A PRACTICABLE CONSISTENCY SCHEME
FOR
FILE REPLICATION

Sebnem Baydere

Department of Computer Science
University College London

A thesis submitted for the degree of Doctor of Philosophy
October 1990

Abstract

Distributed systems provide the opportunity for fault tolerance through replication. This dissertation describes the design and performance of a novel consistency scheme which balances the cost and benefits of file replication. The scheme features some characteristics that have an effect upon steady-state and continuous availability, and the correctness in the face of network partitions with a small number of copies; especially two. The work has proceeded along five fronts; characterization of the consistency problem with a small number of replicas; a hybrid design proposition; a series of comparative statistical analyses of availability in partition-free networks with extension to a simple partitioning case; a study of reliability and its resilience to configuration changes in partitioned networks through simulation, and a discussion of the practicality and performance issues including some proposals for reducing communication cost of the control operations. Apart from the algorithm presented in Chapter Three, the original contribution of the work is threefold; a statistical analysis of availability with an extension of partitioning case for which neither statistical nor real-time analysis has been found in the literature. This analysis has shown the importance of the two-copies case. An analysis of reliability including the effect of network partitions and resilience to configuration changes in Chapter Five is also original and opened further areas in the field through this new resilience property. Chapter Six contains a new algorithm for reducing the communication cost of history operations. The introductory chapter presents an approach for comparing consistency schemes through their effectiveness while, after a general summary, the concluding chapter details further work and the future of file replication in general purpose computing environments.

Acknowledgements

I wish to thank Mr. Benjamin Bacarisse, for his encouragement, continuous support and supervision during the course of my research. Without his guidance I would not have been able to carry out this study.

I am indebted to Prof. Steve Wilbur who has given valuable advice at the initial stages of the research and spent a great amount of time reading and commenting during the writing phase of this dissertation.

I am also grateful to Prof. M. Sahinoglu and Karen Paliwoda who helped with the statistical analysis and to Dr. Ken Moody for his constructive comments on the thesis.

I would like to make a special note of thanks to my husband for the support he has given me during the period of my research.

Throughout my research I have been sponsored by the Government of Turkey to which I am also grateful.

Table of Contents

List of Figures

List of Tables

Chapter One

Introduction

Distributed systems provide the opportunity to improve the fault tolerance of data through replication. It is often desirable to have multiple copies mainly for applications where interruptions of service due to node crashes or communication link failures cannot be tolerated and the complexity of the replication system can be justified by the unacceptable cost of failure. However, the designers of general purpose distributed systems have concentrated mainly on the advantages of data *sharing* and efficient remote access rather than on high availability through replication [1].

Several projects are underway to create a general-purpose computing and information processing environment that will include hundreds of self-contained workstations. Replicated file systems which offer the desired availability can only become commonplace in these environments if the benefits of fault tolerance can be balanced against the costs and complexities introduced by replication. For this technology to be adopted in a wider range of applications, the following three main design criteria should be considered: the storage cost of replicating files must be kept low, the communication cost inherent in the replicated system must be kept within acceptable bounds and the mechanisms provided for managing physical copies must be practicable and flexible in the sense that they allow control over the level of reliability required for different sets of files. In addition, replication is required to be transparent; that is its only observable affect is to make the data more available.

The basic aspect of replication management is to guarantee that there is no logical conflict in the user's view. In other words, when the data is replicated accesses to it must be managed so as to maintain consistency. The exact nature of consistency changes from application to application. This will be discussed in more detail below. If updates may be concurrent the locking protocols [2, 3] used to ensure serialization to prevent physical conflicts may also provide the required consistency control. However, these two are separate concerns; consistency problems are inevitable consequences of replication whereas the concurrency problem may arise in any concurrent or pseudo-concurrent environment.

All schemes developed so far that provide users with a consistent view of replicated data are based on one of two basic principles: The simplest principle is *read any/write all.* Unfortunately, this principle improves the availability of the file only for read operations by reducing the availability for write operations. In 1984, Bernstein and Goodman refined the principle giving a scheme that reads from *any available* copy and writes to *all available* copies [4]. This *available copies* scheme configures out failed nodes from the system and configures them back in when they recover; so that, in effect, the algorithm is *read any/write all.* This method gives optimal availability provided the underlying network never becomes partitioned into more than one independently functioning set of nodes. If the network does become partitioned, this algorithm fails to preserve consistency.

The second basic principle is *read some/write some.* The algorithms which follow this principle use a quorum-consensus approach, each node votes for participation in a read or write operation. In order to read a file a read quorum must be collected. Similarly, in order to write a file a write quorum must be collected. This principle was first used by Thomas in 1978. He suggested a simple majority voting scheme [5]. In 1979, Gifford proposed assigning different weights to different copies and having different read and write quorums [6]. Recently, many different variations to these basic schemes have been suggested [7, 8, 9, 10]. These schemes are discussed in Chapter Two.

The principal disadvantage of voting schemes is that at least three copies of the data are required to give higher availability than a single copy. Five copies are needed to improve availability further. This is a significant storage cost compared to the available copies method that gives considerably improved availability with only two, but which cannot continue to work if the network becomes partitioned. These claims are justified in Chapter Four.

1.1 Objectives of the Thesis

There are many possible approaches to the problem of consistency control in replication. This dissertation investigates a design in which the storage cost of replicated files as well as the gain in availability is considered. Such a low-cost implementation would be suitable for a wide range of applications in general-purpose computing environments. The design focuses on providing high availability with a small number of copies (especially two) and on the correctness of the algorithm in the face of partitions. Some implementation techniques are also described to enhance the performance beyond that of the basic design. The first objective of the work is thus to show that going from a single copy to two copies results in a greater improvement in availability than going from two to three copies or beyond.

The second objective is to compare the network and system architecture assumed by the various algorithms to determine their practicability. This comparison also includes an examination of the facilities provided for reconfiguration (changing the location of copies and altering the degree of replication).

The third objective is to characterize the effect of network partitions on the accessibility of replicated files. Common measures of accessibility include *availability*, which is the steady-state probability that the file is accessible at any given moment, and *reliability*, which is the probability that a replicated file will remain continuously accessible over a given period of time. This objective has proved to be more difficult to satisfy mainly because of the problem of adequately generalizing the characteristics of partitioning. A detailed theoretical analysis of accessibility has been done for partition free systems and this analysis has been extended to include a simple case of partitioning. This partitioning analysis has been carried out through simulation, and the behavior of the proposed algorithm and the related algorithms have been analyzed in some typical topologies. Some interesting results have been obtained concerning the sensitivity of different algorithms to changes to the network topology and copy placement. The proposed design was presented at the *IEEE COMPCON'*89 conference in San Fransisco [11] and the original work on the effect of partitions on reliability of replicated files will be presented in November 1990 at the *IEEE Workshop on Management of Replicated Data* in Houston[12]).

1.2 Outline

The remainder of this chapter presents a model of a distributed environment, the underlying communication medium and the abstract definition of the file system including the level at which replication is introduced, before outlining the consistency problem of replicated files. Later some alternative ways of building replication control algorithms into the file system are discussed together with the effectiveness of replication in terms of storage cost and abstract performance measures such as availability and reliability.

Chapter Two contains a general survey of consistency control schemes examining their behavior, requirements and effectiveness when the degree of replication is low — typically two or three.

Chapter Three introduces a new low-cost hybrid algorithm called *reliable histories* for maintaining consistency of replicated files in applications where the storage cost must be kept down.

Chapter Four is the first of the two analysis chapters. First the steady-state availability of a replicated file is analyzed using two different techniques: k-out-of-n reliability theory and Markov processes. The analysis focuses on the minimum number of copies and processing nodes required before the *reliable histories* algorithm provides better availability than other algorithms. Secondly, the management of replicated copies in a network that may become partitioned is examined; and the resiliency of various consistency schemes to random copy placement and network topology is investigated.

Chapter Five contains an original analysis of the reliability of a replicated file both in partition-free and partitioned networks. The reliability offered by various consistency schemes is compared using different failure models and the comparison is extended to include the effect of partitions in various topologies. Since the technique known as *regeneration* affects reliability (not availability), its integration with the *reliable histories* algorithm is also analyzed in this chapter.

Chapter Six concerns the performance and the practicality of the *reliable histories* algorithm. It focuses on the number of network operations inherited by the algorithm and proposes an algorithm called *range*, for reducing the cost of history operations.

Chapter Seven includes a summary of the basic results obtained from the analytical models and simulation. This is followed by some suggestions for future work including the investigation of the interfaces required by users and system administrators. The benefits of added dynamicity through calculation of overall reliability are also discussed.

1.3 Distributed System Models

This section summarizes the models for building a distributed system and discusses the model on which the thesis has been developed.

Many different models have been suggested. Tanenbaum states that these models can be grouped into three general categories [13]: The first model consists of a number of minicomputers each with multiple users. Each user logs onto one machine with remote access to other machines. This system is similar to a central time-sharing machine.

In the second category each user has a single workstation usually equipped with processor, memory and a disk. This system becomes distributed when it supports a single global file system so that the data can be accessed regardless their location.

The third category is an evolutionary step. All processors are kept in a pool and allocated upon request by the clients. When the job is completed, allocated processors return to the available pool. This model might become widespread when the CPU's become much cheaper. However, there have been some attempts to combine the second and third models providing each user with a workstation in addition to the processor pool for general use. An example of this type is the Ameoba Operating System [14].

Systems consisting of workstations (called processing nodes throughout the dissertation) connected by fast local area networks are becoming widespread. These systems offer a general purpose distributed computing environment for a large number of applications. The possibility of connecting a large number of processing nodes makes them suitable for replicating objects such as files, replication histories, etc. The replication control protocol which the dissertation presents and analyzes is designed for an environment in which a large number of processing nodes are spread across a series of local area networks connected by bridges. An example topology is illustrated in Figure 1.1.

1.4 Replicated File System

We divide the criteria used to compare replicated file systems into two groups. The first group determine the *efficiency* of the system. This is the cost of its operation which can be measured in terms of communication delay and the cost of extra storage required.

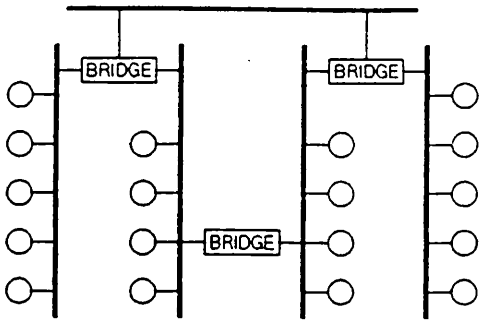


Figure 1.1. *A workstation environment (Topology-1)*

The efficiency is mainly determined by the environment and the features related to the environment such as the resources provided by the distributed system model, the underlying communication medium and the failure to repair ratio of the individual components of the network. The second group determine the *effectiveness* of the replication system. This is predominantly a property of the consistency control algorithm. Effectiveness measures the accessibility of a file (its availability and reliability) together with other abstract properties of the algorithm, such as any assumptions made for its operability and correctness. This includes the failure modes of the network that can be tolerated and whether individual components share the same view of the status of the other components or not.

The effectiveness of a replication system is determined by the control protocol used to manage the replication. Sometimes effectiveness may trade off efficiency and thereby the performance of the whole system. The reverse is also true to a lesser extent.

The aim of this section is to create a general view about the distributed file system model in which the proposed design can perform efficiently.

When connecting two or more distinct systems together, the first issue that must be faced is how to merge the file systems. In distributed file systems three approaches have been tried [13]. In the first approach file systems are not merged. Access to a remote file can only be done by running special file transfer protocols that copy the remote files to the local machine. This approach has been used in early designs and cannot provide replication transparently.

The next step towards a DFS is to have adjoining file systems. In this approach, programs on one machine can open files on another machine by providing a path name which determines where the file is located. This is either done by creating a virtual superdirectory above the root directories of all the connected machines (as in the New-Castle Connection [15] and Netix [16]) or by providing a remote mount operation (as in Sun's NFS [17]). Replication can only be employed statically since the operating system cannot move files around among nodes by itself.

The third approach is the distributed operating systems approach: having a single global file system visible from all processing nodes. This approach allows the operating system to move files around among nodes. The system can maintain replicated copies of files [18,19,20,21,22,23,24].

Sturgis has grouped the basic issues that DFS designers are faced into five categories [25]: communication primitives required, naming and protection, resource management, choosing the services to be provided and fault tolerance. It is the last problem for which replication is a solution.

The replicated file system model runs on a cooperating set of processing nodes which together create the illusion of a single logical file store. The file system data is distributed across many servers in order to get the benefit of a multi-machine environment without losing transparency. A file is modeled as a finite sequence of bytes which can be referred to by a unique file ID. The create operation introduces a new file ID that refers to an empty file. The data referred to is accessed and modified only by read and write operations. Read is used to return an arbitrary, contiguous sub-sequence of the file's bytes, while write is used to replace such a sub-sequence with any other byte sequence.

Once the data in a file is no longer needed, the application can delete the file ID, so that the file system may reclaim the space occupied by the file.

In this light we may now define a replicated file as a set of file copies, each one implemented on a different node in the distributed system.

A replicated file system presents applications with the abstraction of a logical file consisting of a sequence of bytes and identified by a unique identifier. In the thesis, *data* and *file* are used synonymously with the term *logical file*. Logical files are implemented by a set of physical files each holding a complete copy of the file and each residing at a single distinct processing node. Both the terms *copy* and *replica* will stand for a full copy of the file. The degree of replication is defined as the number of the file copies. The files are created, accessed for read or write and deleted by means of logical operations defined on them. A replicated file can have different active versions at one time as a result of failures and repairs of the processing nodes holding them. A *read* on the file will return the current version and a *write* is assumed to be an update on the current version.

The multiple copies of a replicated file are managed by a replication method. This is an algorithm for managing the distributed copies of the files so that its functional behavior is equivalent to that of a file having only a single copy. This property is known as *one-copy serializability* [26]. Consistency problems can arise from two different sources in a distributed environment.

1) Consistency in the face of failures: the data needs to be correct. Incorrect behavior should not occur as a result of system failures such as node crashes, network partitions or timing anomalies.

2) Consistency in the face of concurrent updates: when two or more accesses to the data run simultaneously, it is necessary to ensure that incorrect behavior can not occur as a result of concurrent access by multiple users.

A replication method may address these problems independently. Consistency constraints are defined to ensure that the data meets the above conditions; at one level a standard concurrency-control protocol synchronizes access to the individual components and at a higher level, a replica-management protocol reconstructs the file's consistent state from its distributed copies without concern for concurrency. This distinction is made in order to discuss the problems separately. These problems have a lot in common and it may be difficult to distinguish them in practice. The definition of consistency given in Section 1.4.2 will justify why the concurrency control protocol is considered to be the lower of the two.

1.4.1 Concurrency Control Problem

This section summarizes the protocol required in the lower layer if the updates may be concurrent. This problem has been actively investigated within the environment of centralized and distributed databases in recent years. Concurrency control algorithms within centralized and distributed environments are surveyed by Bernstein *et al* [26] and Kohler [27].

In a concurrency control protocol, logical operations are composed of a series of accesses, called *transactions*, that change the state of the system from one consistent state to another. There are two possible anomalies that are to be considered when the transactions are running: updates might be lost or the retrieval might be inconsistent because of interleaved access to the data. The correctness of a concurrency control algorithm is defined relative to users' expectations. Bernstein defined two correctness criteria regarding the above anomalies [26]:

a) Users expect that each transaction submitted to the system will eventually be executed.

b) Users expect the computation performed by each operation to be the same whether it executes alone in the system or in parallel with others.

In order to satisfy these requirements all concurrent operations are required to be *atomic* which means *all−or−nothing* [28]. An atomic operation would only modify the file if it is completed successfully, otherwise has no effect on the file. Atomic commitment protocols are discussed by Gray *et al* [29] and Hammer *et al* [30].

A concurrency control protocol must ensure that concurrent execution of a set of transactions, where requests belonging to different transactions are interleaved, produces the same result as if those transactions were executed serially. These transactions are said to be *serializable.* The seminal paper on serializability theory was written by Papadimitriou [31].

There are two synchronization problems that the protocol should consider separately: read-read and read-write synchronization. Many different mechanisms have been proposed. The three primary mechanisms are two-phase locking [32, 2], timestamp ordering[33] and so called optimistic methods [34].

The two-phase locking method synchronizes reads and writes by explicitly detecting and preventing conflicts between concurrent operations. Before reading a file a transaction must own a read-lock on it, likewise a write-lock must be obtained before writing. The ownership of locks is governed by two rules:

1) Different transactions cannot simultaneously own *conflicting locks.*

2) Additional locks may never be obtained once a transaction surrenders ownership of a lock.

The definition of a conflicting lock depends on the type of synchronization being performed. For read-write synchronization two locks conflict if both are on the same data and one is a read-lock while the other is a write-lock. For write-write synchronization, two locks conflict if they lock on the same data and both are write-locks.

Timestamp ordering is a technique whereby a serialization order is selected a priori and transaction execution is forced to obey this order. Each transaction is assigned a unique timestamp and conflicting operations must be processed in this timestamp order. The definition of conflicting operations is the same as for two-phase locking.

The concurrency control protocols based on commit protocols are intended for applications where reads predominate. They are poorly suited for applications such as ticket reservation systems where write operations occur frequently. Herlihy proposed optimistic concurrency protocols [35] for the applications where write operations predominate. A concurrency protocol is optimistic if it allows transactions to execute without synchronization, relying on commit-time validation to ensure serializability.

Since this dissertation investigates a consistency control scheme without being concerned with the details of concurrency control protocols, the synchronization techniques will not be discussed further. Besides the above references, interested parties can refer to Bennett [36] and Gelenbe [37].

1.4.2 Consistency Control Problem

The work presented in this dissertation relies on the following definition:

Definition 1.1. Let a , b be two distinct consecutive operations on a replicated file, f, satisfying $a \rightarrow b$ where "$\rightarrow$" is the *happened before* relation which defines an arbitrary total ordering of the events in a distributed multiprocess system as an extension to their partial ordering [38]. Let $U_i \equiv R_i \cup C_i$ is the set of up-to-date copies of f, after the operation i is completed, where R_i is the set of up-to-date copies which has directly accepted the operation i and C_i is the set of up-to-date copies which has become up-to-date by copying from R_i. Consistency is preserved if and only if $p \equiv p_1 \Rightarrow p_2$ is true where p_1 and p_2 are the following propositions.

p_1: a has succeeded on a set of physical copies, U_a, and b is applied to R_b

p_2: $R_b \subseteq U_a$

The result of distinct operations, a and b, that run concurrently is undefined. The term *concurrent* refer to Lamport's definition [38]: two distinct operations a and b are said to be concurrent if $a \rightarrow$b and $b \rightarrow a$.

It is the responsibility of a consistency control protocol to satisfy the integrity constraints explained as the first form of consistency in Section 1.4. These constraints assure the correctness of the consistency protocol which guarantees that incorrect behavior should not occur as a result of network failures such as node crashes, network partitions or timing anomalies. It is the job of the consistency scheme to coordinate the accesses and updates to the file copies so that clients of the replicated file system see a consistent view of the file. That is, any client that reads a file after a write operation has succeeded will see the data as it was left by the write operation.

1.4.3 Communication System

This section describes the assumptions on which the underlying communication system is based. There is server software in each node which implements a set of operations that can be invoked over the network. Individual processing nodes in the system are assumed to provide this abstraction of a file through locally connected hardware. The local connection is important since it allows to assume that the success or failure of a file operation can be determined by the local file system. In contrast, it is assumed that nodes connected by the data network can only determine the outcome of an operation performed remotely by another node by the arrival of a message from the remote node. Messages may be lost in transit but we assume that corrupted messages are detected and removed by the communications software. In particular, we assume that the network may become partitioned. Failed components such as nodes, bridges, etc. can recover spontaneously or because of system maintenance.

The DFS sending a message to another processing node is not concerned with the low level protocols used for transmission. Its only concern is the message has been acted upon and the ensuing results from its operation. It is assumed that in the communication layer a transport level protocol will provide reliable error free communication between the nodes. In order to provide a uniform and easily understood abstraction, an RPC mechanism that offers at-most-once semantics [39,40,41,42] is likely to be the best communication protocol for the system.

The characteristics of the communication medium have a major effect on the performance. Many of the low–level operations required to support replication would benefit from a multicast request-response mechanism. [43] If the underlying communication system uses a broadcast link level protocol, the cost of such a mechanism is a function of the number of *replies* required from a request, not the number of servers to which the request was sent, nor the size of the request parameters.

1.4.4 File vs Block Level Replication

Some system designers choose to introduce the replication at block level while others prefer to do so at file level [44]. Many studies [45,46,47] of distributed file systems (including my own results of the actual performance of NFS — Sun Microsystem's Network File System) have shown that most file accesses are whole-file transfers. An analysis of file access patterns in the UNIX [1] Operating System has been done by Ousterhout [45]. This reveals that more than 90% percent of all files processed sequentially and more than two thirds of all file accesses are whole-file transfers. These figures also show that while operations on small files predominate, large files account for almost 20% of all file accesses. Table 1.1 presents the results from this study together with the results of my own study of network file accesses in the last column. The values presented show percentage of the accesses. For example, first row is the percentage of whole file read transfers of all read-only accesses etc.

This suggests that in some application environments, whole-file replication might be more advantageous and more practical than block-level replication. Therefore the thesis is based on the replication of whole files.

[1] UNIX is a trademark of Bell Laboratories.

	System-1	*System-2*	*System-3*	*System-4*
All read accesses	69%	63%	70%	75%
All write accesses	82%	81%	85%	90%
Sequential read-only accesses	92%	91%	93%	90%
Sequential write-only accesses	97%	96%	98%	98%
Sequential read-write accesses	19%	21%	35%	

Table 1.1 — *Percentage of accesses that represent whole-file transfers.*

1.4.5 Building Replication into the File System

Even if it is conceptually simple, building replication into the file system while trying to preserve file system semantics is very complicated. In most cases the file system is part of the operating system kernel. In these systems replication can be implemented on top of the operating system as a set of library procedures as in Figure 1.2(a) or can be moved into the operating system kernel. In the first case the implementor must provide an interface that preserves the semantics of the original file system using only available system services. An entire replicated file system must be built on top of the original file system. The second case is more complicated because it requires the modification of the operating system kernel.

In order that replicated file systems can become commonplace in general purpose computing environments, they should provide fault tolerance efficiently as an extension to simple file systems.

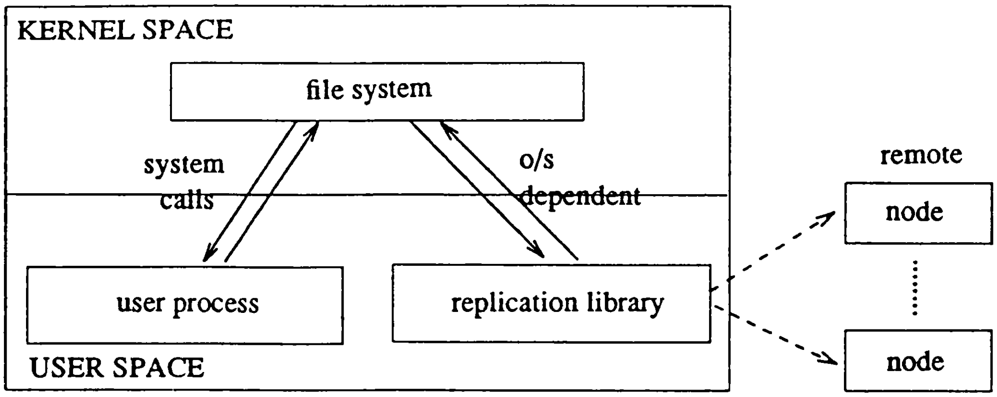


Figure 1.2(a). *Building replication on top of the file system*

One suggestion for implementing replication at the block level is a reliable device [44] which appears to the file system as an ordinary block-structured device but implemented as a set of server processes on several nodes. Because it presents the same simple interface as an ordinary device, it provides replication while leaving the operating system kernel and the file system unchanged. This approach has the advantage that existing programs can operate on replicated files without modification.

In the case of a conventional operating system where the file system is part of the operating system kernel, it has been suggested by Carroll [44] that a device driver stub could receive requests for access from the file system and forward those requests to a server which would perform the data access and consistency control algorithms. Such a scheme is illustrated in Figure 1.2(b).

In this system, a user-state process makes a file system request to the operating system kernel. The file system consults internal data structures to ascertain if it has the requested file in the buffer cache. If the block is not present then the file system requests the device driver to fetch the file. The device driver stub then communicates this request to the user-state server which executes the consistency control and data access algorithms.

