at a particular object not by lowering its level, but by altering the binding between its current level and its quorum sets. By rebinding level *n* to the normal set of quorums, subsequent transactions executing at level *n* will use the normal quorums for that object, and for all objects that have carried out a similar rebinding. Because the binding between levels and quorum sets is local to each object, restoration is a local process involving only the DM's for the object being restored.

We are now ready to describe the restoration scheme in detail. Restoration has the effect of rebinding level *n* to the quorums for level *n-1*, preserving the constraint that every initial quorum at level *n* include an initial quorum at level *n-1*. Just as in Chapter Four, information about quorums is stored in the object itself. Each DM for an object stores a *quorum assignment table* containing the bindings between each level and its quorum set. Each binding has its own timestamp. The quorums for observing and altering the binding for level *n* are chosen in conjunction with the quorums for the operations at level *n*. Each quorum for a level *n* operation must include a quorum for reading the binding for level *n*.

Restoration is carried out by the following three steps, which must be executed as a transaction:

1. Construct a view containing all level *n* entries by reading from from a level *n* initial quorum for every event.

2. Write out the level *n* entries to a level *n-1* final quorum for every event.

3. Change the binding for level *n* in the quorum assignment table. (This step requires updating the quorum assignment table at a level *n* coquorum for every event.)

Each TM maintains a cached copy of the quorum assignment table. Whenever the TM sends a message to a DM on behalf of a level *n* transaction, it includes the timestamp for level *n*'s binding in the cached table. Whenever a DM receives a message with an out-of-date timestamp, it notifies the TM of the new bindings. The requirement that every quorum for a level *n* operation include a quorum for observing the binding for level *n* ensures that the TM cannot locate a level *n* quorum without being notified that its cache is out of date. When a TM receives such a notification, it updates its cache, and writes out its view to the larger final quorum.

To illustrate this technique, let us review the queue example presented in the previous section. Instead of a mode flag, each DM maintains a counter, initialized to zero, recording the highest level transaction that has observed or updated its log. When the queue is initialized, level zero is bound to *Enq* quorums of (0,2) and *Deq* quorums of (2,2), and level one is bound to *Enq* quorums of (0,1) and *Deq* quorums of (3,1). The absence of explicit bindings for levels higher than one indicates that they are bound to the same quorums as level one. The constraints on quorum inclusion imply that the quorum for reading the level zero bindings consists of any two DM's, and the quorum for reading the level one bindings consists of any single DM.

```
DM1                 DM2                 DM3
Max: 0              Max: 0              Max: 0

t0: Level 0         t0: Level 0         t0: Level 0
Enq = (0,2)         Enq = (0,2)         Enq = (0,2)
Deq = (2,2)         Deq = (2,2)         Deq = (2,2)

t0: Level 1         t0: Level 1         t0: Level 1
Enq = (0,1)         Enq = (0,1)         Enq = (0,1)
Deq = (3,1)         Deq = (3,1)         Deq = (3,1)
```

After the computation described in the previous section has been carried out, the state of the queue is shown in the following picture, where timestamps generated by level zero transactions are written in lower-case, and those generated by level one transactions are written in upper-case.

```
DM1                 DM2                 DM3
Max: 0              Max: 0              Max: 0

t0: Level 0         t0: Level 0         t0: Level 0
Enq = (0,2)         Enq = (0,2)         Enq = (0,2)
Deq = (2,2)         Deq = (2,2)         Deq = (2,2)

t0: Level 1         t0: Level 1         t0: Level 1
Enq = (0,1)         Enq = (0,1)         Enq = (0,1)
Deq = (3,1)         Deq = (3,1)         Deq = (3,1)

t1: [Enq(a);Ok()]   t1: [Enq(a);Ok()]   t1: [Enq(a);Ok()]
t2: [Enq(c);Ok()]   t2: [Enq(c);Ok()]   t2: [Enq(c);Ok()]
t3: [Deq();Ok(a)]   t3: [Deq();Ok(a)]   t3: [Deq();Ok(a)]
T1: [Enq(b);Ok()]
T2: [Deq();Ok(c)]
```

To restore normal operation, it is necessary to rebind level one to the normal set of quorums.