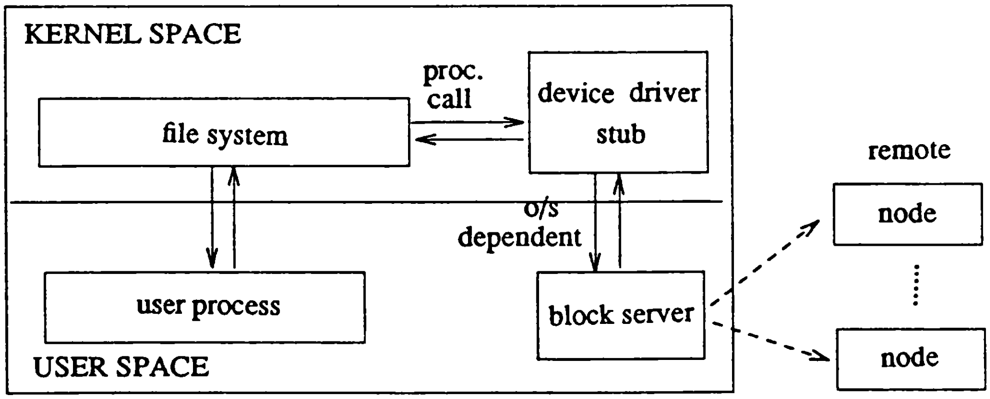


Figure 1.2(b). *Building replication as a block-structured device*

1.5 Types of Failure and Recovery

In this section a discussion on typical failures that can occur in a general distributed environment is followed by the failure specifications and the assumptions on which our failure model is based. The section concludes with a discussion of some methods used for recovering data after the failures.

A failure in the system can be defined as an event at which the system does not perform according to its specifications. We divide distributed system failures into four categories:

1) Node Crashes

2) Bridge Failures

3) Communication Link Failures

4) Byzantine Failures

The first three of these can be thought of as failures causing the network to become partitioned. When the network is partitioned the system is divided into two or more disjoint sets within which communication is possible. There is no communication between any two of these sets in the sense that all messages between them will be lost. A failure is detected when a node fails to receive a response to its message after a certain duration of time. A fault can only be suspected; the absence of a reply might be merely an indication that the recipient is slow to respond but we do not consider timing anomalies. Since communication link failures occur very rarely in today's network technology, they are not discussed further, although their properties could be simulated by a highly interconnected network of unreliable bridges. Arbitrary partitioning of the system caused by bridge failures is important for replication and is studied at length in Chapter Four and Chapter Five.

The last category consists of software failures that cause the system to operate incorrectly in the absence of hardware failures by exhibiting so called Byzantine faults

[48]. We assume that the software is correct so we need not consider this sort of failure further.

As far as communication is concerned node crashes can be viewed as a special case of partitioning: all incoming and outgoing messages are lost. From the point of view of the integrity of the files we must make the following assumptions:

1) Machines are fail-stop. That is, at any moment, each machine is either up or down.

2) Local hardware failures are detectable. It is assumed that a device controller satisfies this assumption.

3) Absence of storage media failures, faults that cause crashes of a file server are classified into two groups: server failures and disk controller failures. Since the replicated file system creates copies of the file on distinct processing nodes rather than by local disk replication, this requirement is satisfied.

Although the effects of a crash cannot be completely hidden, they can be limited to a single well-defined event. The details and requirements of low-level protocols to achieve this aim is described by Schlichting [49] and by Bernstein *et al* [50].

There are many techniques used to restore data in a system to a usable state when the system recovers from a failure. In order to cope with failures, additional components or algorithms must be added to the system. These components ensure that incorrect behavior cannot occur as a result of node crashes. Replication is only one of the methods. Its advantages over the other techniques is that it can be used to improve the availability of data as well. This mechanism and the problems associated with its management forms the bulk of the thesis. Some other recovery techniques are summarized below. Some of these techniques increase the availability partially but none of them improves the availability of the up-to-date copy transparently. The techniques explained below are discussed by Verhofstad [51] and Kohler [27] in detail. Some variations of them are also proposed by Lindsay [52] and Bhargava [53].

Incremental Dumping: Copying of updated files onto archival storage after a job has finished or at regular intervals. This creates checkpoints to updated files. Backup files can be restored after a crash. Powell *et al* [54] have described a redundant system that puts very little additional load on the process being backed up. In their system all messages sent on the network are recorded by a special "recorder" process. From time to time each process checkpoints itself onto a remote disk (Figure 1.3).

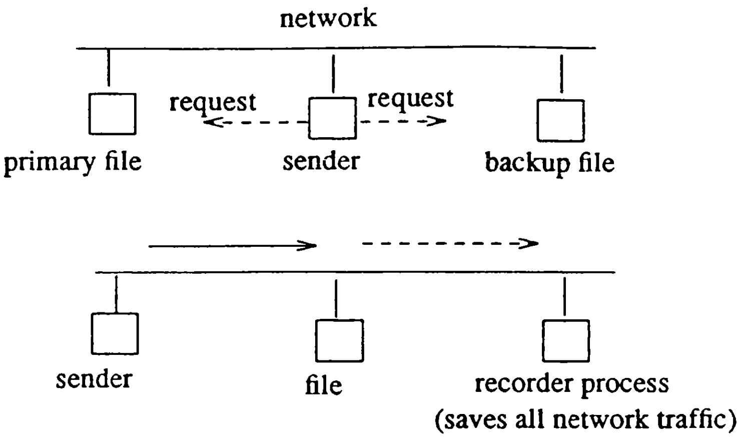


Figure 1.3. *Backup/recorder process during incremental dumping*

Differential Files: A file can consist of two parts; the main file which is unchanged, and the differential file which records all the alterations requested for the main file. The main files are regularly merged with the differential files. Records in the differential files can be stored with the process identifier, a time stamp and other identification information to aid recovery.

Backup/Current Version: The files containing the present values of existing files are the current versions. Files containing previous values are backups. Backups can be used to restore files to previous values.

Careful Replacement: The principle of this method is to avoid updating any part of the object in place. Altered parts are put in a copy of the original; the original is deleted only after the alteration is complete and has been certified.

Table 1.2 lists the methods dealing with recovery and availability.

	Recovery	*Availability*
Incremental Dumping	*X*	-
Differential Files	*X*	-
Backup Version	*X*	*X*
Careful Replacement	*X*	-
Replication	*X*	*X*

Table 1.2 — *Techniques used for recovery and/or availability*

1.6 Measures of File Accessibility

When files are not replicated, they obviously become unavailable during the crash and recovery of the node holding the file. No updates may be made and the data is simply not available for either read or write until the node recovers. No special operations are required upon recovery to be sure of consistency of the copy as there is only one copy. If the file is required while the node is down, it can manually be reloaded into another operating node from a back-up resource. Then, one must make sure that no inconsistencies exist after the crashed node is returned to service. Manual loading sometimes may be useful but it is not transparent.

The higher availability requirements of some applications in distributed systems have increased the interest in keeping copies of the same information at different nodes of the network. Replication of data allows information to be located close to its point of use, either by statically locating copies in high use areas or by dynamically creating temporary copies as dictated by demand. Replication of data increases availability by allowing many nodes to service requests for the same information in parallel and by masking partial system failures. For example, in a system where the independent availability of a node is 0.833 (this corresponds to a failure to repair ratio, ρ=0.2, nodes that are repaired five times faster than they fail), it is possible to increase the overall availability of data to 0.98 with only two replicas (the availability is analyzed in Chapter Four). Maintaining copies can be costly but the reliability of the system is the benefit. Because of its high cost and complexity the reliability offered by replication is only used in certain applications where the cost is justifiable. The rest of this section concerns the measures of file accessibility and methods to compare them.

In a replicated file system design it is essential to know the effectiveness and tradeoffs of different options in improving performance and dependability. Since the main goal of replication is to increase the accessibility of data by tolerating system failures and making the file more available than a single copy, the simplest measure of accessibility is *availability*. In fact, it is possible to distinguish the factors effecting availability into two: environmental effects such as failure frequency, network topology etc. and the limitations of the replication method used to manage the file copies. In order to simplify the availability analysis and ease the comparison between methods, topological factors have usually been disregarded and partitioning has been ignored, although it is a common problem. The following sub-sections explain different forms of availability; steady-state and continuous availability. Continuous availability will be referred as *reliability* throughout.

1.6.1 Availability

The success of a file operation on a replicated file depends on a number of individual nodes being operational at the time of the request. If a sufficient number of nodes is not available which is required for a consistent read or write then the data is not available. *Availability* is a probabilistic measure calculated in terms of the probability of required number of independent components being up at the time of the request. There are two possible availability measures in general:

1) *Instantaneous availability* is a function of time and defined as the probability that the system is performing properly at a given time t. This is equal to the reliability for non-repairable systems i.e. once the system has failed they cannot be repaired and put into function. This is not true in replication systems since the independent nodes are assumed to recover after a repair period [55].

2) *Steady-state availability* is the availability when the system is in steady state. This is the equilibrium state when time goes to infinity. Steady-state availability can be defined as the probability that the file will be accessible at any random point of time. Since it is assumed that the replication system is repairable i.e. failed nodes are always recovered after a certain period of time, only the steady-state availability of files is considered in the thesis. The following is the definition of the steady-state availability of a replicated file.

Definition 1.2. The *availability* $P(A(n,m))$ of a replicated file with n replicas in a system of m processing nodes where $(m-n)$ nodes do not contain a replica of the file is defined as the probability that the system will operate correctly at any given point of time as time goes to infinity given that initially m nodes were operating correctly.

Availability has two facets according to the type of the access: availability for a read access (read-availability) and availability for an update (write-availability). It is a feature of the consistency scheme to determine whether these availabilities are equal or one trades-off the other.

Availability behavior of a consistency scheme can be modeled analytically. This analytical model is an abstraction of the various assumptions about the systems' behavior as a function of the failure/repair probabilities of individual nodes. Under the assumption of exponential failure/repair rates, it is possible to derive a Markov model for the cases where the number of possible states that the system can be in is within reason. Unfortunately, reality tends to deviate from exponential models because exponential repair rate is not realistic for computer systems [56].

If the systems are too complicated to analyze with Markov processes, k-out-of-n reliability theory [57] can be used. The disadvantage of this method is that it makes too many simplifying assumptions about the failure model. In the work presented here, both analytic methods have been used in combination to support each other where applicable. Simulation is used to reach a solution and verify the results of analytic models.

1.6.2 Reliability

Reliability can formally be defined as the conditional probability at a given confidence level that the file system will perform its intended function (read/write access) properly without failure and satisfy the specified requirements of continuous availability during a given time interval $\{0, t\}$. In other words, reliability is the continuous availability of a file over a given period of time.

Definition 1.3. The *reliability* R(n,m,t) of a file with n copies in a system of m processing nodes — including the nodes holding a copy, is defined as the probability that the system will operate correctly over a time interval of duration t given that initially m nodes were operating correctly at time t=0.

Availability has received much more attention, because its analysis is more tractable than that of the reliability [58]. In fact, there are some applications in which the reliability of a system is a more important measure of its performance than its availability. These applications include process control, data gathering, and tasks requiring interaction with real-time processes, where the data will be lost when it is not available. The computer systems used for stock trading are an example of this situation. If these machines were to fail, the resulting chaos would halt trading.

Reliability analysis through analytic models is too complicated. It is possible to derive closed-form solutions for differential-difference equations if the number of possible states is small and the system does not partition. Analytic models become too complicated to solve in the analysis of network partitions. In the reliability analysis a Monte-Carlo simulation is done and the results are validated by an analytic model for a simple partitioning case.

1.7 Summary

After formulating the problem of replication management when the number of replicas are bound to be very small, mainly two (at most three), this introductory chapter outlines the replicated file system model on top of which the consistency control schemes are going to be built in the following chapters.

The most important points made in this chapter are as follows:

i. Storage cost is a very important issue to be considered in order that replication can become common place in general purpose computing environments.

ii. Going from single to two copies has much higher advantages than going from two to three, four copies.

iii. Concurrency and consistency are two separate concerns; the consistency problem is an inevitable consequence of replication whereas concurrency problem can occur in any concurrent or pseudo-concurrent environment.

iv. Two common principles are used in various algorithms for managing the consistent view of replicated files: *read any/write all* and *read some/write some*. Among the algorithms, correctness trades off performance.

v. All consistency schemes in the literature up to June 1989, become inefficient when the number of copies is small (especially two) either because of administrative complexity and requirements from the hardware or providing a desired level of availability especially with small number of copies.

vi. There are two major characteristics of an effective replication control algorithm. Correctness: it should work correctly during network partitions as well as node crashes *and* accessibility: the probability of file being available at any given moment (availability) or a given period of time (reliability) must meet the requirements.

Chapter Two

Consistency Control Schemes

This chapter examines the requirements (assumptions made for its operability and correctness) and the behavior of consistency control schemes.

The algorithms are studied in two groups: Voting algorithms [5,6,59,60] and Available Copies algorithms [4,9]. Voting algorithms use a quorum-consensus approach whereas the Available Copies algorithms provide high availability as a modification of two older methods: Unanimous Agreement [61] and Single-Primary Update strategy [62,61].

Many variations, especially of voting algorithms [63,56] have been proposed recently. Dynamic voting techniques [64,65,66,67] are excluded from this discussion because these algorithms work only if the node failures are distinguishable from network failures and this requirement is at odds with the failure model of the outlined distributed environment. First, the main algorithms and their variations are analyzed using the measures of effectiveness discussed in Section 1.4. Secondly, the advantages of the *regeneration* technique and how it can be applied to a consistency scheme with an additional operation cost is explained.

2.1 Unanimous Agreement Update

This approach requires that all copies should be identical before and after each operation. In other words, it uses *read any/write all* principle. Updates are propagated to all replicas immediately. Since all physical copies of a logical file are kept in the same state, a single copy image of the data is achieved. As the algorithm assumes that every node in the system has a replica of the data, all read requests can be performed locally. This assumption reduces the traffic on the communication network for read requests.

Unanimous agreement enhances read availability, but as the number of replicas are increased, the file will be less available for updates. A replicated file with any number of copies will provide lesser availability for update than a single copy file. Additionally, the system is required to support control message traffic in order to send the update to all replicas and confirm or cancel it, based on whether or not unanimous agreement was obtained. Although the idea is simple, its implementation requires a two-phase commit protocol for confirmation as it cannot afford an inconsistency among the copies.

This approach does not tolerate node crashes for updates. As it is not realistic to consider a failure-free distributed system, it offers very low reliability compared to all the other approaches. Its advantages are high read availability and consistency even if the network is partitioned by preventing updates in any partition that does not have access to all replicas. If a replicated file has a very high ratio of read requests to update requests, unanimous agreement (Figure 2.1) might be cost-effective for small degree file replication.

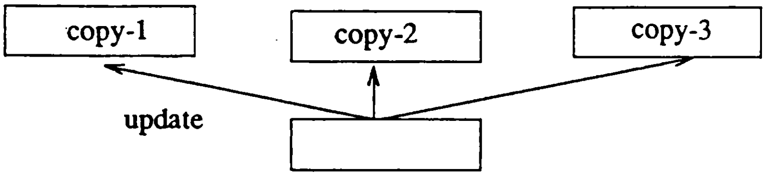


Figure 2.1. *A unanimous update performed on 3 distinct copies*

2.2 Single-Primary Update

This algorithm designates one replica as *primary* and all the others as *secondaries*. Update requests are sent to the primary replica which serves to serialize updates and thereby preserve data consistency. The primary acquires a lock, performs the update, broadcasts the change to all the secondaries and releases the lock. There are three different schemes for this broadcast:

1. Update request is sent to the secondaries immediately,
2. Updates are packaged and sent at the end of the transaction,
3. Updates are broadcast only at specific intervals — once an hour, overnight, etc.

In all the primary-secondary schemes, the delay caused by update propagation from primary to secondary can increase the response time to a local read issued after an update to the data, if the update is at a remote primary, and the read is at the local replica.

This scheme does not tolerate the failure of the primary copy but it maintains consistency in the face of network partitions. In the case of a partition failure, only the partition containing the primary copy can access the data. Updates are forwarded to secondaries at recovery to regain consistency. The availability of the data is simply the probability that the primary is up and communicating. Therefore it provides the same write availability as single copy. Replication enchances only read availability. As it is simple and practicable, it has been used in many designs [68, 69].

2.3 Moving-Primary Update

This algorithm is proposed by Alsberg [62] and it is an extension to the single-primary strategy. The principle is that an update can be made to the primary copy or any secondary copy. The initiator is not aware of which node is functioning as the primary for any particular update.

If the receiving node is the primary one, it performs the update and then sends a cooperation request to one of the secondaries informing it of the update. The secondary performs the update, acknowledges to the primary and also the local node before passing the request on to the next secondary (Figure 2.2). Once the primary has received the acknowledgement from the secondary, it is certain that two-host resiliency has been achieved. The update is lost only if both primary and the cooperating secondary fail.

If the node receiving the request is a secondary, it forwards the request to the primary and algorithm proceeds as above.

If the primary fails, the secondaries will discover it when they forward their next request. Then, they elect a new primary among themselves. In a two-host resilient scheme, all $n-1$ secondaries, where n is the number of copies, must participate in this election. One way of electing a new primary is to assign numbers to nodes and to choose the secondary with the highest number in the participating set as the next primary. In the second step, all other secondaries are informed of the primary change. When the old primary recovers and attempts to ask cooperation for an update, it is informed by the secondary of the change and the request is forwarded to the new primary. The old primary then becomes a secondary.

In general, in an m-host resilient scheme at least $n-m+1$ secondaries must participate in the election of the primary. The rest of the algorithm is the same as the two-host resilient scheme.

This approach works well if node failures are distinguishable from network failures. If this is the case and primary fails, a new primary can be elected (for a discussion of election protocols see Garcia-Molina [70]. However, if it is uncertain whether the primary failed or the network failed, the assumption must be that the network failed and no new primary can be elected. Moving-primary variation enhances the write availability if there are more than two copies. If there are two copies, both copies are required for updates; one as primary the other as cooperating secondary. So, in two-copy case, all synchronization-site approaches behave in a similar way as unanimous agreement.

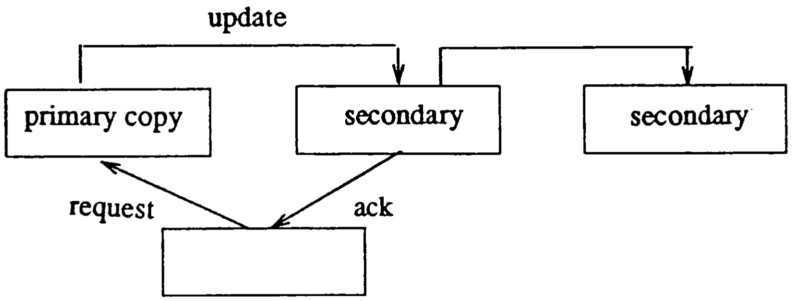


Figure 2.2 *An update operation maintained by Moving-Primary approach*

2.4 Voting Algorithms

Voting algorithms use a quorum-consensus approach. Every node maintains a number of votes for read and writes. Each request must gather a quorum of votes before being accepted. All voting algorithms are robust as a side affect of normal operation. They remain consistent in the case of communication failures which can cause partitioning in the system as well as in the face of individual node crashes.

In general, different quorums for read and write operations can be defined and different weights, including zero, can be allocated to every copy. This form is called as *weighted voting* [6]. Read transactions must collect a read quorum of r votes to read a file, and a write quorum of w votes to write a file. The values r and w must be chosen such that $r+w$ is greater than the total number of votes assigned to the file. There is then always an intersection between the set of servers participating in read and write transactions, so every read quorum is guaranteed to include an up to date copy. Weighted voting introduces version numbers as an alternative to timestamps in order to unify the updates. Each time an update is performed, the version number of every copy in the participating set is increased by one. The highest version number in the read quorum is the version number indicating which copies hold the current state.

In *majority voting*, which is the earliest and simplest form of voting algorithms, every copy has one read and one write vote. For a request to be accepted a majority of the copies need to approve it. The algorithm in its original form employs timestamps both in the voting procedure and in the application of updates. The file is available to update requests so long as there is a majority of nodes in communication. Since only one majority can be formed at a time, the file remains consistent even if the network is partitioned.

In all previous algorithms, read requests were always local. In voting algorithms, if w is less than the number of copies then a read quorum is required to obtain the current version number. If majority voting is applied, then all write quorums are preceded by a read quorum which is the majority in either case. If w is equal to the number of copies then voting degenerates into unanimous agreement allowing any one copy to be read.

Voting algorithms provide serial consistency which means that it appears as if each transaction is running alone. However, they require a minimum of three copies to be of any practical use.[2] Having three copies of the file, in order to increase the availability and the reliability, the best solution is to assign equal votes to all copies. The file will then be accessible as long as two out of three copies are available. The increased level of availability and reliability incurs a storage cost. If the size of the file is very large, then the storage cost of an extra copy may not justify the increase in availability.

2.5 Efficient Variations of Voting

In their original forms, consistency schemes relying on voting become more effective in providing availability as the number of copies are increased (five or more). In these algorithms read availability trades off write availability. In the following sections

[2] Consider a replicated file having two physical copies: if equal weights are assigned to each copy both copies must be available, to acquire a majority in order to update the file. As a result, the availability of the file for either read or write is less than that of a single copy. Should a higher vote be assigned to one of the copies, this copy is the only one required to be available in order to access the file. The second copy has then absolutely no effect on the availability or reliability of the file.

two recent extensions to weighted voting are discussed. The first tends to reduce the storage cost and the second increases write availability.

2.5.1 Reducing Storage Cost with Witnesses

Paris proposed to replace some of the copies with small records that keep only the status of the file but not the data. These records are called *witnesses* [63]. The witnesses have weights just like the normal copies and can participate in a quorum. Read and write quorums are collected as if the witnesses were conventional copies. The only restriction is that every quorum must include at least one current copy. Two copies and one witness provide similar availability to that of a file having three full copies. But still, voting algorithms with three copies provide lesser availability than that of Available Copies method with only two copies. The Available Copies method is discussed in Section 2.6. The availability analysis done with k-out-of-n reliability theory (Section 4.1) has shown that availability with voting becomes reasonably comparable with available copies algorithms when five or more copies are used. For smaller number of copies, available copies algorithms provide higher availability than any variations of voting. The result of the analysis is discussed in Chapter Four.

2.5.2 Enhancing Availability with Ghosts

Voting with Ghosts is proposed by Van Renesse [56]. This algorithm increases write availability in the cases where one or more node crashes mean that a write quorum can no longer be acquired so the data will not be available for writing. It replaces crashed nodes with processes called *ghosts*. Ghosts have the same number of votes as the crashed nodes but do not have the physical copy. Ghosts can be thought of as dynamically created witnesses. The algorithm assumes that the network can only be partitioned at gateways or bridges connecting so called segments. These segments cannot be partitioned. If a segment is down i.e. the communication link has failed, the nodes within that segment cannot communicate with each other or with nodes on other segments. Crashes are detected on each segment with a *boot service* which keeps the status of each node by polling them in regular intervals. This service is replicated as well. Van Renesse argues that, since the segments cannot be partitioned, the boot service can be controlled by either Weighted Voting or Available Copies algorithm.

Voting With Ghosts enhances write availability compared to Voting With Witnesses, but it has strong administrative requirements such as; a separate replicated boot service for every segment and a recovery process to restore the recovered nodes. If the boot service becomes unavailable the algorithm degrades to Weighted Voting. Besides, since ghosts do not have storage, they cannot participate in a read quorum, so read availability remains the same. If the file has only two copies, ghosts have no use in the case of partitioning. Therefore, it has restrictions which makes it unsuitable for small degree replication.[3] Additionally, although it may be a minor overhead, having replicated boot services on every segment generates extra network traffic continually during polling, and updating the service when a node is repaired.

2.6 Optimum Vote Assignment or Coteries

One difficulty with voting algorithms is how to assign the votes optimally. If the failure characteristics of the nodes and the network system is varying, then the optimum vote assignments vary also. It has been proved by Garcia-Molina *et al* [71] that there are up to 2^{n^2} different vote assignments, where n is the number of copies. This shows that the choice of assignments are increased rapidly.

There is an alternative to vote assignment. The above mentioned authors introduced the term *coteries* to define the sets that can perform the read/write operations on the file. Coteries are the sets of set of nodes. Empty set is not a member of a coterie and each

[3] Consider there are two copies, this system can have only one ghost and allows access when either copy is available. But, since the ghost is created only when a node is crashed, if both copies are up on different segments, in the case of partitioning file will be unavailable in both partitions. Therefore, in the case of partitioning voting with ghosts will have no use. Although it preserves consistency, availability is highly reduced in partitioned system when the Voting With Ghosts algorithm is used.

pair of members of a coterie have at least one node in common, but none of them is a subset of another. Coteries can formally be defined as follows:

Definition 2.1. Let U be the set of nodes that compose the system. A set of sets of nodes S is a *coterie* under U iff each member of S obey the following three conditions:

Say G, H are subsets of U

i) $G \in S$ implies that $G \neq \emptyset$ and $G \subset U$.

ii) If $G, H \in S$, then G and H must have at least one common node.

iii) There are no $G, H \in S$ such that $G \subset H$.

Each pair of coteries should have a node in common to guarantee serializability. Up to five nodes, coteries and vote assignments are equivalent. It is easier to think of in terms of coteries but, votes are more efficient in implementation. Garcia-Molina argue that votes take less space to represent and are easier to implement. Adding votes and checking for a majority is also faster than checking if a group of nodes is in a coterie.

Also, with five or fewer nodes, the number of choices for vote assignment or coteries are small enough for designers to inspect all choices and select the one that yields the best reliability for the given hardware.

They prove that for systems with more than five nodes, coteries are more powerful. There are coteries that cannot be represented by votes, not vice versa. However, in this case the number of coteries is huge. Therefore, some heuristics are needed to trim down the number of choices.

As stated above, for the systems with more than five copies, either assigning optimum votes or choosing optimum coterie is a tautology and difficult task for system designers.

2.7 Available Copies

In this algorithm failed nodes are automatically detected and configured out from the system. Recovered nodes bring themselves up-to-date by copying from other available nodes before accepting any user transactions. Reads are initiated to any available copy but writes must be done to all available copies. This form of unanimous agreement provides better availability than all other methods, but the file's consistency cannot be maintained in the presence of partitions. Each copy maintains a directory list of available copies for use. The algorithm runs status transactions to keep these lists up to date as nodes fail and recover. Since the algorithm can detect only node crashes, if the network is partitioned, different partitions can update different copies and leave the system in an inconsistent state. The original algorithm requires a method to handle the total failure situation specifically. This situation occurs when all the copies are failed. In this case, the last failed node is determined and updates are not accepted until this node is recovered. In order to use this algorithm, a transport protocol must provide reliable, error-free communication between nodes.

This algorithm can only perform well in a partition-free network if nodes fail infrequently. When a node fails, the algorithm updates the directory information on all the other copies of all data items stored in that node. When a node recovers, it informs the directories again. Maintaining these status lists is a costly work.

2.7.1 Handling Partitions

El-Abbadi extended the original Available Copies scheme to *Accessible Copies* scheme in order to handle partitions [9]. The extended scheme which is critically surveyed by Davidson *et al* [72] is based on the following intuitive *read one/write all* protocol:

(1) A data item can be read and written within a partition only if a majority of its copies reside on member nodes of the partition. In this case, the item is said to be *accessible.*

(2) A read operation on an accessible data item is implemented by reading the nearest copy of the item residing on a member of the partition.

(3) A write operation on an accessible data item is implemented by writing all copies residing on members of the partition.

The first rule ensures that only one partition can access the file. The second and third rules guarantee that the file remains consistent within a partition.

This protocol ensures one-copy serializability in an *ideal* network, where partition failures are clean and nodes can detect partition failures almost instantaneously. If either property of the ideal network is violated, incorrect execution can occur. The principal idea in the Accessible Copies algorithm is the implementation of an abstract communication layer on top of the real communication network, where the behavior of the new layer approximates that of the *ideal* network. The consistency scheme is implemented on top of the abstract layer. The abstract layer creates and manipulates *virtual partitions*, which are rough analogs of the actual partitions that occur in the real network.

This variation offers similar availability to voting for updates. Virtual partitions require status transactions and directory lists at each node in order to keep track of the failures and recoveries. Although the principle is simple its requirements are costly in implementation.

2.8 Regeneration

The reliability of a replicated file depends on maintaining the set of up-to-date replicas. Usually space limitations make it impossible to have enough copies to guarantee the level of fault tolerance required. If new replicas of a file can be created faster than a failure can be repaired, then better reliability can be obtained by creating new replicas on available nodes in response to node failures. This technique is known as *regeneration.* The idea of regenerating replicas to replace failed nodes was first proposed by Pu *et al* [73]. This algorithm is an extension to the Available Copies method and carries the same weaknesses. It allows reads to continue as long as one up-to-date copy is available. If fewer than the initial copies are accessible during a write operation, then new copies are regenerated on other available nodes. If there is no spare node, then the write fails. Regenerating to the maximum number of copies whenever an update occurs (as long as adequate number of nodes exist) increases the availability of an up-to-date copy for further writes.

Recently, it has been shown that regeneration is a generally applicable technique that can be combined with many replica control protocols in order to increase reliability [58].

However, regeneration requires some work to be done when a node recovers and joins the net. For all the replicated data in the recovered node, the replication system must check whether the maximum number of copies is already available or not. If so, the recovered copy is deleted. If not, the system must check whether there has been an update or not during the failure of this node. If there has been no update during the failure then the copy is used; otherwise it should be deleted because its replacement exists but vanished temporarily due to another node crash. Figure 2.3(a) illustrates a situation when the copy on node 3 is recovered, it can be used as there has not been an update during its failure. Figure 2.3(b) shows a situation where there has been an update during the failure and as a replacement copy exists on node 4, the recovered copy should be deleted.

Additionally, there is a communication cost associated with regeneration. When a node fails or recovers, network message traffic is increased.

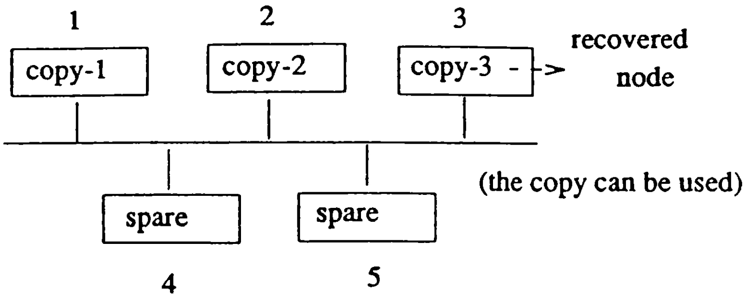


Figure 2.3(a). *Regeneration: recovery of a repaired node (case-1)*

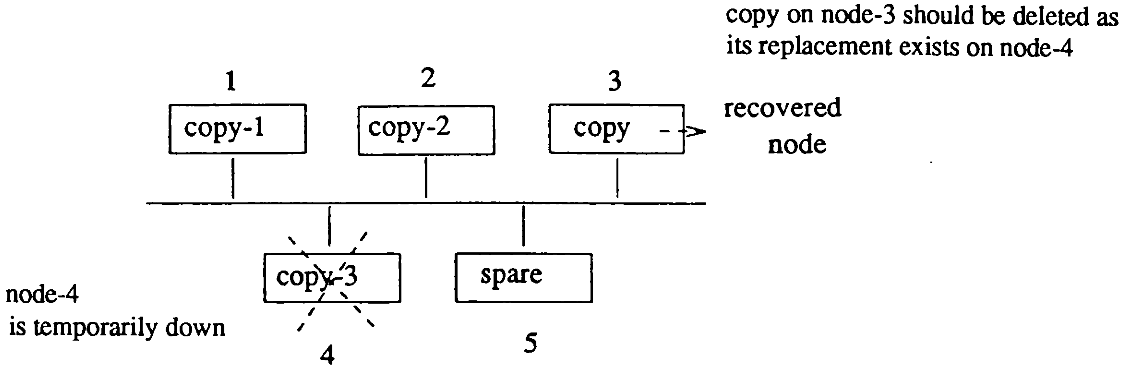

copy on node-3 should be deleted as its replacement exists on node-4
1
2
3
copy-1
copy-2
copy
recovered node
copy-3
spare
node-4 is temporarily down
4
5

Figure 2.3(b). *Regeneration: recovery of a repaired node (case-2)*

If the size of the replicated file is very large, data transmission during each regeneration might be very costly if nodes are failing frequently. For this reason, regeneration may be best suited for small size data rather than large files. There are some other ways of reducing cost. One way is to regenerate when the number of replicas falls below a certain threshold. Another way is to delay regeneration for a certain time period following the failure. This may prevent wasting resources by reacting to transient failures.

Regeneration-based replica protocols offer higher reliability but the analysis is not tractable. It is only possible to derive closed-form solutions for the reliability of the protocols for small number of nodes. Long [58] obtained the solutions for regenerative algorithms. These solutions which can be applied to different algorithms employing regeneration are discussed in Chapter Five.

2.9 Discussion

All consistency schemes discussed above become less efficient in a distributed system where nodes fail and recover frequently, and are less effective when the degree of replication is small; especially two. As the effectiveness and dynamicity of the algorithms are increased to provide better availability, they become less practicable. Although the actual performances of the algorithms are implementation specific and depend on the underlying hardware, there are some abstract performance measures (such as number of control messages required etc.) that can be used to estimate the likely communication overhead associated with the algorithms. Still, with efficient implementations, communication cost can be kept within acceptable bounds even for the algorithms which require interactions between large number of nodes. Implementation aspects related to communication cost will be discussed in Chapter Six. However, the requirements of a consistency scheme which are listed below must be met within the bounds of required degree of availability and reliability. These requirements would allow replication to be used in general purpose computing applications.

a) *Fault tolerant*: as far as communication is concerned, the scheme should tolerate all forms of partitioning in the system including node crashes and bridge failures.

b) *Low storage cost*: it should improve the availability and reliability even with two copies.

c) *Flexible*: it should provide a dynamic reconfiguration facility in order to reach the desired level of accessibility with an easy administration.

d) *Practicable*: it should be implementable as an extension to the existing file system when the underlying communication medium satisfies the requirements.

e) *Simple*: It should have a simple failure model. For example: it should not require all nodes in the system have the same understanding of which nodes are available and which are not. It should not expect the communication system to detect partitioning failures instantaneously and distinguish network failures from node failures.

All the algorithms discussed in this chapter fail to achieve one or more of the requirements.

2.10 Summary

The consistency algorithms are discussed for their effectiveness in a distributed environment where partition failures can occur and the degree of replication is small. The family of available copies algorithms allow access to the file as long as one up-to-date copy is available whereas voting algorithms require a number of copies of its quorum size. Although some techniques are developed to improve the effectiveness of voting strategies (ghost processes, witness copies), these algorithms become effective when the number of votes is five or more and require at least three copies for any practical use. Besides, optimum vote assignment is a complicated problem in practice for the systems with undeterministic failure characteristic. Dynamic strategies are proposed to improve the availability by changing the votes assigned to the copies dynamically to reach a quorum but these algorithms have stringent requirements from the underlying communication system such as recognition of failures instantaneously and by their types (node crash or network partition etc.).

Chapter Three

A Hybrid Replication Algorithm

In Chapter Two, various consistency schemes are examined for a low degree of replication in a large-scale distributed system. The critical considerations about the communication system were:

a) Whether the algorithms work if all nodes do not have the same understanding of which nodes are up and which are down

b) Whether the processing nodes are required to distinguish the failures and recognize them instantaneously.

Under the above requirements, dynamic vote assignment strategies [64, 67] have no use as they require partitioning failures to be distinguishable from node crashes and the failures to be detected instantaneously.

As far as effectiveness is concerned, Available Copies (AC) schemes provide the optimal availability for two copies but cannot tolerate partitions unless the nodes have the same view about the state of the network and the same failure detection requirements as dynamic strategies are required. On the other hand, static voting algorithms do not have the failure recognition requirements but have the disadvantage of needing at least three copies to give higher availability than a single copy. The availability with voting can be improved with the use of witnesses; records of the current state of the file that replace the full copies. Voting With Witnesses (VWW) do yield almost as much availability as the majority voting (MV) with three copies, but still voting algorithms cannot reach the optimal availability provided by AC with only two copies.

One important property which is not investigated as a part of the consistency scheme in the literature is ease of reconfiguration; the file might require different degrees of availability for different periods of times during its lifetime. This availability can be achieved either changing the locations of the copies — moving the copies to the nodes that are going to be up for the required amount of time — or creating extra (temporary) copies to reach the desired level. An effective file replication scheme should provide a flexible and easy reconfiguration facility to the user as a property of the consistency algorithm without any additional system administration requirements. The degree of dynamicity in reconfiguration is also important.

The algorithm proposed here overcomes the basic disadvantages of static voting algorithms (minimum three copies requirement) and available copies algorithms (fail under partition) in a distributed system where each node has its own view of the state of the network (this state indicates which nodes are up and which nodes are down). It is an alternative control scheme for low degree replication, especially two copies, and has a dynamic reconfiguration facility. It provides high availability allowing access to the replicated file as long as one up-to-date copy is available and keeps the consistency of the data in the face of node crashes and network failures that can cause partitioning in the network.

The algorithm replicates a small amount of data concerning the location and status of a file's small number of copies (its *history*) a large number of times using a variation of the voting strategy, and uses this highly reliable vital information to determine the update strategy. The consistency of the file is preserved when the system is partitioned due to bridge failures since the history can be accessed only in one partition in which a majority can be collected. Although independent schemes are used for two entities (files and histories) to achieve consistency, the interaction between these entities provides a consistent view of the file's small number of copies during partitioning failures. This hybrid protocol will be called *reliable histories* (RH) throughout. The interaction between the protocols will be described shortly.

The algorithms for the logical operations are presented here in a pseudo-code based on set notation and predicate calculus mainly because they rely heavily on set manipulation and require very little in the way of conventional control structures. The following sets are used throughout this chapter:

$T = \{true, false\}$

N the natural numbers

M the set of processing nodes

F the set of file identifiers

$V = 2^{M \times N}$ (sets of server and version number pairs)

Σ^* the set of byte strings

The notation 2^S denotes the power set of S, i.e. the set of all subsets of S. The conventional set notation used in the algorithms is given in Appendix A.

3.1 Replication Control Service

A replicated file is defined as a set of file copies, each one implemented on a distinct node in the distributed system. It is the job of the consistency control algorithm to coordinate the accesses and updates to the file copies so that clients of the replicated file system see a consistent view of the file. That is, any client that reads a file after a write operation has succeeded will see the data as it was left by the write operation. It is assumed that processors act synchronously and have access to a local clock. No assumptions are made about the synchronization of these clocks at this point (some assumptions will be made in Chapter Six for performance optimization).

The file system presents applications with the abstraction of a logical file consisting of a sequence of bytes and identified by a unique identifier $f \in F$. It is considered to be a sequence of bytes, any subsequence of which may be read or replaced by any other byte sequence. The file system provides five operations, four of which are the operations on these logical files that are assumed to exist for local ones: *create*, *read*, *write* and *delete*. The *create* operation introduces a new file ID that refers to an empty file. The data referred to is accessed and modified only by read and write operations. Read is used to return an arbitrary, contiguous sub-sequence of the file's bytes, while write is used to replace such a sub-sequence with any other byte sequence. Once the data in a file is no longer needed, the file ID is deleted, so that the file system may reclaim the space occupied by the file. The fifth operation, *configure*, maintains the file's history for a new configuration set. This operation enables the client to change the location of the copies or the degree of replication during the life time of the file. The *configure* operation will be described separately in Section 3.4.

Logical files are implemented by a (possibly empty) set of physical files each holding a complete copy of the data in the logical file and each residing at a single, distinct processing node. Two protocol layers within the Replication Control Service coordinate access to the physical copies so as to ensure that read requests return the most up-to-date version of the file. The *availability control* protocol determines the appropriate update strategy for a file operation based on the file's history table. The history table records the location and version number of each copy along with a boolean flag that marks the file as having been deleted. The Replication Control Service deletes the file ID so that the storage can be occupied by the file system. There is no assumption about how and when the storage is reallocated. This is left to the file system. Marking the deleted files instead of returning them to the storage pool immediately allows easy recovery of the current version if the file is wanted to be recreated shortly after the deletion. The history is maintained by the *history table control* protocol.

The number and location of the copies of each file are controllable by its owner, and both may change during the file's lifetime. This is implemented as an interface into the lower control layers of the replication control system.

Any client wishing to read a file consults the history information to determine which file copies are up to date. Write operations perform update on all the up-to-date copies and copy the current version to obsolete copies. Clients multicast a write request, including the highest version number in the file's history, to all servers that hold a copy of the file. This write request is not applied immediately to the file but is held pending until the next read or update request. Subsequent requests to read or update the new version cause the pending update to be applied (if there is one). If a request is made for the old version, any pending update is discarded. This mechanism ensures that the file copies are kept in step with the history table, even when a network partition during an update prevents the new history from being saved after the file copies have been updated. Such a failure will be reported to the client, and any subsequent operations will cause the unsuccessful update to be discarded. The algorithms do not configure out the obsolete copies. Obsolete copies are also active entities in the system but do not participate in write operations. The replicated file system can provide an alternative read operation which returns the available most up-to-date copy (not necessarily the current version). The obsolete copies therefore increase the read availability for the files in this category (it may be better to read an old version of a host address table than none at all). Figure 3.1 illustrates the components of the system.

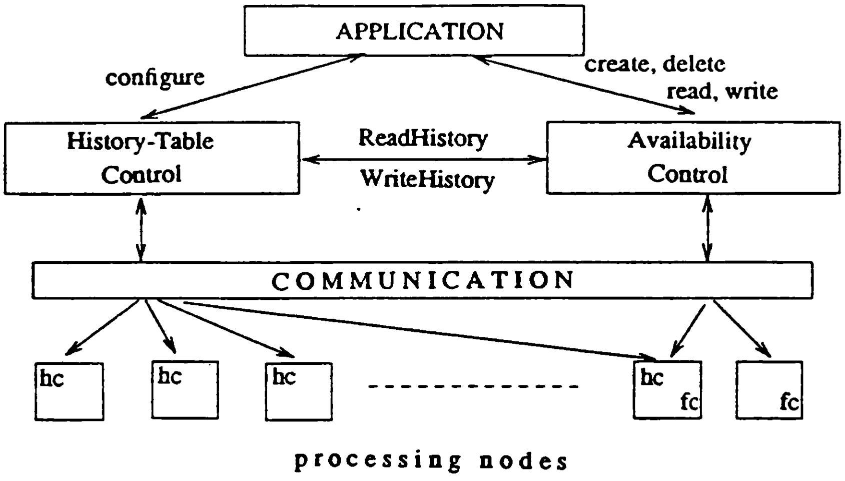


Figure 3.1. *Interacting components of the replication control system*

3.2 Availability Control Protocol

Each logical file $f \in F$ has a history table $h(f) \in V \times T$ that records the version numbers and locations of every physical copy of the file along with a boolean flag used to mark the file as having been deleted. Two operations,

$$ReadHistory\colon\ F \rightarrow V \times T$$
$$WriteHistory\colon\ F \times (V \times T) \rightarrow \{success, error\}$$

are provided to manipulate this table. A read of the table for a particular file returns this flag together with the locations and corresponding version numbers of its physical copies. A write to the table records new machine and version number pairs and can be used to set the delete flag. This history information is itself replicated using a separate algorithm described in the next section.

At file creation time, all copies are assigned version numbers zero:

```
$create(f, S)\colon\ F \times 2^M \rightarrow \{success, error\}$
    let $h \leftarrow \{(m, 0) \mid m \in S\}$
    return $WriteHistory(f, (h, false))$
```

No physical file copies are created until the file is first written to. For example: when the file is logically created on a set of nodes, $S=\{m_1, m_2\}$, the files' history, $h = \{(m_1, 0), (m_2, 0)\}$ will be written to a set of nodes, H. The sets H and S may be disjoint or not. Figure 3.2 illustrates the order of operations during file creation.

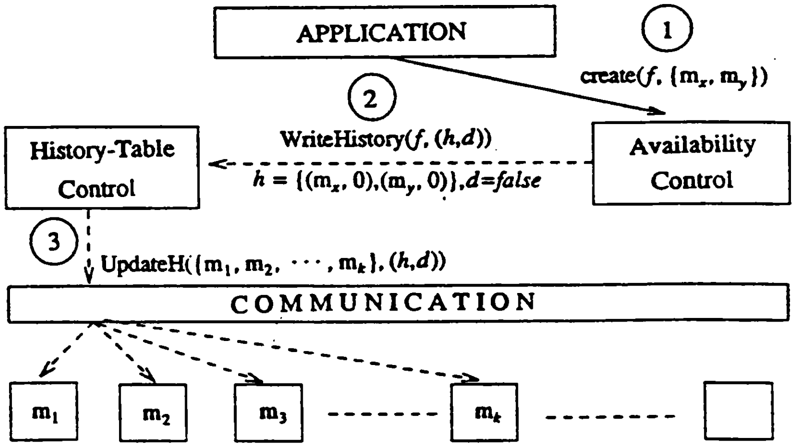


Figure 3.2. *Layer interaction during create operation*

The *delete* simply attempts to write a new history that records the file as having been deleted using the flag already mentioned (Figure 3.3).

```
delete (f): F → {success, error}
    return WriteHistory (f, ({ }, true))
```

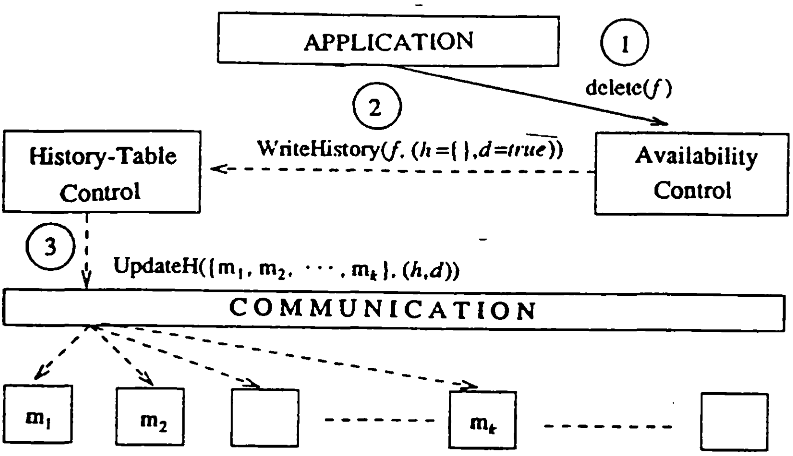


Figure 3.3. *Layer interaction during delete operation*

A file may be in any one of four *availability states*, determined by inspecting its reliable history table.

1) All copies are available and up-to-date.

2) All available copies are up-to-date but some copies are unavailable.

3) Some of the available copies are not up-to-date.

4) No up-to-date copy is available.

The availability control protocol determines the appropriate access and update policy for each state. In order to read a file it must be in state 1, 2 or 3, i.e. at least one available copy must be up-to-date.

If the file has been deleted (d = *true*) or the file's history is not available ($h = \varnothing$) then the operation fails. The up-to-date copies have the maximum version number in the history table. The copies with zero version numbers are the new copies which are created by *configure* operation but not physically written yet. These copies are brought up-to-date by copying from the existent copies when a write operation is performed. The set of servers holding copies with the highest version number is found and a read request is multicast to them. Figure 3.4 illustrates the order of the physical and logical operations for a read request.

```
$read(f, posn, Size)$: $F \times N \times N \rightarrow \Sigma^* \cup \{error\}$        (algorithm-1)
    let $(h, d) \leftarrow ReadHistory(f)$
    if $(d = true \vee h = \varnothing)$
    return $error$[1]
    let $latest \leftarrow \max(\{i \mid \exists (m, i) \in h\})$
    let $U \leftarrow \{m \mid \exists (m, latest) \in h\}$
    return readF$(U, f, posn, Size, latest)$
```

The following multicast operation is invoked on physical copies:

readF$(U, f, posn, Size, v)$: $2^M \times F \times N \times N \times N \rightarrow \Sigma^* \cup \{error\}$

> returns data (specified by size and position) obtained from any server in the set U, or an error indication if no server responds.
>
> Each server compares its copy's version number with v. If they are equal, any pending update is applied to the copy before returning it. If they are not equal then the server discards the pending update, and decrements the copy's version number.[2]

[1] The first condition occurs if the file is deleted in a subsequent operation. It is possible to recreate these files if the space has not been allocated to another process. This requires only setting the delete flag to *false* again in the history. The recreation property can be added to the *configure* operation if it is required. The second condition occurs if the history table is not accessible as a desired number of nodes are not operational in the system.