The first step is to collect the level one entries from a level one initial quorum for both *Enq* and *Deq*, here consisting of all three DM's. The level one entries are then written out to a level zero final quorum for both *Enq* and *Deq*, here consisting of any two DM's, say DM1 and DM2. The final step is to update the level one binding at all three DM's.

DM1	DM2	DM3
Max: 0	Max: 0	Max: 0
t0: Level 0	t0: Level 0	t0: Level 0
Enq = (0,2)	Enq = (0,2)	Enq = (0,2)
Deq = (2,2)	Deq = (2,2)	Deq = (2,2)
T3: Level 1	T3: Level 1	T3: Level 1
Enq = (0,2)	Enq = (0,2)	Enq = (0,2)
Deq = (2,2)	Deq = (2,2)	Deq = (2,2)
t0: Level 2	t0: Level 2	t0: Level 2
Enq = (0,1)	Enq = (0,1)	Enq = (0,1)
Deq = (3,1)	Deq = (3,1)	Deq = (3,1)
t1: [Enq(a);Ok()]	t1: [Enq(a);Ok()]	t1: [Enq(a);Ok()]
t2: [Enq(c);Ok()]	t2: [Enq(c);Ok()]	t2: [Enq(c);Ok()]
t3: [Deq();Ok(a)]	t3: [Deq();Ok(a)]	t3: [Deq();Ok(a)]
T1: [Enq(b);Ok()]	T1: [Enq(b);Ok()]	
T2: [Deq();Ok(c)]	T2: [Deq();Ok(c)]	

Now suppose a TM with an out-of-date cache attempts to enqueue an entry on behalf of a level one transaction. When the TM sends an *Enq* entry to a DM, the DM detects that the timestamp for the TM's cache is out of date, and responds with an up to date binding for level one. Upon observing that the new final *Enq* quorums, the TM updates its cache and sends the *Enq* entry to two DM's.

The normal course of events can be summarized as follows. Initially, all transactions use the normal set of quorums for each object. Following a partition, some transactions choose to

execute at a higher level, using a different set of quorums. The higher level gradually propagates through the objects in the system, causing lower level transactions to be restarted at the higher level. Eventually, all transactions execute at the higher level. When the partition is rejoined, each object in the system individually restores its own normal quorums to the current level, and eventually all transactions will be using normal quorums again.

5.4 Discussion

We have demonstrated a way to extend General Quorum Consensus to provide a higher level of availability in the presence of partitions. Transactions in distinct partitions may use different quorums for the same object. Each transaction chooses the quorum best suited to its current partition. The method preserves consistency, it is independent of any particular concurrency control method, and it imposes negligible costs when it is not used. Unlike some methods discussed below, it does not require explicit partition detection, static transaction classes, or undoing of committed transactions. The increased availability provided by our method is purchased at the cost of external consistency: events that are widely separated in time occurring in distinct partitions may be serialized in an order different from the order in which they are observed to occur from outside the system.

Our scheme was originally motivated by consideration of a file replication method proposed by Eager and Sevcik [Eager 83]. Both our scheme and their scheme increase availability in a manner that preserves serializability but not external consistency. Both methods incur a negligible overhead when not used. Our method makes two improvements: it is applicable to arbitrary data types, not just to files, and it provides a more practical way to restore normal operation once partitions are repaired.

As noted in the previous chapter, the *available copies* replication algorithm for databases [Goodman 83] fails to preserve consistency in the presence of partitions. If a partition

should occur, each partition's directories would mark the others as unavailable and continue processing, causing the database states in the two partitions to diverge. The difficulty here is that the method used to manage the replicated directory does not tolerate communication failures. If a more robust algorithm were used to manage the directory, the result would resemble the single-level reconfiguration method described in the previous chapter.

SDD-1 [Hammer 80] is another example of a replication method that does not tolerate partitions. If a site is up, then it is operating normally and able to communicate with all other sites, and if it is down, then it cannot communicate with any other sites. A message intended for a site that is down may be stored for later delivery at a replicated FIFO queue associated with that site. The correctness of the replicated queue (as well for other components of SDD-1) relies on the assumption that a site that does not respond to a message is down. This assumption is invalid in the presence of partitions and similar communication failures.