[2] If v is not one less than the copy's version number or there is no pending update, the system has become seriously inconsistent (the algorithms ensure that this case never occurs).

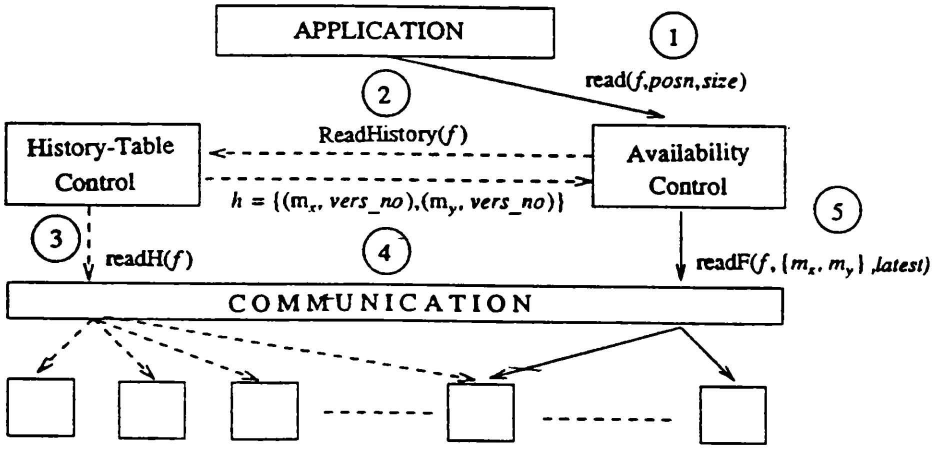


Figure 3.4. *Layer interaction during read operation*

An alternative read operation can be provided for files in availability state 4 that will read the most up-to-date version that is available. This operation can be used when all up-to-date copies have been lost forever (by disc failure, for example) or the file has very weak consistency requirements. For example it may be better to read an old version of a host address table than none at all.

The algorithm is similar to that for the read operation above, except that it iterates if the readF request fails, picking all servers that hold versions one less than *latest* until the lowest (positive) version number has been tried. .

A *write* operation will succeed if at least one file copy can be updated and the file's new history can be recorded. The update is multicast to all up-to-date copies, and servers holding out-of-date versions are asked to copy the new file. The set of servers which accepted either the update, R, or a new copy, C, will hold up-to-date versions and this is recorded in the new history with an incremented version number for these nodes. Untouched servers have their history table entries copied into the new table from the old one. The formal description of the write operation is as follows:

write $(f, data, posn)$: $F \times \Sigma^* \times N \rightarrow \{success, error\}$ (algorithm-2)

```
let $(h, d) \leftarrow ReadHistory(f)$
if $(h = error \vee d = true)$
return $error$ — see footnote 1
let $latest \leftarrow \max(\{i \mid \exists (m, i) \in h\}$
let $U \leftarrow \{m \mid \exists (m, latest) \in h\}$
let $R \leftarrow \mathrm{updateF}(\mathrm{U}, f, data, posn, latest)$
if $(R = \varnothing)$
return $error$
let $S \leftarrow \{m \mid \exists (m, i) \in h\}$
let $C \leftarrow \mathrm{copyF}(\mathrm{R}, f, (S - R), latest)$
let $h' \leftarrow \{(m, latest{+}1) \mid m \in C \cup R\} \cup \{(m, i) \in h \mid m \notin C \cup R\}$
return $WriteHistory(f, (h', false))$
```

The following support operations are invoked on physical file copies:

updateF(U, f, $data$, $posn$, v): $2^M \times F \times \Sigma^* \times N \times N \rightarrow 2^M$

Multicasts a request to write *data* to the file f to all the servers in the set U. It returns the set of servers that accepted the request.

copyF(R, f, X, v): $2^M \times F \times 2^M \times N \rightarrow 2^M$

Copies the file f from any server in the set R to all the servers in the set X. Again, the result is the set of servers that accepted the operation.

Those servers that accept the updateF or copyF operations first decide whether to apply or discard any pending write (if there is one) by employing the same rules as readF. Secondly, they attach a version number one higher than $v(latest)$ to the copy and do not commit the current update until the next request. Therefore up-to-date copies are always between two versions (v and $v{+}1$). The order of operations for a write request is shown in Figure 3.5.

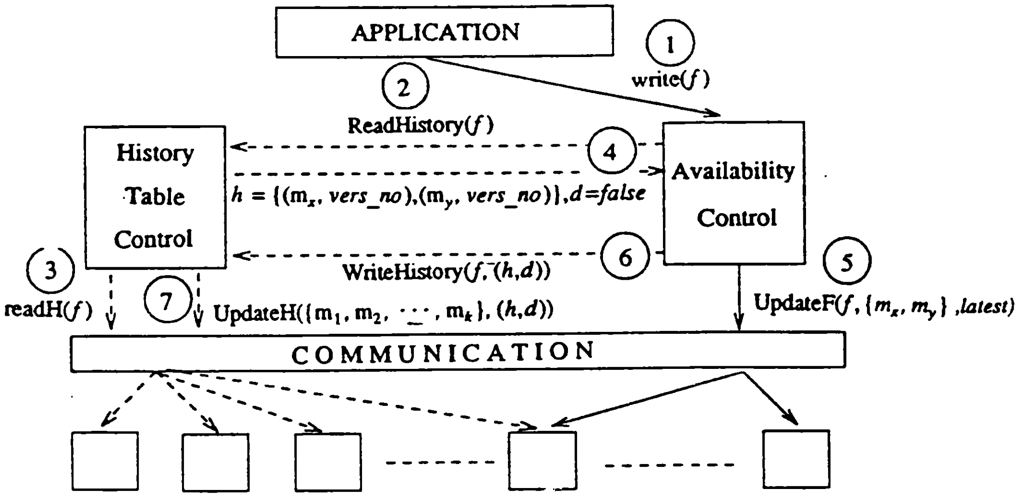


Figure 3.5. *Layer interaction during write operation*

3.3 History-Table Control Protocol

The *reliable histories* algorithm presented above, requires a highly reliable, consistent history table to be maintained for each file. The History Table Control Protocol provides this requirement using the operations:

$$\mathit{ReadHistory}:\ F \longrightarrow V \times T$$
$$\mathit{WriteHistory}:\ F \times (V \times T) \longrightarrow \{\mathit{success}, \mathit{error}\}$$

The history records whether the file has been deleted (the truth value is interpreted as a "deleted" flag) and a set of machine and version number pairs ($V = 2^{M \times N}$).

The history table control layer supports these operations by replicating the table using a variation of the basic majority voting algorithm so that the file histories are consistent in the face of network partitions. The history tables are made highly available by replicating them on k sites, where $k \gg n$, the number of file copies.

In its simplest form, $k = \lfloor m/2 \rfloor + 1$, where m is the total number of processing nodes, each node is assigned one read and one write vote, regardless of whether or not it holds a copy of the file's history. The algorithm will allow a read to succeed even if only one copy of the history table is available, so long as the majority of nodes are up. Writes to the table require a majority of nodes to accept the new table version. In the case of random node crashes the method will offer a high degree of read availability. Random network partitions will reduce availability more seriously but the table will still be consistent. Analysis of partitioning is done in the following chapters.

3.3.1 Communication Layer

Each replica of the history table of $f \in F$ keeps a timestamp ts(h) of the last update on itself such that if ts(h_i) = ts(h_j) then h_i and h_j are the same, if there is a timestamp such that ts(h_i) < ts(h_j) then h_i is an out-of-date table.

The control operations multicast the following requests in order to read/write a replica of the table.

readH(M, f): $2^M \times F \rightarrow 2^{M \times N \times (V \times T)}$

Invokes a lookup and mapping request to all processing nodes and returns the table versions together with the last update time of the returned version and the set of nodes returning that version. If the node is not holding a copy of the table then it returns a negative indicator.

updateH(M, f, h): $2^M \times F \times V \rightarrow 2^M \cup \{success, error\}$

Invokes an update request on table to all nodes and returns the set of nodes which accepted the new history. At least $\lfloor m/2 \rfloor + 1$ acceptances are required for successful completion, where m is the number of processing nodes in the system.

The physical history read operation given above is invoked when the *ReadHistory* function is called. If less than a majority of the nodes reply then the file is unavailable. Otherwise, the table which has the highest update time is returned. The formal description of the algorithms for the table functions are given below.

```
ReadHistory(f): $F \rightarrow V \times T$        (algorithm-3)
    let $w \leftarrow readH(M,f)$
    if $|\ w\ | \leq \lfloor m/2 \rfloor + 1$
    return $error$ — table is not available
    let $S \leftarrow \{s \mid \exists (s,t,(h,d)) \in w\}$
    let $maxtimestamp \leftarrow max(\{t \mid \exists (s,t,(h,d)) \in w\})$
    let $L \leftarrow \{s \mid \exists (s,maxtimestamp,(h,d)) \in w\}$
    return $(\{(h,d) \mid \exists s \in L \wedge (s,t,(h,d)) \in w\})$
```

The corresponding update on the table is initiated when the *WriteHistory* function is called.

```
$WriteHistory(f,(h,d))$: $F \times (V \times T) \rightarrow \{success, error\}$
    return $(updateH(M,f,h))$
```

3.4 System Configuration

The *configure* operation changes the placement of the copies at any time during the file's lifetime by using the history table control protocol. Since the new locations for the copies are added to the history with a zero version number the configure operation must ensure that the new set contains at least one up-to-date copy (highest version number greater than zero).

If the intersection between the old and the new sets contains an up-to-date copy then no copying of the file is required during the configuration. New locations are added to the history table with zero version numbers. Any *write* operation following the *configure* operation will attempt to overwrite the copies with zero version numbers by copying from the updated version.

If the intersection does not contain an up-to-date copy then it is necessary to ensure that the current version is copied to at least one of the nodes in the new configuration. If none of the above conditions is satisfied, the configuration of the file is left as it is. The following algorithm combines the two possible cases explained above in one operation. This operation ensures that file *f* is configured on the servers in the set *P*.

$configure(f,P)$: $F \times 2^M \rightarrow \{success, error\}$ (algorithm-4)

```
    let $(h, d) \leftarrow ReadHistory(f)$
    if $(h = error \vee d = true)$
    return $error$[3]
    let $S \leftarrow \{m \mid \exists (m, i) \in h\}$
    let $latest \leftarrow \max(\{i \mid \exists (m, i) \in h\}$
    let $U \leftarrow \{m \mid \exists (m, latest) \in h\}$
    let $h \leftarrow AddH(h, P-S)$
    if $(U \subseteq (S-P))$[4]
    then let $C \leftarrow$ copyF(U, $f, P$)
        if $(C = empty)$
        return $error$[5]
        let $h \leftarrow DeleteH(h, (S-P))$
        let $h' \leftarrow \{(m, latest) \mid m \in C\} \cup \{(m, i) \in h \mid m \notin C\}$
    else let $h' \leftarrow DeleteH(h, (S-P))$
    return $WriteHistory(f, (h', false))$
```

The following support functions are used. These functions are performed locally and the changes multicast to history copies if the new configuration is succesful.

AddH(h, S): $V \times 2^M \rightarrow V$

Returns the history h, augmented by the history information $(m, 0)$ for each machine m in set S.

DeleteH(h, D): $V \times 2^M \rightarrow V$

Returns the history h, less any pair (m, x) with m in set D.

[3] This condition can be extended to include the recreation of previously deleted copies as discussed in Section 4.1. See also footnote 1.

[4] New configuration includes no up-to-date copies.

[5] This condition occurs when none of the up-to-date copies exist in the new configuration and all up-to-date copies are unavailable therefore cannot be copied into the new configuration. In this case, the file configuration is left as it is.

3.4.1 Example Scenarios for the *Configure* Operation

The following scenarios show the changes in the history of a file during consecutive calls of the *configure* operation.

The following terms are used:

$h_i = \{(loc, version), (loc, version) \ldots\}$:

The resulting history table after the *configure* or *write* operation is completed in scenario *i*. They denote the location and version number of copies in pairs.

$A_i = \{ \cdots \}$:

The set of nodes that are up during scenario *i*.

The set names are the same as are used in the *configure* algorithm. *U*: the set of up-to-date copies, *P*: the new configuration set, *C*: the set of nodes on which the copying of current version has succeeded. Each scenario applies an independent configuration request but takes up the system parameters (e.g. history table) from the previous scenario. There are total 10 nodes in the system (numbered 1..10) three of which keep a copy of the file. Initially, $h_0 = \{(1,1), (2,1), (3,1)\}$.

In order to show the progress of configuration clearly, it is assumed that no update occurs between the first three scenarios, therefore the history is changed only by the *configure* operation in these cases. The last scenario applies a combination of updates and configuration together.

Scenario One:

$A_1 = \{ 1, 5, 6, 7, 8, 9, 10 \}$,

$configure(f, P = \{ 1, 3, 4 \})$

Here, the file is to be configured on $P = \{ 1, 3, 4 \}$. Since the $S = U = \{ 1, 2, 3 \}$ is not a subset of $S - P = \{ 2 \}$, the fact that the nodes 3 and 4 are down does not affect the acceptance of the new configuration.

Copy 4 is added to the history with zero version number indicating an attempt to create it must be made at the next update and the history information for copy 2 is removed. The entries for nodes 1 and 3 are left as they were.

History becomes $h_1 = \{(1,1), (3,1), (4,0)\}$

Scenario Two:

$A_2 = \{ 1, 4, 5, 7, 8, 9, 10 \}$,

$configure(f, P = \{ 4, 5, 6 \})$

In this case, $U = \{ 1, 3 \}$ is equivalent to $S-P = \{ 1, 3 \}$. This means that the configuration, P, does not contain an up-to-date copy. Therefore at least one copy operation from node 1 or 3 to the nodes in the new configuration (4, 5, 6) has to be successful before accepting the configuration.

copyF($U = \{ 1, 3 \}$, $P = \{ 4, 5, 6 \}$) returns $C = \{ 4, 5 \}$, the copying on $\{ 4, 5 \}$ is successful (node 6 is down).

History is changed to: $h_2 = \{ (4,1) , (5,1) , (6,0) \}$. Node 6 will be brought up-to-date during the first successful write after it recovers.

Scenario Three:

$A_3 = \{ 1, 2, 3, 6, 7, 8, 9, 10 \}$,

$configure(f, P = \{ 6, 7, 8 \})$

In this case, $U = \{ 4, 5 \}$ is equivalent to $S-P = \{ 4, 5 \}$. This means that, there is not an up-to-date copy in the resulting set as in the previous scenario. Therefore, at least one copy operation from node 4 or 5 to the nodes in the new configuration (6, 7, 8) has to be successful.

copyF($U = \{ 4, 5 \}$, $P = \{ 6, 7, 8 \}$) returns $C = \varnothing$. Copying is unsuccessful as both nodes 4 and 5 are down.

New configuration is not accepted and the history remains unchanged: $h_3 = \{ (4,1)$, (5,1) , (6,0) }.

Scenario Four:

$A_4 = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \}$,

write (*f*),

configure ($f, P = \{ 6, 7, 8 \}$),

write (*f*)

1) All copies are brought up-to-date by the write operation, so: $h_{4.1} = \{ (4,2) , (5,2) , (6,2) \}$.

2) A request is made to configure the file on $P = \{ 6, 7, 8 \}$. $U = \{ 4, 5, 6 \}$ is not a subset of $S - P = \{ 4, 5 \}$, therefore a copying operation is not required.

   The new configuration is: $h_{4.2} = \{ (6,2) , (7,0) , (8,0) \}$.

3) The *write* after the reconfiguration brings all the copies up-to-date.

   The history becomes: $h_{4.3} = \{ (6,3) , (7,3) , (8,3) \}$.

3.5 User Requirements

The *reliable histories* method is aimed at the applications in general purpose computing environments where storage costs must be kept low, and where the units of data that are to be kept consistent are typically several kilobytes or more, such as, document preparation, program development etc.

The answers given to a questionnaire in our department has shown that, more than 50% of the academic staff and research students are keeping multiple copies of about 15% of their vital files for robustness and increased availability but the files they replicate have variable degrees of availability requirements during different time periods. In a replicated file system, one of the basic requirements of the user is to have control over defining the high availability periods. The desired level then can be reached by moving the copies or increasing the degree of replication. The flexibility of configuration therefore is an important property for reducing the administrative requirements of replicated files. Another advantage of the replication history is that it allows the algorithm to work properly when the replication degree is only one (single copy). The storage cost can be reduced by keeping only one copy for the periods in which the file does have low availability requirements. How to define measurement metrics to reach a certain degree of continuous availability when the periods are defined by the user is a new direction for future work in this area.

3.6 Discussion

History tables provide a mechanism which gives consistent updates in the presence of network partitions. This is not possible with the original available copy approach. Although the history table must be replicated using a voting strategy and requires a high level of replication in order to give the degree of fault tolerance required, it is relatively small compared to the size of the file itself.

In order to reduce the communication cost, the number of history table copies can be reduced by assigning higher weights to some table copies. This reduces the number of responses required to complete an operation. In the following chapters, the availability analysis is based on the assumption that all nodes are assigned one vote for read and one vote for write. This case has shown the worst case behavior. Fortunately, the history table information is quite small — eight bytes per copy per file is quite sufficient, so high levels of replication are not costly in terms of storage.

A simple locking scheme is required to ensure that the file state and the history table are kept in step — the table being locked when it is read and unlocked when it is written back. A more subtle scheme is possible, but from the studies of active file stores concurrent update of replicated files is likely to be very rare in practice [45].

Many areas require further study. In particular, there are several systems administration questions that arise only with replicated files: Who may alter the file's replication? How does the user or system administrator specify the replication — explicitly or by asking for a given level of fault tolerance? Should the positioning of files be decided automatically, by users. or by administrators?

3.7 Summary

A new dynamic technique for maintaining consistency of replicated files has been proposed and the algorithms for logical operations are described in a pseudo-code based on set notation separating the control system into two interacting layers. The algorithm offers:

a) high availability,

b) low storage cost for file replication (practical for two copies),

c) dynamic reconfiguration (all other dynamic strategies change the votes dynamically whereas this algorithm offers flexibility for changing the location and the number of copies dynamically),

d) practicable in the sense that it can be implemented simply as an extension to the existing file system,

e) allows easy extension for user control in replication management.

In the following chapters, the reliable histories algorithm is analyzed and compared with the others for availability, reliability and the likely communication overhead involved. The results of the analysis are very encouraging and these issues are potential areas for further investigation.

Chapter Four

Steady-State Availability

In this chapter the steady-state availability of a replicated file, which is the probability of its being available for access (usually read or write) at any particular moment, is analyzed. The analysis is focused on two problems:

1) the availability provided by the *reliable histories* algorithm concentrating on the minimum requirements for gaining advantage over the other methods,

2) the effect of partitioning on the availability provided by the replication methods.

The second is a very critical issue because most of the work in the literature concentrates on tolerating partitions but none of them considers the effect of partitions on the degree of availability provided [74,64,75,76,63,77,58].

By making various simplifying assumptions about the failure and repair rates of file servers and about the possible failure modes of the network, first, a combinatorial model based on k-out-of-n reliability theory [57,78,79] is developed and thereby the steady-state availability of randomly placed files in a partition-free environment is estimated. The results are compared where applicable with various voting algorithms and the available copies method.

A similar method is later used for the analysis of partitioning in a simple topology where failure of a bridge divides the network into two self-communicating groups of nodes. The results are validated by simulation.

In the second step, a more realistic Markov model is derived for analyzing the availability offered by the *reliable histories* algorithm. Because of the number of states involved, the algorithm becomes too complicated to model when a large cluster of nodes are involved. Therefore, only two copies in a distributed system of maximum five nodes is considered for this part of the analysis. This has given a lower bound on the availability.

In the analysis, a distributed system is viewed as a finite, large number of processing nodes linked by a data network. Each copy of a replicated file resides on a different node. The total number of processing nodes, m, is larger than the number of copies, n, of a replicated file ($m > n$). The nodes or the network may fail independently and the system might become partitioned as a result of bridge failures. When a node fails, a repair process is initiated immediately. This repair process always succeeds. The copy on a recovered node is left as it is. An attempt is made to bring obsolete copies up-to-date during the following write request hence a special recovery procedure following a repair is not required. If the assumptions are different for some methods in the comparison, they are stated where necessary. Each node has its own view about the state of the network (this view indicates which nodes are up and which nodes are down). Among the nodes, these views may be inconsistent. In other words, it is possible that a node (or nodes) is incorrect about the status of another node. The replication algorithm running on a processing node can determine the status of any other node only by receiving a reply to its messages. In the reliable histories approach, the file access succeeds as long as the majority of the nodes are available and at least one of these nodes holds an up-to-date file copy. The number of available history copies is not important since the majority of nodes holds at least one up-to-date history.

4.1 Combinatorial Analysis of Availability

In this section availability expressions are derived for $P(A)$ — the probability that a replicated file is accessible. In the analysis a file replicated n times is assumed to have physical copies located on n distinct file servers, chosen from M, the set of server machines. The total number of server machines in the system is m. The RH method assumes that each file's history table is replicated (and up-to-date) on a majority of these m servers. All the servers fail independently with the same probability in such a way that the probability that a server is up, at any instant, is p. The update and read requests originate at random from any machines in M, which are up. Relaxing these assumptions severely complicates a combinatorial analysis, later Markov modeling (Section 4.2) abandons them in favor of more realistic ones.

In its simplest form, the available copy algorithm makes a file with n replicas available with probability (without considering the histories),

$$P(A_f) = 1 - (1 - p)^n \tag{4.1}$$

As m, the number of file server nodes increases, the update availability of *reliable histories* approaches this. To show this, the following demonstrations are required:

a) A_t, the availability of the history table and A_f, the availability of the file, are asymptotically independent events:

With k table and n file copies chosen from m nodes, the probability that a node holds a copy of both the table and the file is $\frac{k}{m} \times \frac{n}{m}$ which tends to zero as $m \rightarrow \infty$.

b) The probability that a file is available for update, $P(A)$, is asymptotically equal to the probability that the file is available:

For an update to succeed, both the file history and at least one copy of the file must be available. The table, replicated using majority voting, will be available with probability

$$P(A_t) = \sum_{k>m/2}^{m} p^k(1-p)^{m-k}\left[\begin{matrix} m \\ k \end{matrix}\right] \tag{4.2}$$

which tends to 1 as $m \rightarrow \infty$, for $p > 1/2$. Since A_t and A_f are (asymptotically) independent, for large m,

$$P(A) \approx P(A_f)P(A_t) \approx 1-(1-p)^n \tag{4.3}$$

By considering conditional probabilities a more realistic formula can be derived for the reliable histories algorithm. Let $P(A_f^c)$ be the probability that an up-to-date file copy is not available, and let $P(A_t^c)$ be the probability that the file's history is not available. We then have

$$\begin{aligned} P(A) &= 1 - P(A_f^c \cup A_t^c) \\ &= 1 - P(A_f^c) - P(A_t^c) + P(A_f^c \mid A_t^c)P(A_t^c) \end{aligned} \tag{4.4}$$

In the first step, we will show that $P(A_t^c) = (1-p)^k$ when the table is held on k distinct servers. In the following expression $P(N_i)$ [9] is the probability that i nodes are down (not necessarily holding a table copy) and $P(T_i)$ [10] is the probability that all tables are on those i nodes.

$$P(A_t^c) = \sum_{i=k}^{m} P(N_i)\,P(T_i)$$

$$= \sum_{i=k}^{m} \frac{(m-k)!}{(m-i)!(i-k)!} p^{m-i}(1-p)^i$$

put $j=i-k$ and $i=j+k$ and we get

$$= (1-p)^k \sum_{j=0}^{m-k} \frac{(m-k)!}{(m-k-j)!j!} p^{m-k-j}(1-p)^j$$

and since the second part is a binomial expansion [11] we can write,

[9] $P(N_i) = \left[\begin{matrix} m \\ i \end{matrix}\right] p^{m-i}(1-p)^i. = \frac{m!}{k!(m-i)!} p^{m-i}(1-p)^i$

[10] $P(T_i) = \frac{i}{m}.\frac{i-1}{m-1}....\frac{i-k+1}{m-k+1} = \frac{i!}{(i-k)!}.\frac{(m-k)!}{m!}$

[11] $\frac{(m-k)!}{(m-k-j)!j!} p^{m-k-j}(1-p)^j = \left[\begin{matrix} m-k \\ j \end{matrix}\right] p^{m-k-j}(1-p)^j$

$$P(A_t^c) = (1-p)^k \tag{4.4.1}$$

Now it is simple to show that $P(A_f^c) = (1-p)^n$. By working from the observation that

$$P(A_f^c \mid A_t^c) = \sum_{i=0}^{n} P(S_i)\,P(R_i) \tag{4.4.2}$$

where $P(S_i)$ is the probability that $n-i$ copies are on the servers that hold tables, and $P(R_i)$ is the probability that none of the remaining servers hold available copies, from (4.4.1) and (4.4.2) we get [14]

$$P(A) = 1 - (1-p)^n - (1-p)^k + \sum_{i=0}^{n} \frac{k!}{(k-n+i)!}\,\frac{(m-n+i)!}{m!}\,(1-p)^{i+k} \tag{4.5}$$

Table 4.1 compares file availabilities when $p=0.7$ and $p=0.9$ for five common replication strategies: unanimous update, single primary, moving primary, majority voting and available copies; with the results of reliable histories from (4.5) with $m=10$. Figures for n = 2,3,4 and 5 are shown. As the results in the table have shown, even with high failure rates (0.3), for small n, the history table reduces availability only 0.1 percent (three point accuracy) when compared to the original available copies algorithm. This availability is still considerably better than the other methods.

	p=0.7				*p=0.9*			
Method	*n=2*	*n=3*	*n=4*	*n=5*	*n=2*	*n=3*	*n=4*	*n=5*
U.Update	*0.490*	*0.343*	*0.240*	*0.168*	*0.810*	*0.729*	*0.656*	*0.590*
S.Primary	*0.700*	*0.700*	*0.700*	*0.700*	*0.900*	*0.900*	*0.900*	*0.900*
M.Primary	*0.490*	*0.784*	*0.916*	*0.969*	*0.810*	*0.972*	*0.996*	*0.999*
M.Voting	*0.490*	*0.784*	*0.651*	*0.837*	*0.810*	*0.972*	*0.948*	*0.991*
A.Copies	*0.910*	*0.973*	*0.992*	*0.997*	*0.990*	*0.999*	*0.999*	*0.999*
R.Histories	*0.909*	*0.972*	*0.991*	*0.996*	*0.989*	*0.998*	*0.999*	*0.999*

Table 4.1 — *Availability offered by various replication schemes.*

[14] Hint: $P(A_f^c \mid A_t^c) = \sum_{i=0}^{n} \frac{k}{m}.\frac{k-1}{m-1} \cdots \frac{k-n+i+1}{m-n+i+1}.(1-p)^i$

4.2 Stochastic Analysis of Availability

In this section, the simplistic failure model of the combinatorial analysis is abandoned by using Markov modeling. In the above analysis the independent nodes were considered being up with probability p. Here, the availability behavior of the voting methods and the reliable histories algorithm are analyzed considering node failures and repairs as independent events. Since the voting method requires a minimum of three copies for acceptable improvement in the availability, the most reasonable solution is to assign equal weights to all copies and to have both read and write quorums equal to two. The file then remains available as long as two out of three copies, in other words majority of the nodes, are accessible. This method will be called majority voting (MV). The other method in comparison is voting with witnesses(VWW) in which one of the three copies is replaced by a witness copy. Since these methods provide the same availability for read and write, in the following analysis the availability derived stands for both read and write.