The Locus distributed operation system [Popek 81] implements replicated files and directories that continue to provide service during partitions. Inconsistencies between replicas in different partitions are allowed to develop, but application-dependent measures are used to reconcile them when the partitions are rejoined. For example, a transaction in one partition may observe that file *a* exists but file *b* does not, while a transaction in another partition observes that *b* exists but *a* does not. Both files might be observed to exist after the partitions are reconciled. The file system's state is internally consistent, but it is not compatible with the observations made by all committed transactions. This approach is suitable for some applications, but not for others.

Wright [Wright 83] has developed a method for making progress in the presence of partitions that requires transactions to be placed in classes with fixed read and write sets. A static analysis is used to determine which transactions classes can run in the presence of particular partitions. Cycles of dependency are broken by allowing transactions to read

older versions of files. Unlike our scheme, in which a partition is detected implicitly when a DM is unable to locate a quorum, this scheme requires an explicit mechanism for determining the extent of a transaction's partition. Wright also discusses *optimistic* methods, in which transactions are allowed to execute to completion. If inconsistencies are revealed by an after-the-fact analysis, committed transactions are undone to restore a consistent state. Like the approach taken by Locus, transaction classes and optimistic methods are suitable for some applications, but not for others.

Chapter Six

Conclusions

This thesis has proposed new techniques for managing replicated data in a distributed system. We assume that replication should be transparent: its only effect should be to enhance availability in the presence of failures. Our model of computation admits two kinds of failures: communication interruptions and site crashes. We assume that these failures can be detected, but not necessarily distinguished. Our techniques make use of an underlying transaction system to help preserve invariant properties of the replicated data.

We have proposed techniques to address four problems associated with replication: the representation and manipulation of replicated data, concurrency control, on-the-fly reconfiguration, and techniques for increasing availability in the presence of partitions. A summary of our results follows.

- We introduce a new type-dependent method for representing and managing replicated data. A replicated data object is not represented by a collection of copies; instead each site keeps a timestamped log of the operations that have been applied to the object. By taking advantage of type-dependent properties in the replication method itself, our method can realize a wider range of availability properties than existing replication methods (Chapter Two).

- An analysis of the algebraic structure of the data type is used to derive necessary and sufficient constraints on correct implementations, and hence on realizable availability properties. This analysis technique is both systematic and general. It is applicable to arbitrary data types, including types having partial and non-deterministic operations (Chapter Two).

- Although our method permits replication and concurrency control to be implemented independently, a higher level of concurrency can be supported if the concurrency control method is integrated with the replication method. We first propose a simple and efficient concurrency control method based on predefined operation conflicts. This method is optimal for the amount of information it uses: no concurrency control method based exclusively on predefined conflicts can support a higher level of concurrency. Nevertheless, the method has two disadvantages: because it does not take advantage of

state information, it is inherently limited in the level of concurrency it can support, and it provides poor support for partial operations (Chapter Three).

- To remedy these shortcomings, we also propose a more powerful concurrency control method employing state information. This method is general: it can be used to implement any concurrency control scheme that preserves serializability, but at the potential cost of increased message traffic and additional constraints on availability (Chapter Three).

- We present a reconfiguration technique that allows the availability properties of replicated data to be changed dynamically. The ability to reconfigure incurs a negligible cost when it is not used. When the object is actually reconfigured, the technique imposes a cost in the form of a temporary period of increased message traffic and reduced availability. A replicated reference counting scheme is introduced to discard information rendered obsolete by reconfiguration (Chapter Four).

- We also propose an extension to our method that increases availability in the presence of partitions. This technique preserves serializability, but not external consistency. It does not require explicit partition detection or static transaction classes, it is independent of the concurrency control method, and it imposes negligible costs when it is not used (Chapter Five).

Perhaps the single most important characteristic that distinguishes our work from most earlier work in this field is that our techniques take advantage of type-specific properties of the replicated data. The traditional approach to replication is 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, we have seen that it places unnecessary constraints on availability, concurrency, flexibility of reconfiguration, and tolerance of partitions. The major contribution of this thesis has been the development of systematic and general techniques for taking advantage of type information to derive more effective replication methods.