The state-transition-rate diagram is a network of states representing different combinations of machine and file copy availability. Events such as machine failures and repairs, as well as file updates, may cause a transition from one state to another. In the following analysis, it is assumed that individual node failures and individual repairs are independent events distributed according to a Poisson law. In other words, the probability that a given node will experience no failure during a time interval of duration t is $e^{-\lambda t}$ where λ is the failure rate. Similarly, the probability that a given node will be repaired in less than t time units is $1 - e^{-\mu t}$ where μ is the repair rate. File updates occur at rate u, again obeying a Poisson law. It is assumed that repairs are initiated as soon as a failure occurs (and at no other time) and that a failure is only possible once a node has recovered.

The availability of the system A is the limiting value of the probability $p(t)$ that the replication system will be operating correctly (can access to the file while preserving consistency) at time t.

$$A = \lim_{t \to \infty} p(t)$$

If transitions from state S_i to state S_j occur at a rate r, then the expected number of transitions into state S_j is just $P_{S_i} r$, the probability of the system being in state S_i multiplied by the rate. For the system to reach equilibrium, the expected number of transitions into and out of each state must be equal, giving rise to a set of simultaneous equations in the probabilities P_s that the system is in state s. A file will only be available when the system is in any one of known number of states. The file availability is determined by summing the probabilities of the system being in any of these states in which the file is available. The availability model which is defined as states and transitions between these states is solved using a software package developed by Sahinoglu [80,81,82]. This package generates the matrix, $Q = (q_{i,j})$, infinitesimal generator of the Markov process $P_i(t)$ which satisfies the forward equation

$$(d/dt)P_j(t) = P_i(t)q_{i,j}$$

This Markov process yields a solution following the method given in Appendix B. As the number of states increases rapidly as more file servers are modeled, here, only an analysis of three and five server systems are given. The behavior of the reliable histories method with two copies is compared with the availability provided by majority voting with three copies and voting with witnesses (two copies and one witness). The analysis is restricted to the case where all nodes regardless of containing a copy or not have equal failure rates λ and equal repair rates μ. This is not a restriction of the method but allowing different rates for each node would complicate the equations further. Below two models are derived. In the first model, the system has only three processing nodes and two or three of them hold file copies (depending on the algorithm). In the second model, the number of processing nodes is five and two (or three) of them hold a copy.

The aim of this analysis is to determine the limiting conditions of the system operating under reliable histories algorithm in order to gain advantage (in terms of increased file accessibility and reduced storage cost) over other methods.

4.2.1 Modeling Three Nodes

The state space of the model defines a finite-state Markov chain because there is a finite number of states representing the system where each state is shown by a letter and two numbers. The letter represents the possible states of the two copies.

'S' denotes states in which both copies have the same version number.

'D' the copies have different version numbers but the up-to-date copy is available.

'W' the copies have different version numbers and only the out-of-date copy is available.

The first subscript denotes the number of available copies (0 to 2) and second identifies the number of nodes that are up in that state (0 to 3). Clearly, only a limited set of combinations is possible. For instance, the letter *W* only appears when one copy is up and the other is down.

The following rules are obeyed:

1. One or more failure transitions can only occur from states having at least one up node (not necessarily holding a copy). The rates at which the transitions occur are proportional to the number of up nodes and copies which are therefore susceptible to failure. For instance: state S_{23} corresponds to the case where all nodes are up therefore both copies are up and up-to-date. S_{23} has two failure transitions; one is to state S_{12} with rate 2λ which corresponds to the failure of either copy. Two nodes remain available and only one of them is holding a file copy. The other transition is to state S_{22} with rate λ which corresponds to the failure of the node not holding a copy. Two nodes remain available and both of them hold a copy.

2. One or more repair transitions originate from every state having at least one node down (not necessarily holding a copy). The rates of the transitions are proportional to the number of down nodes and copies. Repairs do not include any recovery operation. For example: state W_{12} has a single repair transition with rate μ to state D_{23} when the down copy is repaired. The copies still represent different versions.

3. The only possible transition from an 'S' state to a 'D' state and vice versa occurs when the file is updated. The rate at which this transition occurs is thus given by the rate u, the update rate for the replicated file.

The STR diagram associated with a file replicated twice in a three node system using the *reliable histories* method is shown in Figure 4.1.

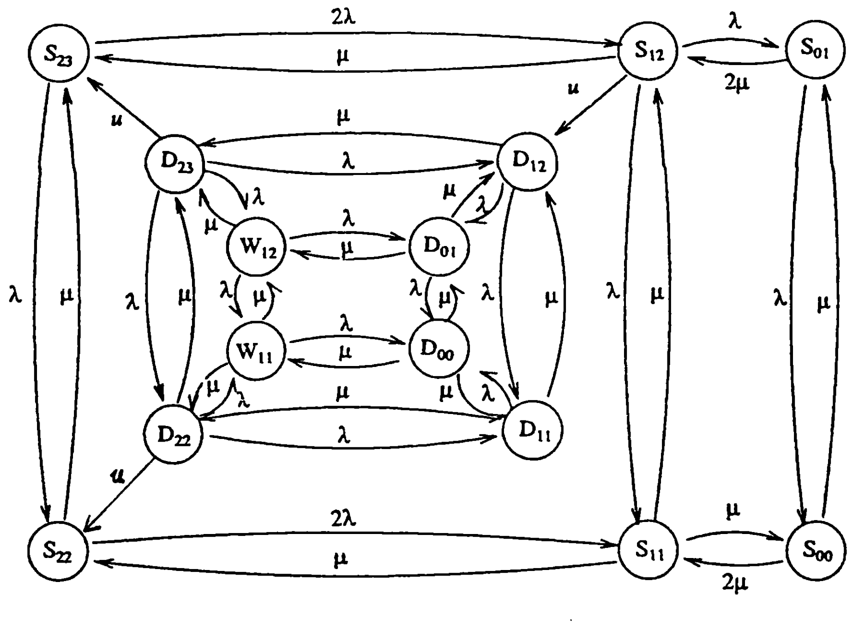


Figure 4.1. *States associated with RH algorithm when m=3*

With only three nodes, two of which carry copies of the file, there will be a total of 14 states. All transitions from/to states obey the rules given above. The file will be available (for read or write)) in only 6 out of the 14 possible states: S_{23}, S_{22}, S_{12}, D_{23}, D_{22}, D_{12}. For this system the equilibrium state probabilities will obey the following equilibrium conditions:

$$\begin{aligned}
P_{S_{23}}3\lambda &= (P_{S_{22}} + P_{S_{12}})\mu + P_{D_{23}}u \\
P_{S_{22}}(2\lambda + \mu) &= P_{S_{23}}\lambda + P_{S_{11}}\mu + P_{D_{22}}u \\
P_{S_{12}}(2\lambda + \mu + u) &= 2P_{S_{23}}\lambda + (P_{S_{11}} + 2P_{S_{01}})\mu \\
P_{D_{12}}(2\lambda + \mu) &= P_{S_{12}}u + (P_{D_{11}} + P_{D_{01}})\mu + P_{D_{23}}\lambda \\
P_{W_{12}}(2\lambda + \mu) &= (P_{W_{11}} + P_{D_{01}})\mu + P_{D_{23}}\lambda \\
P_{S_{11}}(\lambda + 2\mu) &= (2P_{S_{22}} + P_{S_{12}})\lambda + 2P_{S_{00}}\mu \\
P_{D_{11}}(\lambda + 2\mu) &= (P_{D_{12}} + P_{D_{22}})\lambda + P_{D_{00}}\mu \\
P_{W_{11}}(\lambda + 2\mu) &= (P_{W_{12}} + P_{D_{22}})\lambda + P_{D_{00}}\mu \\
P_{S_{01}}(\lambda + 2\mu) &= P_{S_{12}}\lambda + P_{S_{00}}\mu \\
P_{D_{01}}(\lambda + 2\mu) &= (P_{D_{12}} + P_{W_{12}})\lambda + P_{D_{00}}\mu \\
P_{S_{00}}3\mu &= (P_{S_{11}} + P_{S_{01}})\lambda \\
P_{D_{00}}3\mu &= (P_{W_{11}} + P_{D_{11}} + P_{D_{01}})\lambda \\
P_{D_{22}}(2\lambda + \mu + u) &= (P_{D_{11}} + P_{W_{11}})\mu + P_{D_{23}}\lambda \\
P_{D_{23}}(3\lambda + u) &= (P_{D_{11}} + P_{D_{22}} + P_{W_{12}})\mu
\end{aligned}$$

The transition matrix which is derived from the above equilibrium state conditions is not given here because of its size and complexity. The STR diagrams for the MV and AC algorithms can be found in Appendix B.

Three methods are compared. MV has been employed assuming that all nodes keep a copy (as it requires a minimum of three copies). In VWW, one of the copies is assumed to be a witness. This method assumes that when a copy repairs from a failure it copies to itself the up-to-date version. RH also requires any two out of three nodes in order to access the history. As any two of the nodes will contain at least one file copy, the file will be accessible. Therefore, RH behaves in a similar way to VWW when there are only three nodes in the system. The only difference is, RH does not apply a recovery operation. It assumes that the copy will be brought up-to-date in the next successful write after the node is repaired. As the results show, both voting algorithms perform better than RH. Therefore, three nodes are not enough to gain any advantage over either voting approaches. Figure 4.2 illustrates the availabilities provided by RH with 2 copies, MV with three copies and VWW in a system of three processing nodes.

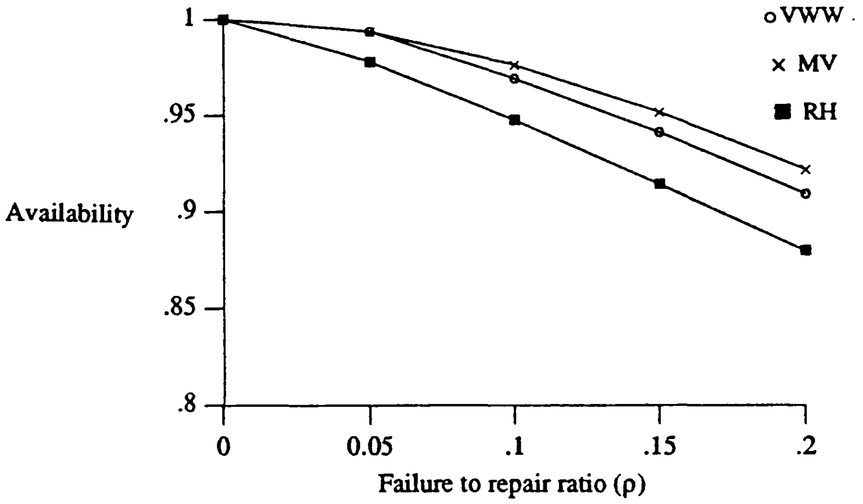


Figure 4.2. *Analytic results when m = 3*

The next step is to compare the results when there are five processing nodes in the model. In the next section, the availabilities in a five processing node system are presented and the improvement in the *reliable histories* method as the number of nodes is increased is shown. Later, the results of the combinatorial analysis are compared with the results of the Markov analysis.

4.2.2 Modeling Five Nodes

The availability model for five nodes is derived in the same way as done for the three nodes in the previous section. [15] With five nodes, two of which carry a file copy, there will be a total of 28 states. The file will be available (for read and write) in 9 out of the 28 possible states: $S_{25}, S_{24}, S_{23}, D_{24}, D_{23}, D_{13}, D_{14}, S_{14}, S_{13}$. The file access succeeds as long as any three of the processing nodes are up and one of them holds an

[15] An example transition in the five-node model: S_{25} has two failure transitions; one is to state S_{14} with rate 2λ by failure of either copy and to state S_{24} corresponding to the failure of one of the three nodes not holding a copy. These transitions therefore occur with the rate of 3λ.

up-to-date copy of the file. The requirements of the other algorithms are the same as in the previous model. The equilibrium state conditions of the system when m=5 and the details of the model are given in Appendix B.

Figure 4.3 illustrates the availability when m=5. As the results show, RH performs slightly better than MV with three copies and requires only two replicas.

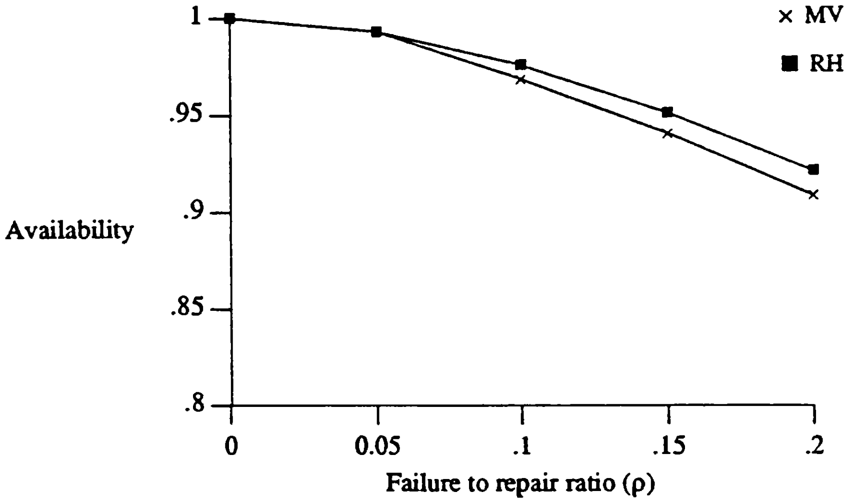


Figure 4.3. *Analytic results when m = 5*

4.2.3 Conclusion

The *reliable histories* algorithm with two file copies provides availability between two bounds determined by the number of nodes in the system. The lower bound is offered when m=5 (five is the minimum number of nodes required to gain availability advantage over the voting methods) and the upper bound is reached when the history table is assumed to be always available.

When m=5, the algorithm provides availability very close to MV (slightly better), but it requires only two copies whereas MV uses three copies. Therefore two advantages are acquired (low storage cost and higher availability) at the lower bound. The stochastic analysis has given the lower bound which specified the minimum requirements of the algorithm. The maximum availability is reached when the file's history is always available which is equal to the availability provided by AC. This is the optimal availability for two copies. As the combinatorial analysis assumes a large *m*, it has given the upper bound of the availability that can be reached (the results were presented in Section 4.1). Figure 4.4 shows that as the number of nodes is increased results obtained from Markov models converge to the results from combinatorial analysis and Table 4.2 presents numerical changes in availability as the number of nodes is increased.

	m=3		*m=5*		*m=10*		*m=50*	
ρ	*A*	*S*	*A*	*S*	*A*	*S*	*A*	*S*
0.05	0.996	0.988	0.997	0.994	0.997	0.996	0.997	0.996
0.1	0.984	0.970	0.989	0.983	0.989	0.983	0.990	0.988
0.2	0.957	-	0.969	-	0.972	-	0.972	-

Table 4.2 —*Availability offered by RH for various m values*

When m=3, the availability digresses more from the combinatorial results than the availability obtained when m=5. This is because of the difference between the failure assumptions in both analyzes; combinatorial derivation is based on the assumptions that $k \gg n$ and $k \geq m/2$. As m increases both results converge.

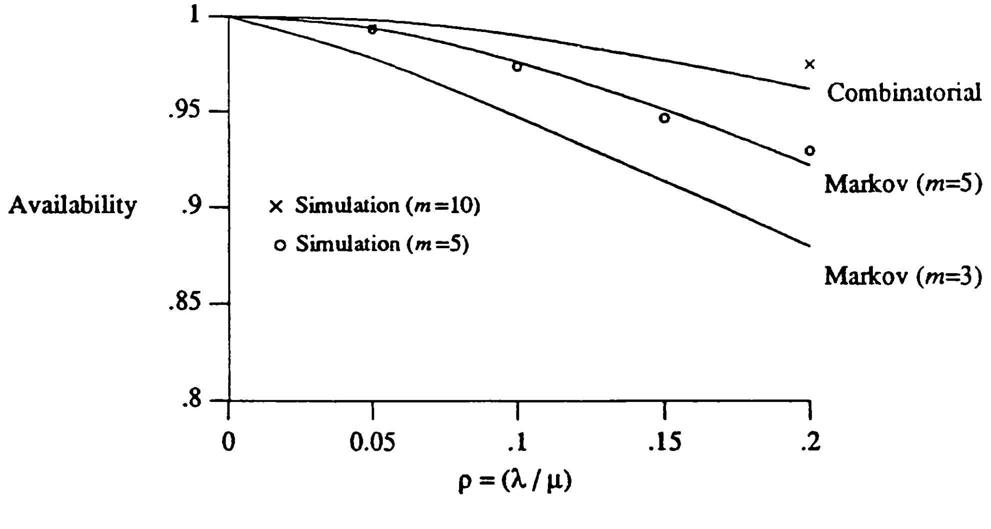


Figure 4.4. *Comparison of availability obtained by different techniques*

4.3 Managing Replicas in a Partitioned System

A *partitioning* of a distributed file system occurs when the nodes in the network split into groups of communicating nodes due to bridge failures. The nodes in each group can communicate with each other, but no node in one group is able to communicate with nodes in other groups. Each such group is referred to as a *partition*.

As discussed in Chapter Two, the available copies algorithm fails to continue functioning correctly in a partitioned system. Generally, algorithms which function correctly in the face of partitions permit a file to be accessed only in one partition. They share the philosophy that mutual consistency is of greater importance than availability. Some powerful dynamic voting schemes [67, 72] have recently been suggested which overcome the drawback that failures can occur in such a way that no updates can be performed anywhere in the system until these failures are repaired. The challenge is to improve availability as well as preserving mutual consistency. But these methods require that partitioning failures are distinguishable from node failures and these failures are recognized instantaneously.

Although there were attempts to improve availability during partitioning I have not seen any analysis of the degree to which the availability is reduced in a partitioned system. In the following sections, availability offered by replication schemes in partitioned networks is analyzed. The term 'partition-free' is used where it is assumed that the system never becomes partitioned.

4.3.1 Combinatorial Approach to Simple Partitioning

The file availability in the presence of network partitions is harder to study mathematically. In this section a simple partitioning case is analyzed as an extension to the combinatorial analysis given in Section 4.1. Later, the results are generalized by simulating partitioning failures in various topologies.

The system is a series of nodes on two subnets connected by a bridge. These subnets are called net-1 and net-2. An example topology is illustrated in Figure 4.5.

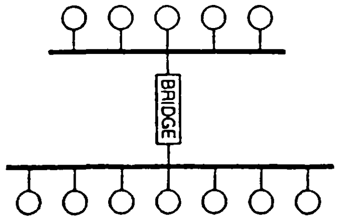


Figure 4.5. *A simple network topology*

It is assumed that communication links never fail. Each node has the same independent probability p of being up and the bridge has the probability p_r of being up. Therefore, $(1 - p_r)$ is the probability that the system is partitioned into two self communicating groups. The availability of the file in this simple system is the sum of the availabilities when the bridge is down, $P(A_d)$, multiplied by the probability of the bridge being down, $(1-p_r)$, and the availability when the bridge is up, $P(A_u)$, multiplied by the probability of the bridge being up, p_r. (Availability when there is no partitioning was derived in Section 4.1). The updates are initiated from randomly chosen nodes. In RH, If the bridge is down, the file can only be available to the users sitting on the side of the majority of the nodes. If the number of nodes is even and node distribution is symmetric, the file becomes unavailable during partitioning and equation (4.6.1) given below cannot be applied.

$$P(A) = (1 - p_r)P(A_d) + p_r P(A_u) \tag{4.6}$$

where (for the *reliable histories*)

$$P(A_d) = \frac{m_2 p}{m} \sum_{i>m/2}^{m_2} p^i (1-p)^{m_2-i} \begin{bmatrix} m_2 \\ i \end{bmatrix} \tag{4.6.1}$$

In formula (4.6.1), m_1 and m_2 are the number of nodes on the subnets net-1 and net-2 respectively, under the assumption that $m_2 > m_1$. This formula gives the probability of the history table and the copy being available on net-2 and at the same time update being originated from this subnet. $P(A_u)$ is the availability when there is no partitioning in the system. From (4.1) and (4.6.1) we get,

$$P(A) = 2 - p(2+2p_r+p^{n-1}+\frac{m_2}{m}) + p_r(2+p^n) + \sum_{i>m/2}^{m_2} p^i(1-p)^{m_2-i}\begin{pmatrix} m_2 \\ i \end{pmatrix} \qquad (4.7)$$

This shows that, even with simplified failure assumptions, the analysis of partitioning is very complicated. All through the following analysis, MV's behavior is investigated assuming that the file has three copies [14] whereas in RH it has only two copies. In Figure 4.6 the file availability provided by RH and MV are compared as a function of p for different values of p_r. [15] The following assumptions are made:

a) There are a total of 12 nodes in the system where m_1=5 and m_2=7,

b) RH has two copies: one is on net-1, the other is on net-2,

c) MV has three split copies: one is on net-1 and the other two are on net-2.

The partitioning has shown very little effect on the algorithms' availability performance when p_r=0.95 for this specific case. In the next section the algorithms will be analyzed in the same topology within a broader context.

[14] MV has no practical use with two copies.

[15] RH is plotted for three values; p_r=1 means that the system is partition-free, p_r = 0.95, the probability of partitioning is 0.05 and p_r = 0.7, the probability of partitioning is 0.3. MV is plotted for the first of two of these p_r values.