References

[Alsberg 76]
Alsberg, P. A., and Day, J. D., "A principle for resilient sharing of distributed resources," in *Proceedings, 2nd International Conference on Software Engineering*, October 1976.

[Bernstein 81]
Bernstein, P. A., and Goodman, N., "A survey of techniques for synchronization and recovery in decentralized computer systems," **ACM Computing Surveys 13,2**, June 1981, 185-222.

[Bernstein 81]
Bernstein, P., Goodman N., and Lai, M.-Y., Two-part proof schema for database concurrency control. In *proceedings of the Fifth Berkeley Workshop on Distributed Data Management and Computer networks*, February, 1981.

[Bernstein 83]
Bernstein, P. A., and Goodman, N., "The failure and recovery problem for replicated databases," Technical Report, Harvard University Center for Research in Computing Technology

[Birrel 81]
Birrel, A. D., Levin, R., Needham, R., and Schroeder, M., "Grapevine: an Exercise in Distributed Computing," *Communications ACM 25,14*, April 1982, 260-274.

[Bloom 75]
Bloom, T., "Evaluating Synchronization Mechanisms," In *Proceedings of the Seventh Symposium on Operating Systems Principles*, December, 1979.

[Chan 82]
Chan, A., Fox, S., Lin, W. T., Nori, A., and Ries, D., "The implementation of an integrated concurrency control and recovery scheme," in *Proceedings 1982 SIGMOD Conference*.

[Cox 65]
Cox, D. R., and Miller, H. D., **The Theory of Stochastic Processes**, John Wiley and Sons, New York, 1965.

[Daniels 83]
Daniels, D., and Spector, A., "An Algorithm for Replicated Directories," in *Proceedings, 2nd Annual Symposium on Principles of Distributed Computing*, August, 1983.

[Dubourdieu 82]
Dubourdieu D. J., "Implementation of Distributed Transactions," in *Proceedings, 1982 Berkeley Workshop on Distributed Data Management and Computer Networks* 81-94.

[Eager 83]
Eager, D., L., and Sevcik, K. C., "Achieving robustness in distributed database systems," *ACM Transactions on Database Systems 8,3*, September 1983,354-381, September 1983.

[Eswaren 76]
Eswaren, 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*, November 1976, 624-633.

[Gifford 79]
Gifford, D. K., "Weighted Voting for Replicated Data." In *Proceedings of the Seventh Symposium on Operating Systems Principles.* December 1979.

[Gifford 82]
Gifford, D. K., "Information Storage in a Decentralized Computer System", Technical Report CSL-81-8, Xerox Corporation, March 1982.

[Goodman 83]
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.

[Gray 78]
Gray, J. N., Notes on database operating systems. in *Operating Systems: an advanced course*, Vol. 60, Lecture Notes in Computer Science, Springer-Verlag, New York, 1978, 393-481.

[Guttag 78]
Guttag, J. V., and Horning, J. J., "The Algebraic Specification of Abstract Data Types," *Acta Informatica 10*, 1978, 27-52.

[Hammer 80]
Hammer, M. M., and Shipman D. W. "Reliability Mechanisms in SDD-1, a System for Distributed Databases", *ACM Transactions on Database Systems 5,4*, December 1980, 431-466.

[Johnson 75]
Johnson, P. R., and Thomas, R. H., "The maintenance of duplicate databases," Network Working Group RFC 677 NIC 31507, January 1975.

[Kohler 81]
"A survey of techniques for synchronization and recovery in decentralized computer systems." **ACM Computing Surveys 13,2**, June 1981, 149-185.

[Korth 81]
Korth, H. F., A deadlock-free, variable granularity locking protocol. In *proceedings of the Fifth Berkeley Workshop on Distributed Data Management and Computer networks*, February, 1981.

[Kung 81]
Kung, H. T., and Robinson, J. T., "On optimistic methods for concurrency control," *ACM Transactions on Database Systems 6, 2*, June 1981, 213-226.