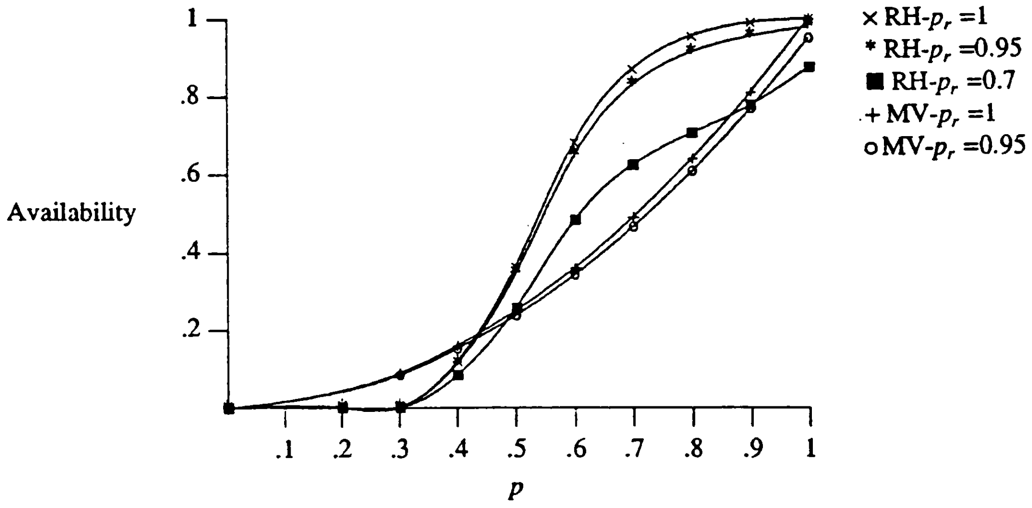


Figure 4.6. *Node availability vs File availability*

4.3.2 Resilience to Copy Placement in Partitioning

In this section, the variance in availability for different configurations is studied as a function of partitioning probability $(1 - p_r)$. The effect of allocating different number of nodes between the two subnets are measured. Three distributions were assumed where the total number of nodes is 12 as in the previous section. Each distribution has been analyzed for two configurations having two different placements of copies: one in which all copies are on the same subnet and the other in which they are split. In the analysis of RH, the file has two copies and in the analysis of MV it has three copies as in the previous analysis. The distribution of nodes were chosen to represent all possible cases: a symmetric distribution, a slightly asymmetric distribution and a highly asymmetric distribution. In the symmetric distribution, there are 6 nodes on both subnets (6-6). In the slightly asymmetric distribution, there are 5 nodes on net-1 and 7 nodes on net-2 (5-7). In the highly asymmetric distribution there are 2 nodes on net-1 and 10 nodes on net-2 (2-10).

In the first configuration, (config-1), RH assumes that one copy is on net-1 and the other copy is on net-2 whereas MV assumes two copies are on net-2 and one copy is on net-1. In the second placement, (config-2), both copies are on net-2 for RH and two copies on net-1 and one copy is on net-2 for MV.

For this system, availability changes in the form of a straight line as a function of partitioning probability. While p_r determines the slope of the availability line, p affects the starting position (availability when $(1 - p_r = 0)$. In the symmetric distribution, MV performs better than RH, but both algorithms' performance stays the same for all configurations of copy placement; when the bridge is failed none of the subnets hold the majority of the nodes. As the file becomes unavailable in any configuration for RH, copy placement does not affect the overall availability. In this distribution, MV performs better than RH because it allows access on the subnet holding a majority of the copies (which is two in this case) during partitioning. Therefore in symmetric distribution, the variance of availability as a function of copy placement is zero for both algorithms. This changes interestingly for asymmetric distributions. Figure 4.7 compares the availabilities for different copy placements, config-1 and config-2, in (5-7) and (2-10) node distributions respectively. As the results illustrate MV and RH are very close in (5-7), the difference is clearer in the (2-10) case.

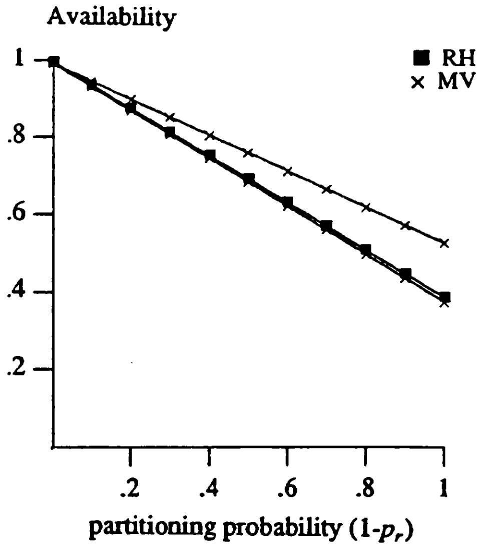


Figure 4.7(a) *Node distribution (5-7)*

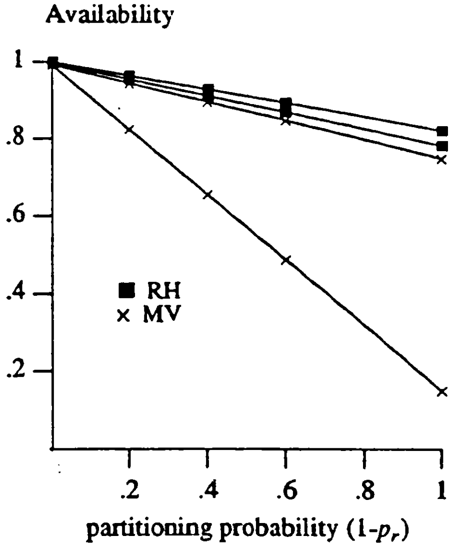


Figure 4.7(b) *Node distribution (2-10)*

If the nodes are equally or slightly asymmetrically distributed over two subnets majority voting performs better. If one of the networks holds majority of the nodes as in (2-10), RH performs better than MV. Although it performs better in some configurations copy replacement causes a dramatic change in the availability of MV whereas RH is more resilient to different configurations in all cases. After the analysis of reliability this result will be reviewed in the next chapter. The availability provided by the reliable histories method (with two copies) has been verified by simulation. The simulation results are illustrated in Figure 4.8 for $p \geq 0.9$.

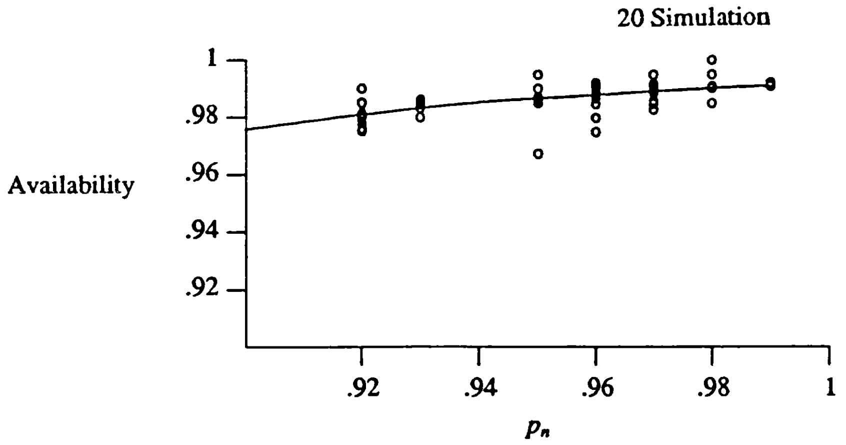


Figure 4.8 *Availability when $(1 - p_r) = 0.05$ — (simulation + analytic)*

As this mathematical approach cannot go beyond a simple topology, only the simulation results for the availability in a partitioned network consisting a large cluster of nodes are given. The simulated topology was given in Figure 1.1. In this topology bridge failures split the system into several self-functioning groups. The copy placement and the local node are randomly chosen at each request during the simulation period. Interestingly, random copy placement has shown very little effect on the availability in this case too. In order to generalize these results, the distribution of the availability was obtained. Figure 4.9 illustrates the variation in availability at p=0.9 for two p_r values;

0.9 (bridge reliability is the same as the node reliability) and 0.95 (bridges are 5% more reliable than nodes).

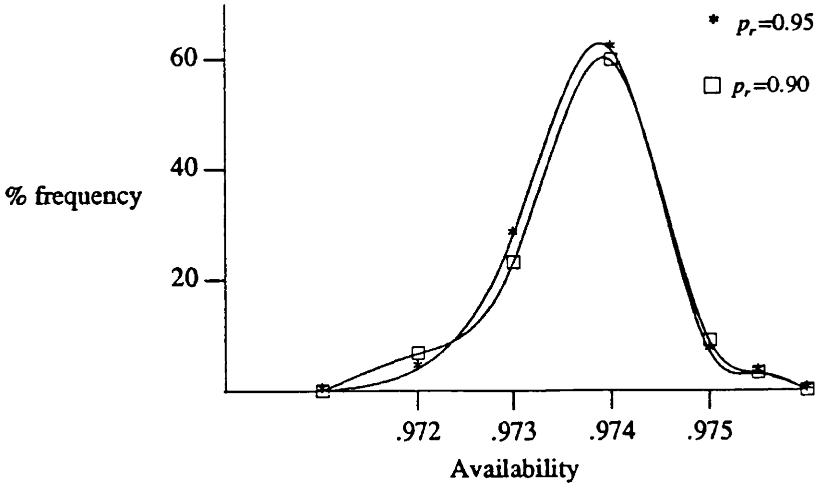


Figure 4.9. *Distribution of availability at* p=0.9

4.4 Summary

The availability at steady-state was analyzed using two different analytic approaches and by simulation. Two issues were investigated: the degree of availability provided by RH and the minimum requirements for gaining advantage over the other methods. It has shown that $m>5$ is the boundary requirement for a file with two copies to provide better availability than any variation of voting with three copies. When $m=5$ both methods provide similar availability but voting requires three copies where RH requires only two.

The second issue is the effect of partitioning on the availability. First, a simple topology was studied extending the combinatorial analysis and later results on the effect of random file configurations were verified by simulating a large scale system. It was shown that, in some topologies, partitioning reduces the availability provided by voting methods dramatically whereas RH is more resilient to configurational changes on average.

Chapter Five

Reliability in Partitioned Systems

Reliability of a replicated file is defined as the probability that the file will be continuously available for a given length of time [55]. It is therefore a function of time, R, with $R(0)$ being the steady-state availability analyzed in the previous chapter and $R(\Delta t)$ being the probability that the file will be continuously available for time Δt. Availability has received much more attention, in part because its analysis is more tractable than that of reliability. Although Long *et al* [58] analyzed the reliability of regeneration-based consistency schemes under the assumption that the network never partitions, the affect of partitioning on the reliability of replicated data is a problem which has not yet been clearly understood. Reliability offered by the consistency schemes is far harder to analyze theoretically for partitioned systems, it is almost impossible for large number of nodes. Therefore a file system simulation has been built in order to measure the reliability provided by various algorithms with that proposed. The results have shown that although the availability afforded by all replication control protocols is quite similar for low fail probabilities of individual nodes, the reliabilities vary greatly. Here, reliability afforded by the *reliable histories* and voting algorithms are studied in some topologies where bridge failures divide the system into many self-communicating partition sets.

This study includes a sensitivity analysis of reliability (provided by different algorithms) to the changes in network topology and to the placement of copies. The results show the degree of change in the algorithm's behavior as the failure mode of the network is changed. As networks grow and evolve, subnets can become bridged together and machines moved from subnet to subnet; more often than not as required by geographical constraints. If the network is an interconnection of sub-networks by bridges, relays or gateways, the failure of a single node can cause a partition, making a replicated file completely unavailable or unavailable to a large part of the network. When the copy replacement is random, the replication algorithm should not be adversely affected by configurational changes. MV and RH algorithms' behavior towards the changes are presented in various graphs. The results are generalized in two graphs illustrating the variation in reliability at 1000 time units, R(1000), and the distribution of reliability decay constants. The chapter has concluded with an analytical model for reliability. This model is also extended to include regeneration of file copies under the assumptions that the copies are regenerated on a spare node when the server node fails, and there are infinite number of spare nodes. A generalized analytical model for regeneration with a limited number of spare nodes is given in Appendix C. The state-transition-rate diagrams of the *reliable histories* method when $m=3$ and $m=5$ are also enclosed. This analysis has been done in line with Long's study on regeneration with finite number of spares. As it is not an original work but an adaptation of it, it has been quoted as an appendix.

5.1 System Model for Reliability

The replication control algorithms which were analyzed for availability are studied here for reliability. A software simulation was built for this study. The results have

[15] Each reliability graph declines with a constant decay value as a function of time period. The variance of decay constant in different configurations shows the degree of change in the reliability behavior.

shown that the parameters of the failure model, intercrash period, (*mtbf*), and failure to repair ratio, (ρ), have different degrees of affect upon the algorithms. Therefore in some graphs in the chapter reliability, $R(\Delta t)$, is plotted for period Δt (elapsed time) in multiples of the mean intercrash period in order to clarify this effect. This scaling has enabled us to compare the reliability of the data offered by the consistency schemes as a multiple of the system's reliability.[16]

The data points shown on the reliability graphs were obtained by simulating the failures and repairs of a system of m nodes, n of which hold a physical copy of the file and noting the time at which the scheme would first deny access to the logical file. The process was repeated for a simulation period of 50,000 time units. In the experiments, various mtbf and mttr pairs (corresponding to a large group of failure to repair ratios in the system) are used which cover failure models of ρ between 0 and 0.2.

The failures of individual nodes and bridges are characterized by a Poisson process: exponentially distributed with the mean values in the range $(100, 300)$ time units. The period for node repairs (mttr) were assumed to be more deterministic: normally distributed with the mean values between 2.5(st=0.5) and 20(st=4).[17]

The environments were chosen to represent systems in which failure of a bridge cause a part of the network to become unavailable to the other parts of the system. A series of bridge failures may divide the system into several self-functioning sets of nodes. In the simulated environment the number of processing nodes varied between 10 to 50. The topologies used in the analysis will be described later. The schemes that are compared (where possible) are the majority voting (MV), the voting with witnesses (VWW),

[15] $\rho = \frac{1-p}{p}$

[16] For example with single copy, the data's reliability is the same as the node's reliability holding the data. Therefore in this case failure to repair ratio has no affect. As the number of copies are increased, reliability becomes a factor of ratio rather than mean intercrash time.

[17] These values are randomly picked but they correspond to a large group of ratios. For example: a system where mtbf=100 and mttr=10(2) has $\rho = 0.1$ and a system where mtbf=200 and mttr=5(1) has $\rho = 0.025$.

the available copies (AC) and the reliable histories (RH) algorithms.

The plotted simulation results are the mean values of 20 runs. Each run simulates file accesses[20] over the simulation period. The requests are assumed to be initiated from randomly chosen nodes. The graph in Figure 5.1 shows a good agreement over 50,000 as simulation period.

Another verification of the simulation was tabulated in Table 5.1. This table shows a good agreement between measured and calculated *k*-out-of-*n* surviving nodes for different values of mean time between failures and mean repair periods. These values correspond to *p* values 0.9, 0.934, 0.983 respectively where *p* is the probability of a node being up.

	mtbf=100, mttr=10		*mtbf=100, mttr=7*		*mtbf=300, mttr=5*	
up	*Measured*	*Calculated*	*Measured*	*Calculated*	*Measured*	*Calculated*
10	*39.06*	*38.51*	*50.05*	*50.83*	*86.02*	*84.24*
9	*38.32*	*38.55*	*37.76*	*35.58*	*13.00*	*14.56*
8	*16.48*	*13.88*	*10.34*	*11.21*	*0.93*	*0.74*
7	*4.32*	*4.63*	*2.04*	*2.09*	*0.03*	*0.05*
6	*0.82*	*0.81*	*0.14*	*0.26*		

Table 5.1 — *% of time nodes were up*

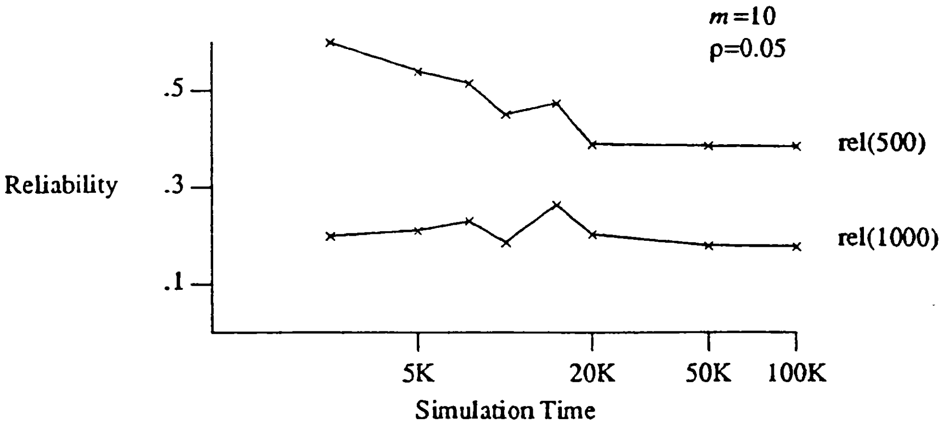


Figure 5.1. *Simulation for various time periods*

[20] As all the algorithms studied provide the same availability for both, the accesses are not specified as read and writes.

5.2 Reliability in Partition-Free Networks

The advantage of replication over a single copy is more observable in terms of reliability than in terms of availability. In the previous chapter, it was shown that going from single copy to two copies improves the availability to a great extent and that increase is greater than that obtained by going to further copies (three or more). In the rest of this chapter a similar result will be shown for reliability.

In this section, the reliability will be estimated without considering the affect of partitioning. This analysis has been done by assuming that the bridges are always up during the simulation period. The only failure that can occur in the system is clean node crashes. This allowed us to compare the reliability provided by the available copies method as well as voting methods. The system is assumed to have 10 processing nodes some of which hold a complete copy of the file. This choice of *m* is backed up by the upper bound availability analysis in Section 4.2. With reference to Figure 5.2, simulation results in various sized systems have supported this claim.

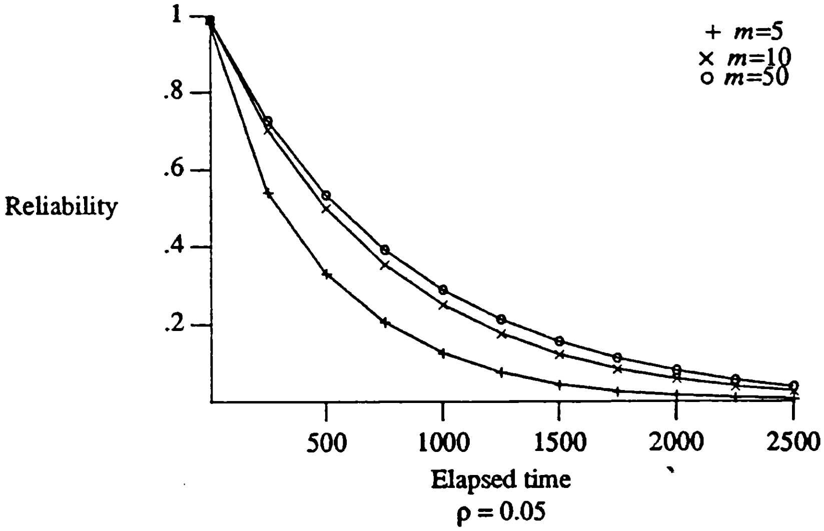


Figure 5.2. *Reliability offered by RH for various m*

Two experiments have been carried out. In the first experiment the reliability offered by RH is found for various failure ratios. Figure 5.3 illustrates the results.[21]

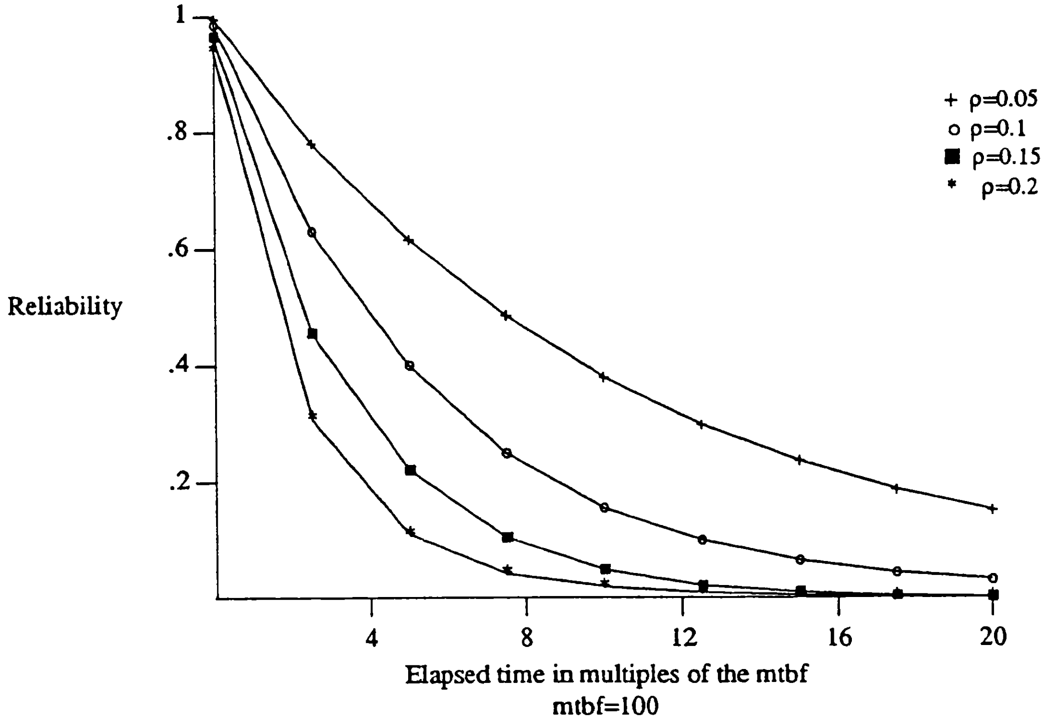


Figure 5.3. *Elapsed time vs Reliability (RH)*

In the second experiment, RH and other algorithms are compared with the reliability of a single copy as a reference. The file is assumed to have two full physical copies in AC and RH algorithms. AC assumes that the file has two randomly placed copies. The nodes holding these copies maintain status lists. The nodes are aware of the status of each other. Updates are propagated to all available copies. When a node holding a copy recovers from a failure, it is configured in by copying the up to date version from the other copy before accepting any request. RH algorithm maintains two randomly placed copies as AC but it also requires a majority of the nodes in the system to be up. It allows access to the file as long as one of the copies is up and a majority of the nodes are operating. The results show that AC and RH behave very closely. As in the availability

[21] The mean intercrash period (mtbf) is taken 100 time units and repair time (mttr) changes from 5(st=1) to 20(st=4). The range corresponds to ρ between 0.05 (repairs are 20 times faster than failures) and 0.2 (repairs are 5 times faster than failures).

analysis, MV algorithm has three randomly placed copies.[22] It only considers the failure of participating nodes: as long as two of the copies are operating the file is available. Reliability offered by the algorithms were illustrated in Figure 5.4(a) and Figure 5.4(b), for two different failure models. Following these experiments, the results are generalized in Figure 5.5 by showing the decay constants of reliability graphs for various failure ratios. As a summary, these experiments have shown that RH behaves very close to AC and with only two copies it provides better reliability than MV with three copies.

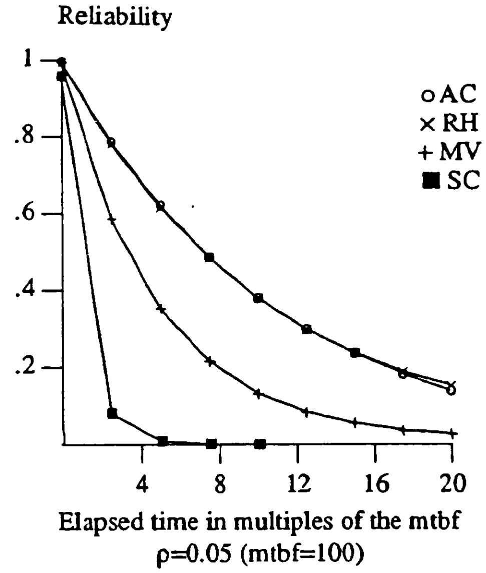


Figure 5.4(a). *Elapsed time vs Reliability*

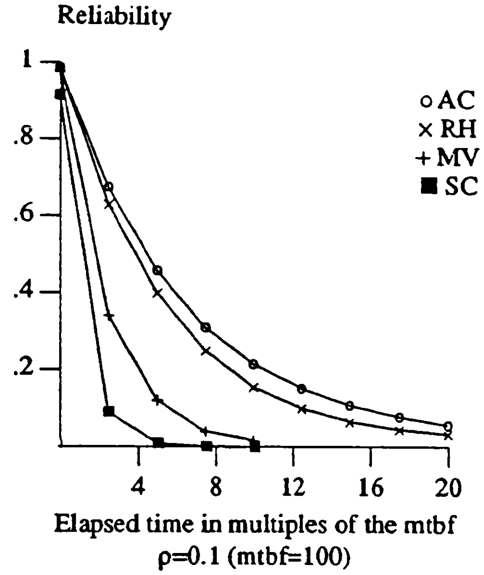


Figure 5.4(b). *Elapsed time vs Reliability*

5.3 Effect of Partitions on The Reliability

The effect of network partitions on reliability is investigated by running the schemes in two network topologies (Figure 1.1 and Figure 5.6). In these topologies the system becomes partitioned when one or more bridges fail. If a bridge connected to a subnet is up then all the operating nodes on that subnet are available to the other parts of

[22] MV algorithm requires three copies for any practical use. The aim is to show that RH provides better availability even with two copies than MV with three copies.

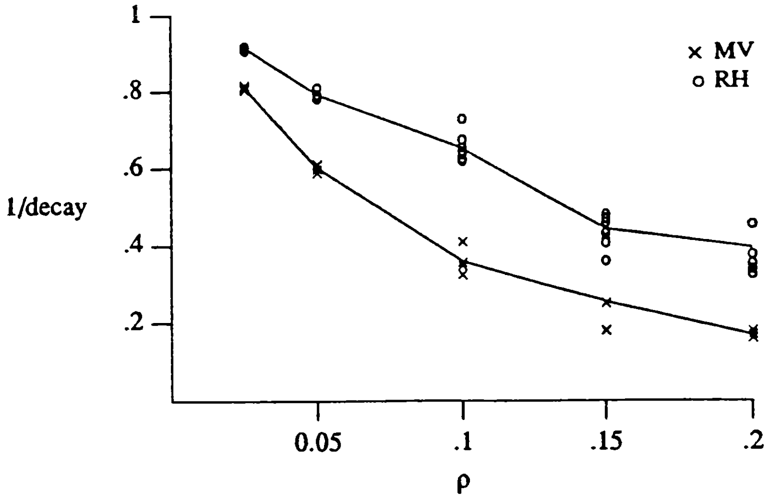


Figure 5.5. *Failure ratio vs decay constant*

the system. Since the available copies algorithm is not applicable in this environment, only the voting algorithm was compared with RH. The algorithms' behavior towards random copy placement is summarized later in Section 5.4. Here, a number of file configurations are analyzed to study the effect these have on reliability. The effect of network configuration and copy placement needs much more further work. Although the simulation can generate random copy placement for a given topology, the notion of ''random'' topology should also be clarified in future. More generalized approaches would then be possible.

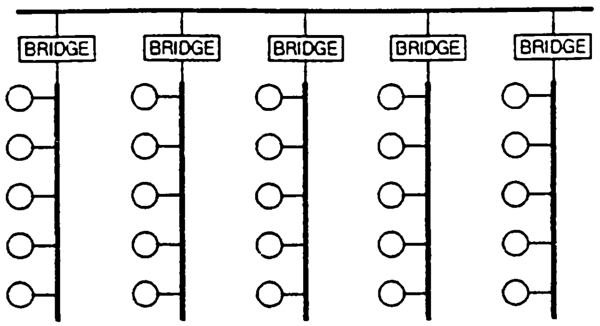


Figure 5.6. *A distributed environment (Topology-2)*

The analysis has been carried out in a system where nodes repair 40 times faster than they fail (ρ=0.025).[21]

In partition-free networks, simulating a failure model and checking the accessibility of a file's history table was simple as the algorithms do not have different behavior according to the user's position in the network. The partitioning analysis is rather complicated. The accessibility of the nodes changes according the user's position. This analysis has been done by representing the topologies as graphs. Every component of the system, bridges, communication links and nodes are represented as the nodes of the topology graph. An edge between two nodes shows a two-way connection between them. Each node of the graph has independent probability of being available. The number of available nodes to a node depends on the position of that node in the graph. Figure 5.7 illustrates the graphs of the two simulated topologies.

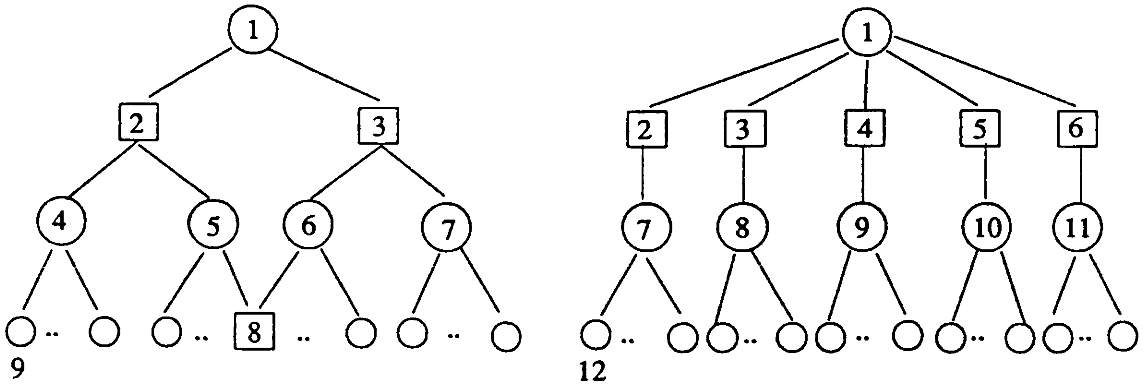


Figure 5.7. *Graph representation*

The number of available nodes is determined by walking through the graph starting from a random node. If a node in the graph is unavailable, all the connected edges

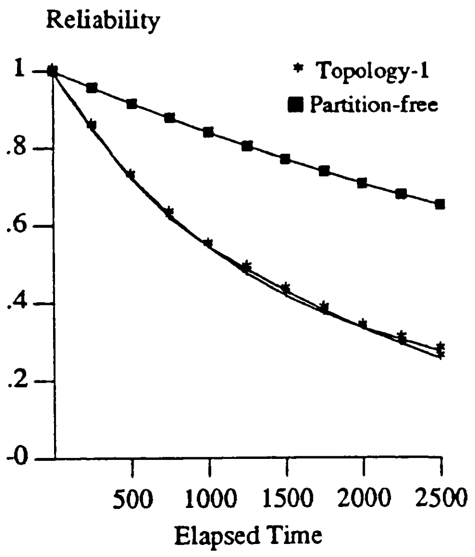
[21] This ratio is simulated with an exponential mtbf=300 time unit where mttr for a normally distributed repair periods is 7(2) time unit. This comparative experiment has aimed to investigate the change in the reliability as file configuration is changed, rather than the reliability behavior as a function of failure ratio. Therefore a typical model is chosen for the analysis.

become unavailable and no further nodes can be visited on that path. The number of nodes that can be visited in the whole graph gives the number of available nodes relative to the starting position. As all available nodes on a subnet have the same view of the accessibility, the starting position is taken to be a randomly chosen subnet. The update could have been initiated from any node on that subnet. A pseudo code representation of the *Graph Walk* algorithm is given in Appendix D.

5.3.1 Reliability in Topology-1

The configurations are represented by the distribution of the nodes on the subnets and the subnet numbers where the copies reside. For example: (10, 15, 5, 12; 2, 4) represents a configuration where there are 10 nodes on net-1, 15 nodes on net-2, etc. and one of the copies is on net-2 and the other copy is on net-4. The distribution of nodes is important because it might effect the degree of fault tolerance. For example: the above configuration can only tolerate one bridge failure whereas (15, 15, 5, 7; 2, 4) can tolerate two bridge failures if the update is initiated from net-1 or net-2. The data points in the graphs are the mean values of 20 simulation runs. Figure 5.8(a) illustrates the comparison between two file configurations in Topology-1 with reference to the reliability in partition-free networks. In the first configuration, the copies are on different subnets and in the second all copies are on the same subnet. The results show that the reliability is reduced considerably in partitioned networks.

Voting algorithms, MV (three copies) and VWW (two copies and one witness) are simulated in the same topology. As expected, MV behaved in a similar way to VWW. The result obtained from this experiment is rather interesting. RH did not change its behavior as the configuration changed but MV did to a great extent (Figure 5.8(b)).



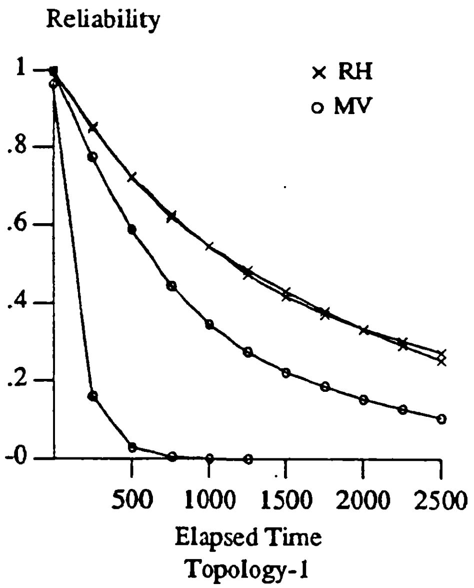


Figure 5.8(a). *Elapsed time vs Reliability (RH)*

Figure 5.8(b). *Elapsed time vs Reliability*

5.3.2 Reliability in Topology-2

Reliability in Topology-2 was analyzed in the same way as the first topology. In this topology, the network becomes more fragmented as the number of bridge failures are increased. The reliability offered by the RH and MV algorithms in the configurations: (5, 7, 7, 10, 10; 3, 4, 5) and (5, 7, 7, 10, 10; 5, 5, 5) are presented in Figure 5.9. Again, in the first configuration copies are on different subnets and in the second all copies are on the same net. The sensitivity of the algorithms is shown to be similar in the two topologies.

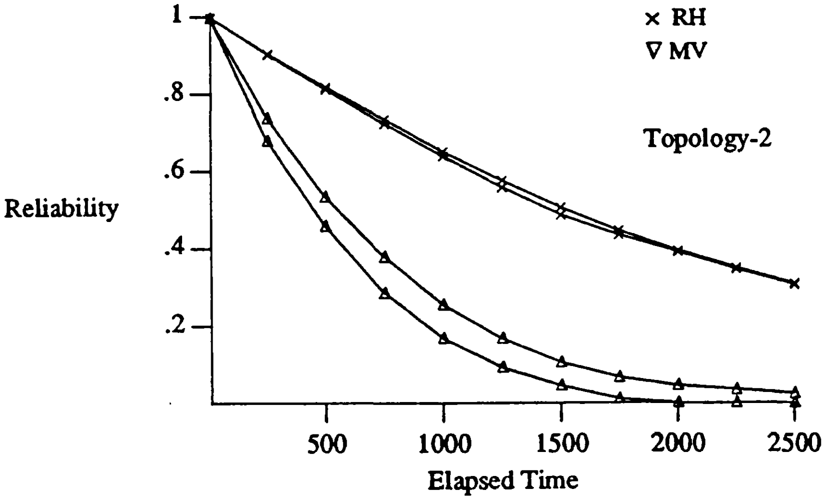


Figure 5.9. *Elapsed time vs Reliability*

5.4 Resilience to Configurational Changes

The above results highlighted the degree of resiliency of the reliable histories and the voting method to the random placement of copies in different topologies. In this section, these results are generalized by two different methods.

The first study is carried out in Topology-2. It attempts to find the likely distribution of the reliabilities at a certain time period. The data points were obtained by simulating the repairs and failures of the system for the reliability at 1000 time units since there is considerable variation for this value. The process was repeated 40 times and the results were sorted to obtain a distribution of the reliabilities provided by both methods. The location of copies was randomly chosen. As before, MV with three copies is compared with RH with two copies, both in a system of 10 processing nodes. The results, shown in Figure 5.10, are rather interesting. MV gave exponentially distributed reliability with mean less than 0.1 where RH gave more normally distributed reliabilities with a sample mean of about 0.23 and standard deviation of 0.028. The relative 95 percent confidence interval of half width is 6% of the sample mean.

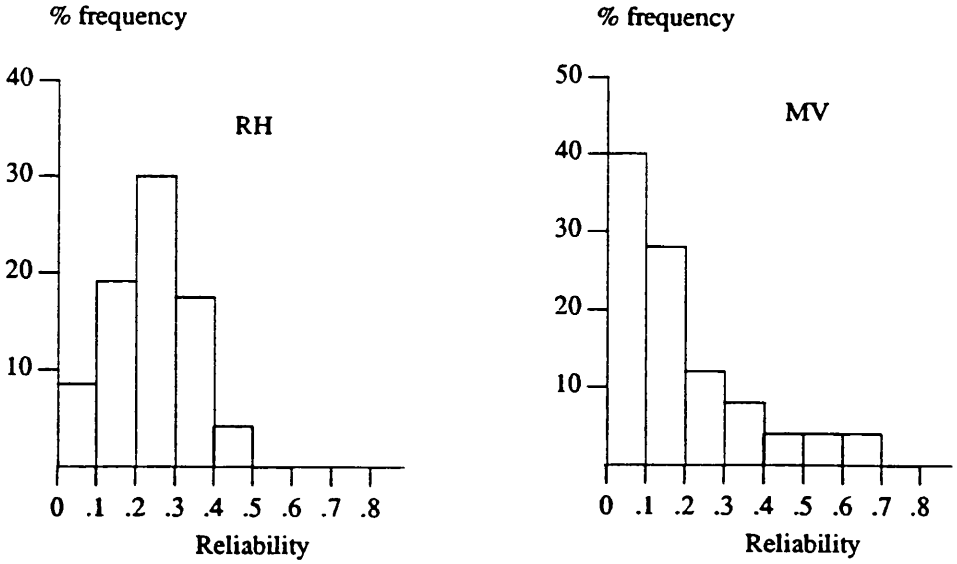


Figure 5.10. *Distribution of reliability at 1000 time units*

The second study is carried out in Topology-1. This time instead of finding the distribution of reliability at a certain time unit, distribution of decay constants for the reliability graphs are found. With reference to Figure 5.11, the data points representing variation in decay constant form a normal distribution for RH in this case too.

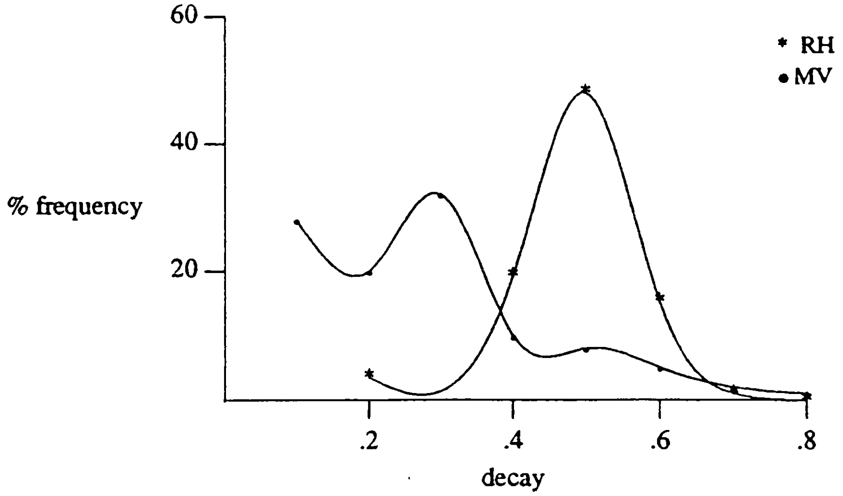


Figure 5.11. *Distibution of decay constant*

5.5 An Analytical Approach to Reliability

Long [58] presented some numerical techniques to predict the fewest number of replicas required to provide the desired level of reliability for partition free systems with estimates of the failure and repair rates under the assumption that when a copy is failed it is regenerated on a spare node. This technique is discussed in Appendix C. Unfortunately, an analytical approach to reliability is far harder when partitioned systems are considered. The following approach models the behavior of the *reliable histories* algorithm under the assumption that the network does not partition and the number of nodes is unlimited. In a network with a large cluster of workstations, the number of nodes is often greater than the desired number of copies. Therefore the number of nodes can be viewed as being effectively unlimited and the history table is always available. The same assumption was used in the combinatorial analysis of availability (see Section 4.1). This assumption is required as the closed-form solutions can only be obtained for the most elementary cases which yield a solution when the history availability is ignored. This section presents the equations arising from n copies managed by the *reliable histories* algorithm (same reliability offered by AC).

The time to notice a node failure and complete a repair is assumed to be exponentially distributed with mean $1/\mu$. Node failures are assumed to be exponentially distributed with mean rate λ. The differential difference equations describing the behavior of a system maintained only by the availability control layer are derived using the STR diagram associated with the AC algorithm for n copies. This diagram is given in Appendix B. There is one additional state 0. The states are labeled to reflect the number of copies that are available. An n copies system is in state 0 if the replicated file has been inaccessible at some point in the past, while for $1<i\leq n$, the system is in state i if the object has been continuously accessible and i copies are currently accessible. No transitions are permitted from state 0, since only the reliability of the system prior to the first failure is of interest.

The set of differential-difference equations arising from n copies is therefore given by

$$\begin{aligned}
\frac{dp_n}{dt} &= \mu p_{n-1}(t) - n\lambda p_n(t), \\
\frac{dp_j}{dt} &= (j+1)\lambda p_{j+1}(t) + (n+1-j)\mu p_{j-1}(t) - (j\lambda + (n-j)\mu)p_j(t), 1<j<n \\
\frac{dp_1}{dt} &= 2\lambda p_2(t) - (\lambda + (n-1)\mu)p_1(t)
\end{aligned}$$

and

$$\frac{dp_0}{dt} = \lambda p_1(t)$$

with initial conditions

$$p_i(t) = \begin{cases} 0 & 0\leq i<n \\ 1 & i=n \end{cases}$$

In this system $p_0(t)$ represents the probability that at time t the system has failed.

Therefore, the reliability of the system is $1 - p_0(t)$.

5.5.1 Improving Reliability with Regeneration

If new copies of a file can be created faster than a system failure can be repaired, better availability can be achieved by regenerating new replicas on other nodes in response to changes in the system configuration. When regeneration is used, reliability trade-offs storage cost. Assessing the costs in terms of network message traffic resulting from regeneration and to estimate the additional storage cost that it incurs are two research areas which require further work.

Copy regeneration can be added to the elementary equations (presented in the previous section) by an exponential distribution with mean γ. In the presence of a total failure, the system is unable to regenerate a copy, and the replicated file will be inaccessible until an up to date copy is repaired. This does not affect the reliability since it is only the behavior of the system prior to a total failure that is of interest.

The equations for a system within which regeneration proceeds in parallel with the repair of a failed copy is quite similar with each μ replaced by $\mu + \gamma$.

5.6 Summary

In this chapter first, the simulation and the parameters used to measure the reliability are justified and the *reliable histories* and voting algorithms are analyzed in partition-free networks and in some topologies where a series of bridge failures divide the system into several self-functioning sets of nodes. In the first case, it was shown that, AC and RH algorithms behave very closely, and the reliability they offer is better than MV and VWW algorithms. In the second case, algorithms were investigated for their behavior in different configurations. The results are generalized using two different approaches (distribution of decay constants and distribution of variation in reliability at a certain time period are presented). The RH algorithm was found to be less sensitive to the network topology and to the location of the copies than voting algorithms.

Secondly, an analytical model for reliability was presented for an elementary case where there are an unlimited number of nodes in the system and the network is partition-free. This model was later extended to include the regeneration technique.

Chapter Six

Performance and Practicality

In this chapter, the cost of network traffic incurred by the replication algorithm will be analyzed in terms of the number of transmissions required. As network congestion is influenced mainly by the number of messages rather than the size of the messages [83], the analysis will focus on the number of high–level transmissions inherited by the *reliable histories* algorithm such as requests for version histories, copy transfers, and the like. The details of the network implementation will determine the actual number of messages generated by a high-level request. While the low–level transmissions may vary with different networks, their number should be proportional to the number of high–level requests. Consequently, this analysis will focus on the number of high level transmissions. This study does not attempt to model systems which guard against concurrent access to files; the consistency scheme would then require further message traffic to implement appropriate commit protocols. Prior to the above analysis, an approach for reducing the communication cost of the history operations is suggested and discussed.

6.1 The Range Algorithm

In many distributed systems, *the number of messages*, not their size is an overriding cost factor. If the underlying communication uses a broadcast link or network level protocol, the network communication cost is a factor of number of replies required for a message, not the number of servers the message is sent to or the size of the parameters [83]. In the *reliable histories* algorithm, the readH operation of the *ReadHistory* function (see Section 3.3.1) is sent to a large number of destinations and all available nodes are required to reply.

When a write request on the file is performed successfully, the new history is written on the available servers. Since the replication control system does not have a recovery procedure, some of the available nodes may hold an out-of-date history. In the original algorithm, all nodes keeping the same version of the history return identical replies to the read. The following algorithm reduces the number of identical replies in a partition-free environment. It cannot preserve consistency if the network partitions. Therefore the *range* algorithm can only be an implementation alternative for the *reliable histories* algorithm in partition-free systems.

Each history server keeps, with every file's history, two sets of nodes called *range* sets. The servers return these *range* sets in their replies. These sets are changed by the local node after history updates and server failures.

a) R_{in}: This set on a particular node is the set of servers that accepted the last history update that this node accepted.

b) R_{out}: This is either empty or contains only the local node number. When a node n recovers from a failure, it sets R_{out} to $\{n\}$ to indicate that it is no longer sure that its histories are up-to-date. An update of a particular history sets R_{out} back to $\{\}$ indicating that the history is known to be up-to-date.

Choosing the Read.Set for the History Table:

Clients of the history servers collect the replies (histories, range sets and timestamps) from the multicast readHR operation and construct a *Read.Set* of nodes that are confident they hold up-to-date histories as follows: If there are n replies with the highest timestamp then the *Read.Set* is:

$$Read.Set = \bigcup_{i=1}^{n} R_{in_i} - \bigcup_{i=1}^{n} R_{out_i}$$

Any history contained in a reply whose R_{in} is a subset of this *Read.Set* may be returned as the result of the *ReadHistory* operation. The following algorithms are more formal descriptions of the *ReadHistory* and *WriteHistory* functions when the *range* algorithm is used. The *ReadHistory* function chooses the *Read.Set* and returns the history table. The *WriteHistory* function updates the histories and records the new *range* sets. Some examples showing the operation of the *range* algorithm are given in Appendix D.

```
$ReadHistory(f)$: $F \rightarrow V \times T$        (algorithm-5)
    let $w \leftarrow$ readHR($M$,$f$)
    if $w = \emptyset$
    return $error$
    let $maxtimestamp \leftarrow max(\{t \mid \exists (R_{in}, R_{out}, t, (h,d)) \in w\})$
    let $S_{in} \leftarrow \{r \mid (r \in R_{in} \wedge (R_{in}, R_{out}, maxtimestamp, (h,d)) \in w\}$
    let $S_{out} \leftarrow \{r \mid (r \in R_{out} \wedge (R_{in}, R_{out}, maxtimestamp, (h,d)) \in w\}$
    let $Read.Set \leftarrow S_{in} - S_{out}$
    return $((h,d) \leftarrow \{(h,d) \mid (R_{in}, R_{out}, t, (h,d)) \in w \wedge R_{in} \subseteq Read.Set)\}$
$WriteHistory(f,h)$: $F \times V \rightarrow \{success, error\}$
    let $R_{in} \leftarrow$ updateH($M$,$f$,$h$)
    let $R_{out} \leftarrow 0$
    return SetRange($M, f, R_{in}, R_{out}$)
```

The following multicast operation is invoked following a history update to the nodes which have accepted the new history. It records the new *range* sets on these nodes.

$SetRange\,(S, f, R_{in}, R_{out})$: $2^M \times F \times 2^M \times 2^M \rightarrow T$

Records the R_{in} and R_{out} on the nodes which have accepted the current changes on the history, i.e. became a member of the new R_{in} set.

The physical table read operation is called readHR in order to distinguish it from readH.

$readHR\,(S, f)$: $2^M \times F \rightarrow 2^{(2^M \times 2^M \times N) \times (V \times T)}$

Invokes a lookup and mapping request on all processing nodes and returns the *range* sets and the last update time as well as the table entries.

Server Rules

Each node follows the following server rules when replying to the read requests on the history.

1. If R_{out} for the file requested is empty, the server listens to the communication link for some randomly chosen time before replying to the request. If, while listening, a node in the file's R_{in} set replies to the same request, the server cancels its own reply. Otherwise, after listening, the server will reply.