[Lamport 78]
Lamport, L., "Time, clocks, and the ordering of events in a distributed system," *Communications ACM 21,7*, July 1978, 558-565.

[Lamport 78a]
Lamport, L., "The implementation of Reliable Distributed Multiprocess Systems," *Computer Networks 2*, 1978, 95-114.

[Lamport 82]
Lamport, L., Shostak, R., and Marshall, P., "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems 4,3*, July 1982, 382-401.

[Lampson 79]
Lampson, B., "Atomic transactions", Distributed Systems: Architecture and Implementation, *Lecture Notes in Computer Science 105*, Goos and Hartmanis editors, Springer-Verlag, Berlin, 1981, 246-265.

[Liskov 82]
Liskov, B. H., and Scheifler R. W., "Guardians and Actions: Linguistic Support for Robust, Distributed Programs." In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, January 1982, 7-19.

[Minoura 79]
Minoura, T., Owicki, S., and Wiederhold, G., "True-copy token scheme for a distributed database system," Technical Report 83-60-1, Oregon State University Computer Science Dept., July 1983.

[Minoura 79]
Minoura, T., Owicki, S., and Wiederhold, G., "Consistent Distributed Database State Maintenance," Technical Report 83-60-2, Oregon State University Computer Science Dept., July 1983.

[Minoura 82]
Minoura, T., and Wiederhold, G., "Resilient extended true-copy token scheme for a distributed database system," *IEEE Transactions on Software Engineering 8, 3*, May 1982, 173-188.

[Moss 81]
Moss, J. E. B., "Nested Transactions: An Approach to Reliable Distributed Computing," Technical Report MIT/LCS/TR-260, MIT, 1981.

[Oppen 81]
Oppen, D., Dalal, Y, K., "The clearinghouse: a decentralized agent for locating named objects in a distributed environment," Xerox technical report OPD-T8103, October, 1981.

[Papadimitriou 79]
Papadimitriou, C.H., "The serializability of concurrent database updates", *Journal of the ACM 26,4*, October 1979, 631-653.

[Popek 81]
Popek, G. J., Walker, B., Chow, J., Edwards, D., Kline, C. Rudisin, G., and Thiel, G., "Locus: a network transparent high reliability distributed system," In *Proceedings of the Eighth Symposium on Operating Systems Principles*, December 1981.

[Reed 78]
Reed, D. "Naming and synchronization in a decentralized computer system," Technical Report MIT/LCS/TR-205, MIT, 1978.

[Reed 80]
Reed, D. P., and Svobodova, L., SWALLOW: a distributed data storage system for a local network. In *Proceedings of the International Workshop on Local Networks.* Zurich Switzerland, August 1980.

[Reed 83]
Reed, D. "Implementing atomic actions on decentralized data," *ACM Transactions on Computer Systems 1,1*, February 1983, 3-23.

[Schlichting 81]
Schlichting R. D., and Schneider, F. B., "An approach to designing fault-tolerant computing systems," *ACM Transactions on Computer Systems 1,3*, August 1983, 222-238.

[Skeen 82]
Skeen, M. D., "Crash Recovery in a Distributed Database System," Memorandum No. UCB/ERL M82/45, University of California, Berkeley, May 1982.

[Stonebreaker 79]
Stonebreaker, M., "Concurrency control and consistency of multiple copies of data in distributed INGRES," *IEEE Transactions on Software Engineering* 5,3, May 1979, 188-194.

[Thomas 78]
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.

[Weihl 83]
Weihl, W., and Liskov, B., "Specification and implementation of resilient, atomic data types", In *Proceedings, SIGPLAN Symposium on Programming Language Issues in Software Systems*, June, 1983.

[Weihl 83a]
Weihl, W., "Data-Dependent concurrency control and recovery," in *Proceedings, 2nd Annual Symposium on Principles of Distributed Computing*, August, 1983.

[Weihl 84]
Weihl, W., "Specification and implementation of atomic data types," Ph.D. Thesis, in preparation.

[Wright 83]
Wright, D., D., "Managing Distributed Databases in Partitioned Networks," Technical Report 83-572, Cornell University, Ithaca, New York, September 1983.