2. If R_{out} for the particular file requested is not empty then the server must reply to all read requests for that file's history.

Since the nodes holding the same history return a single reply, more than one reply is received only if there are different *range* sets in the system. In other words, some nodes have failed and recovered between history updates. These nodes might, or might not, hold an obsolete copy of the history table; their status is unknown.

Extra Condition:

If all processing nodes fail and recover between two requests, the resulting *Read.Set*s will be empty since all nodes will be in some R_{out} set. This is an unlikely situation for today's technology in normal circumstances. However, it is possible to add another condition to the *ReadHistory* algorithm to handle this odd case if it occurs.

> If the *Read.Set* constructed by a client is empty, i.e. all responding servers replied with a non-empty R_{out}, then replace the R_{out}s in the replies with empty sets and apply the algorithm again.

6.1.1 Staggering the Replies

Since all the identical nodes consume almost the same amount of server time t_s and all the destinations of a multicast operation receive the request at the same time t_r [43], it is most likely that all nodes will tend to reply to *readHR* request at the time $t_r+t_s+\varepsilon$ where ε is very small. Although in the *range* algorithm all nodes will listen to the link before they reply, as all will tend to reply within a small time interval, it is possible that some nodes would miss the replies from other nodes in the same *range* and reply to the same request unnecessarily. Consequently, the local node will receive multiple replies from the same *range*. The following is a solution to this problem [84].

Each node employs an independent and random listening time taken from an exponential distribution when it receives a request and then listens to the link for that amount of time before attempting to reply. If a node from the same *range* replies during its listening period then it drops the request. Since the listening time is chosen from an exponential distribution, it is highly likely that each time some node will listen for a short time and therefore reply almost immediately. Choosing the mean of this distribution so as to minimize both the number of replies and the response time is a difficult practical problem.

6.1.2 Communication Delay

This section discusses the communication cost of the history operations.

The *ReadHistory* function requires a majority of the nodes to be available. If algorithm-3 (Section 3.3.1) is applied for history access, the maximum delay occurs when all the servers are available and tend to reply to a readH command sent from a local node, given that the local node is available. If the multicast delay for every additional reply is d then the communication delay for the history read operation becomes $(m-1)d$ in this particular case. The minimum delay occurs when exactly the majority of the nodes are available including the local node (if fewer nodes are available the operation fails). Therefore,

$$(\frac{m}{2})d \leq delay_{readH} \leq (m-1)d$$

In the *range* algorithm, the number of replies returned to the *readHR* operation depends on the number of recovered sites which had failed during and/or after the previous history update operation. If the number of failed and recovered sites is α, then there will be α replies from these nodes. Additionally, if the history is updated on the nodes which were always available after the previous update then one reply comes from them as well. Since the update rate is assumed to be much larger than the failure rate, this usually will be the case. The minimum delay for this operation is d and it occurs when none of the nodes have failed and recovered after the previous update. The maximum delay is the same as the previous algorithm and occurs only if all servers fail and recover between two requests. This is an unlikely case especially for large scale systems.

6.1.3 Conclusion

The *range* algorithm allows reads of the history as long as one up-to-date history copy is available, but updates on the history succeed only when a majority of the servers accept it. Therefore, the consistency is preserved. Providing a cheaper history read than history write may improve the performance of the system during the file reads. If the read to write ratio in the system is very high then it may increase the throughput as well. Otherwise it only increases the efficiency of reads.

The *range* algorithm cannot tolerate network partitions. It is possible to increase the tolerance level by adding an extra condition: If the resulting R_{out} is not empty then the algorithm must ensure that the majority of the nodes are available. This condition preserves consistency if the system is partitioned into two self-communicating groups. When the system is partitioned into three or more groups, the algorithm still might fail.

In the previous sections a different approach has been proposed for reading the history. The first algorithm provides a more expensive read but can tolerate all possible partitionings in the system. The *range* algorithm provides a cheaper read operation but can tolerate only the failures that the available copies algorithm can tolerate. This algorithm might be an alternative way for implementing the original available copies algorithm on a less reliable network where configuring unavailable nodes out is more difficult than implementing histories.

6.2 Efficient Implementation of the Scheme

In this section some ideas for efficient implementation of the protocol are discussed. The model requires a read in the table for every file operation and the history is updated after every file write. Many studies have shown that read operations predominate in most general purpose file systems [85, 45, 46]. Therefore reading the history prior to reading the file will increase the communication cost of the algorithm to a great extent as network traffic analysis presented in the next section has shown. If the problem of concurrent write operations is ignored (as it very often is in non-replicated file systems), then it is possible to increase the performance of this algorithm by adding a file *open* operation that caches the file's history locally, writing it back only when a corresponding *close* operation is performed.

The history is read for two purposes: to perform a write on the file, or to read the file. When the history is read for file-read a lock on it is not necessary. If it is read for file-write then a read lock is required in order to preserve consistency. The read-lock is released when the history is written back, i.e. the file is closed.

The characteristics of the communication medium have also a major effect on the performance. Many of the low–level operations required to support this algorithm would benefit from a multicast request-response mechanism. If the underlying communication system uses a broadcast link level protocol, the cost of such a mechanism is a factor of the number of *replies* required from a request, not the number of servers to which the request was sent, nor the size of the request parameters. The actual cost of the algorithm in terms of response time can only be seen in real implementation. As the originality of the work presented in this dissertation has shown with analytical models and the results are verified with a simulation model (chapters three to five), a pilot implementation is planned as future work rather than presented as a part of the dissertation. This is because of the time constraints of the research period.

6.3 Network Traffic Analysis

In this section, the number of transmissions required by the scheme will be analyzed for a multicast environment in which a single transmission may be received by several sites and unicast networks which require transmissions to be addressed to each individual node.

If the history is not cached locally, voting and available copies algorithms incur negligible traffic compared to the reliable histories method. In voting and available copies only the nodes holding a file's copy participate in the operations whereas in the reliable histories algorithm additionally history operations require access to the majority of the nodes in the system. If the file's history is cached during the first request, subsequent accesses incur the same amount of traffic that the available copies algorithm does. The network traffic analysis of the available copies and the voting method for block level replication has been done by Carroll *et al* [44].

The following analysis assumes file-level replication. The number of messages generated by a given operation often depends on the average number of nodes participating in the operation. In voting, this depends on the number of operational nodes and in the available copies algorithm involves the average number of available nodes holding a copy. In the reliable histories algorithm, the first read/write and the last write involves access to the history table. The number of participants depends on the history handling protocol used. In the following analysis it is assumed that algorithm-3 (see Section 3.3.1) is used. Therefore the number of participants is the average number of available nodes not necessarily holding the history or the file copy. The *range* algorithm would provoke less messages for the read, but would require extra traffic for writing the *range* sets after the update. Since the history is cached when the file is open, reads require only one node to participate whereas in writes the number of participants depends on the available nodes holding a copy of the file. The average number of nodes responding to a history request from some local node (given that local node is available) can be derived using the state probabilities.

$$R = \frac{\sum_{i=1}^{m} i p_i}{\sum_{i=1}^{m} p_i}$$

where p_i denotes the probability of the system being in the state S_i representing the availability of i processing nodes.

The value p_i is dependent on the equilibrium state conditions. Carroll [44] has shown that, in an x-node network, for voting and available copies the number of participants is given by

$$R_x = x(1-\rho)+O(\rho^2)$$

with $O(\rho^2)$ negligible for values of ρ typical for computer systems. Therefore, when considering RH, for the history operations $x=m$ and for the file operations $x=n$.

In a multicast environment the algorithm broadcasts one message when a read or write is performed. The local node receives a single response to reads and multiple responses to writes from the nodes which accepted the new version. In this case, the number of responses is at most n which is also negligible (typically two) compared to the traffic generated by the history access. The scheme provokes larger traffic for the history operations. The history read/write operations broadcast one message to all processing nodes in the system and receive responses from all available nodes. Therefore it results in

$$1+m(1-\rho)+O(\rho^2)$$

messages. The algorithm incurs no traffic upon recovery without degrading user access or availability. The number of network operations required (for various m values where n=2) in a multicast environment are given in Figure 6.1(a). A typical value of $\rho = 0.05$ is used. The dependent axis reflects the number of high level transmissions generated by reads and writes.

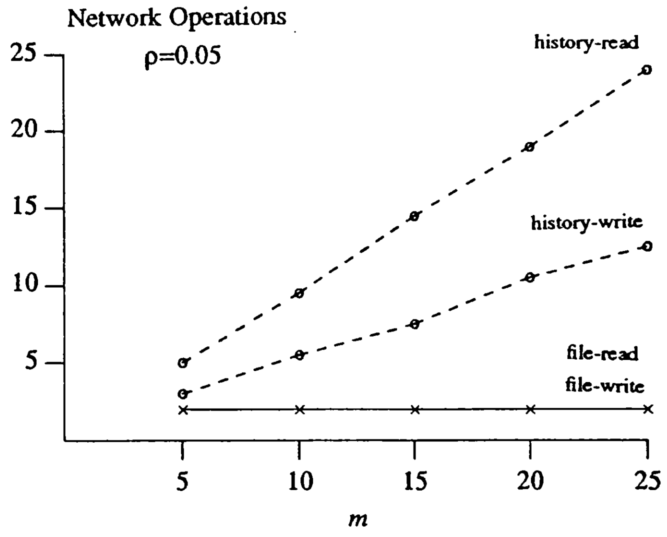


Figure 6.1(a). *Multicast environment*

In the absence of a multicast network, separate messages must be individually addressed to each destination node. In this case the RH algorithm accounts for larger amount of traffic for the history operations. These operations result in

$$2m(1-\rho)+2O(\rho^2)$$

messages this time. The scheme employing different number of nodes in a unicast environment is given in Figure 6.1(b). These graphs show the number of requests and average number of expected replies.

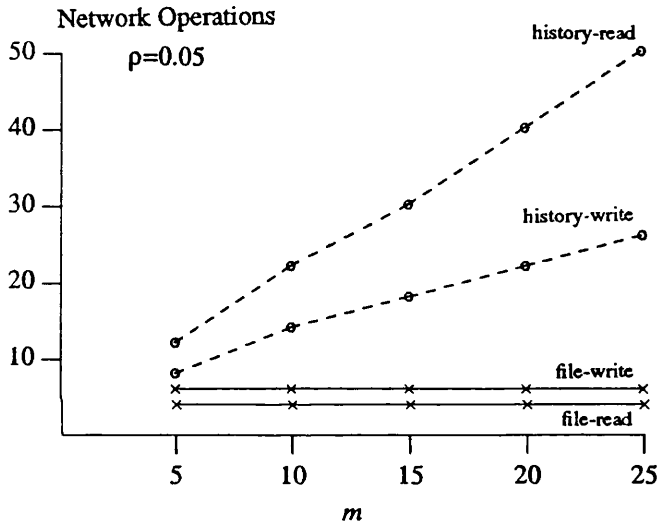


Figure 6.1(b). *Unicast environment*

Table 6.1 tabulates the maximum number of multicast operations inherent in the *reliable histories* algorithm as a function of number of replies expected from the destinations. If the file's history is not cached in the local memory then each read operation would require the total number of interactions for file-open + read and each write would require write + file-close. The figures in the table are found under the assumption that when the file is open, the history is cached and when it is closed the new history is written back. The second column is the number of read/write accesses to the file copies. It gives the number of servers that the message is sent to and the expected number of replies. The third column is the number of read/write accesses to the history copies. The

n is the number of file copies which is very small compared to the number of history copies ($\leq \lfloor m/2 \rfloor + 1$) in the system.

Request	*r/w (file)*	*r/w (history)*
File-Open	-	$1 \rightarrow (m-1)$
Read	$n \rightarrow 1$	-
Write	$n \rightarrow n$	-
File-Close	-	$1 \rightarrow (m-1)$

Table 6.1. — *Network interactions required by RH*

If the *range* algorithm is used then only the number of replies returned to the *file–open* request would change to (α+1) in the table.

6.4 Summary

This chapter has focused on the number of high–level transmissions inherited by the *reliable histories* algorithm such as requests for version histories, copy transfers, and the like. The study does not attempt to model systems which guard against concurrent access to files; each of the consistency schemes would then require further message traffic to implement appropriate commit protocols. Prior to this analysis, an approach for reducing the communication cost of the history operations was described. This approach allows an efficient implementation of the protocol for read operations.

Chapter Seven

Conclusion and Further Work

In this conclusion, a general summary of the dissertation is given in Section 7.1 pointing out the originalities in the algorithm proposed and original findings in the effectiveness analysis of the consistency schemes, based on voting and available copies algorithms. This analysis considers the availability (steady-state and continuous) of replicated data in partitioned networks as well as in partition-free systems. Section 7.2 summarizes the findings and Section 7.3 discusses the areas for further research.

7.1 General Summary

The potential for increased reliability through replication is often given as one of the benefits of distributed systems. This dissertation has analyzed the consistency problem of small degree replication in large-scale distributed systems. The thesis concentrated on the first and second of the three central problems outlined below which are required to be solved before the benefits of replication can be realized in a wider range of applications.

1. When the size of the file to be replicated is large, several kilobytes or more, a replication control algorithm might be required to provide high reliability with only two

file replicas because of the storage cost of extra copies. It has been shown that the *reliable histories* algorithm provides optimal availability with two copies whereas voting methods have no practical use.

2. The reliability provided by the consistency scheme should not be adversely affected by changes to the network topology and therefore to the failure modes of the network. As networks grow and evolve, subnets can become bridged together and machines moved from subnet to subnet; more often than not as required by geographical constraints. As a result, it is reasonable to assume that network partitioning is a relatively likely event which might affect the reliability performance of the replicated data and replication is most likely to operate under random copy placement. The partitioning problem is always considered from the correctness point of view rather than its effect on the performance. As the results have shown, these failures affect different algorithms to different degrees. Although it is a complicated issue, this area has great potential for future work as discussed in the next section.

3. The system should provide a flexible reconfiguration mechanism to alter the reliability of files as users' requirements change. The file might require different degrees of availability for different periods of its lifetime. This availability can be achieved either by changing the location of the copies or creating temporary copies to reach the desired availability level. Either solution requires interruption of the system administration unless the algorithm itself provides reconfiguration facility. The replication protocol should therefore be flexible enough to add this dynamicity as an extension to its functions.

These problems are relatively easy to analyze and open to flexible solutions when the level of replication is whole files rather than blocks. Various algorithms have been proposed and implemented as a consistency scheme for replication. These schemes have been surveyed many times in the literature in terms of correctness, degree of fault tolerance provided (types of faults that the algorithms can tolerate), requirements from the protocols running below; interconnection with the concurrency control protocol etc. Although Long [58] has compared reliability of three regenerative algorithms: dynamic voting, majority voting and available copies, and Paris [44] has analyzed the availabilities provided by majority voting and a variation of available copies for block-level replication, the algorithms have not been measured according to their approach to the central problems outlined above. Distinguishing the effectiveness and the features required for efficiency therefore is an original approach for comparison of the basic principles considering that no comparative data for file level replication (either availability or reliability) is available in the literature.

The introductory chapter identifies general measures applicable to all replication control schemes. These measures are grouped into two as measures of effectiveness and measures of efficiency. Effectiveness measures are the features of the algorithm which are common to all application areas. These measures are basically grouped into two: steady-state availability and reliability (availability over a given period of time). Efficiency measures are more complicated to define as some of them are extensions to the effectiveness measures. The first three of the following are examples of such dependent measures whereas four to six are measures related to the cost of the algorithm (communication and storage):

i. The effect of topological changes on the effectiveness of the algorithms and the degree of resilience to random copy placement.

ii. The degree of reduction in the availability and reliability of the replicated data in the case of partitioning.

iii. The flexibility of reconfiguration and the degree of dynamicity in moving the objects of the replication system; files, replication histories etc..

iv. The lower bound for the number of copies required in order to give a considerable improvement over single copy in terms of the effectiveness measures given above.

v. Number of network interactions required during each read and write; reading version vectors, reading or writing the up-to-date copies and copying the up-to-date version to obsolete copies.

vi. Requirements of the failure model from the underlying communication medium; distinguishing network failures from node crashes, recognizing failures instantaneously and whether all nodes must have the same view about the state of the network (the nodes which are up and which are down).

Chapter Two contains a survey of the existing schemes and a critical analysis of the basic principles, comparing the advantages and disadvantages when the replication degree is small. This survey has concluded that the algorithms grouped together as *read any/write all* provide optimal availability for two copies, but the requirements of the failure model from the underlying communication medium in order to survive when the network has split into more than one functioning group cannot be satisfied in general purpose computing environments. The algorithms under the second principle: *read some/write some*, can tolerate network partitions but these algorithms require a minimum of three copies in order to give improved availability over a single copy.

Considering the fact that three copies has relatively little advantage over two copies in terms of the degree of the reliability and the availability provided, a practicable two copy consistency control algorithm, *reliable histories*, is described in Chapter Three. This algorithm offers high availability over a period of time as well as instantaneously even with two copies and can function properly during network partitions. The benefit of this functionality together with high availability is not provided by any of the other algorithms that were studied. The *reliable histories* algorithm considers the storage cost of replication. It replicates a small amount of data (replication history) concerning the location and status of file's small number of copies, a larger number of times using a regenerative *read some/write some* principle and uses this highly reliable vital information to determine the update strategy. The originality of the *reliable histories* approach is that it is a hybrid algorithm which maintains history tables and file copies with independent schemes under different consistency constraints. This functionality provides optimal availability for two copies and enables the algorithm to survive under partitions. The file's history table is very small (8 byte is often enough) therefore it does not incur as large storage cost as having three copies of the file.

The layering of the replication control system as the *availability control layer* and the *history table control layer* increases the flexibility of the algorithm for reconfiguration and eases the use of regeneration (for file copies) within the algorithm. The regeneration approach has an additional advantage when used in the *reliable histories* algorithm. In fact, in the original form of regeneration as proposed by Pu [73], each time a node recovers, the algorithm must check to see if the maximum number of copies exist or if the file was updated during its absence. If so, the recovered copy is deleted, otherwise, it can be used. This requires a double check during recovery. This recovery procedure is avoided in the reliable histories method. If the copy is regenerated, only the history table is updated. The system automatically brings itself to equilibrium during the next update, therefore a recovery operation is not required when a node is repaired. This flexible reconfiguration also enables the number of copies to change dynamically to reach a desired level of reliability for a certain period of time without any additional administration. This area will be discussed as further work in the next section.

There are two objectives of the steady-state availability analysis in Chapter Four. The first objective is to provide information for the minimum requirements of the proposed algorithm from the distributed environment (number of nodes in the system) in order to gain advantage over the other methods. The second, to measure the availability provided by the methods in partitioned systems. Two analyses were developed corresponding to the first objective and one of these analysis was extended to cover the second. The first statistical combinatorial analysis compares the availability provided by various algorithms under a simple failure model using independent fail probabilities of individual nodes. The second analysis models the behavior of the algorithm in partition-free systems using Markov models. This model uses failure/repair model in which the mean time to failures and the mean time to complete repairs are randomly distributed according to a Poisson law. This analysis has shown that with two copies, the advantage in availability is acquired by the proposed algorithm in systems with ≥ 5 nodes. This analysis gives the lower bound of availability and shows that the proposed algorithm provides better availability with two copies than any form of voting with three copies when the boundary requirements of the system are satisfied (size is larger than five). The third analysis extends the combinatorial approach to include a simple partitioning case. Though simulation always exists (with some restrictions explained later) to quantify the decrease in availability when the network partitions no other method is currently available to give the required information through theoretical models. This combinatorial analysis analyzed the change of behavior of the algorithms under various configurations. The first outcome which was obtained through this analysis (which later resulted in an interesting conclusion after reliability analysis) is that the proposed approach behaved very steadily in the sample system whereas the configurational changes caused dramatic differences in the availability provided by voting methods. The validity of the analysis is supported by results obtained through a software simulation. The simulation model is later used to generalize the results obtained above by analyzing the behavior of the algorithms in some specific topologies where bridge failures can break the system into several self communicating partitioned groups. The availability results obtained in several network topologies supported the outcome of the analytic results obtained for the simple partitioning case.

Chapter Five presents the results of an original reliability analysis through simulation in partition free and partitioned systems. The results have shown that although the methods provide relatively close availabilities, reliabilities vary greatly. This analysis was developed to present the decrease in reliability during network partitions when copy placement is random. Although partitioning has been shown to reduce the reliability to some extent, the new *reliable histories* algorithm behaved very similarly to available copies in terms of reliability performance. In most of the configurations, the new algorithm with two copies provided better availability and reliability than voting with three.

An important originality of this research is its approach to looking at the dependency of the algorithm's performance on the placement of the copies and on the topology of the data network. The analysis has led to a very interesting conclusion. It has shown that although it performs worse in some configurations, the reliable histories approach is more resilient to changes of the copy placement and performs better on average. Resilience to configurational changes is an important issue in the design of a general purpose replication system where the configuration cannot be planned to maximize file reliability, say, because it is dictated by physical considerations as explained in Chapter Five. This chapter concluded with an analytic model incorporating regeneration of file copies in order to increase the reliability. The *regenerative–reliable histories* model has derived in line with Long's analysis of regeneration with finite and infinite numbers of spares.

The efficiency of the proposed approach in terms of the high-level transmissions it inherits and the practicality of the algorithm are studied in Chapter Six. The low-level transmissions may vary with different networks but their number is proportional to the number of high-level transmissions. This analysis was based on two different proposals for history management. The first proposal was discussed in Chapter Three as a part of the main algorithm. Another algorithm (*range*) was proposed later which reduces the communication cost of the history operations; based on the idea that in many distributed systems, the number of messages not their size is an overriding cost factor. The *range* algorithm reduces the number of replies returned to a history read request but requires synchronized clocks. An approach for logically synchronizing the clocks in multi-responses has also been proposed. The characteristics of the communication medium for an efficient implementation of the scheme were discussed. Many of the low-level operations required to support the algorithm would benefit from a multicast environment. The final section of Chapter Six analyzes network traffic for both unicast and multicast networks. This analysis has shown that the *range* algorithm reduces the traffic to a great extent when implemented in a multicast environment.

7.2 Summary of Findings

Going from single copy to two copies is more advantageous than going from two to three or from three to four copies etc. in terms of the improvement in the availability provided and the cost of storage. The performance and behaviour of consistency schemes have different patterns for small degree replication than large degree replication; especially for two copies. Dynamic strategies and voting algorithms have no practical use with two copies.

The degree of fault tolerance and the degree of availability provided by consistency schemes trade off each other. Voting strategies can tolerate network partitions but available copies algorithms cannot although they provide optimal availability. A hybrid approach combines the benefits of these two group of algorithms by replicating the file's history (location and version numbers) with a variation of voting and applying an update strategy with a variation of available copies algorithm. This approach also allows for dynamic reconfiguration to alter the reliability as the user's requirement is changed.

Consistency schemes exhibit different behaviour as the network configuration is changed. Resilience to changes is an important property of the algorithm but voting algorithms have been found to be less resilient than the hybrid algorithm.

Whereas various schemes offer similar availabilities, their reliabilities differ greatly. With single copy, a file's reliability is the same as the system's whereas with only two copies it is possible to increase the file's reliability to a great extent. For example: a file may be eight times more reliable than the system when ρ= 0.05 for a period of 250 time units and four times for a period of 1000 time units. In a system where ρ=0.1, these figures change to seven times for Δ t=250 and to three times for Δ t=1000.

7.3 Further Work

The next major advance in research which could help adoption of the replication techniques in general purpose applications would be a characterization of users' involvement and the design of user interfaces for replicated file systems. Replication techniques are used in many areas but its control and use is usually done by system administrators. On the required degree of user control and its effect on the performance, many areas can be further studied:

- If the user is given control of copy placement, what are the administrative requirements during the configuration of the system and during the file operations?
- What is the measurement matrix of reliability for a given time period in order to calculate dynamically when the places of the copies are given as parameters? What should be the involvement and responsibilities of the user and system administrator in dynamic reconfiguration?

Regeneration is an active research area. The proposed algorithm employs a regenerative technique for history manipulation. As the histories are very small, how to reclaim the storage of out-of-date copies is not considered in the thesis as a major problem. In fact, once a failed node is repaired, the regenerated replicas become superfluous and the additional storage can be reclaimed. This can become a problem if the file copies are regenerated as well. The question of which replicas should be reclaimed is an area for future research. Reliability analysis through regeneration still requires further work in terms of assessing the additional storage cost it incurs and estimating the network traffic resulting from regeneration. The performance of protocols that would only regenerate a fraction of the initial number of copies should be evaluated to determine their effect on the reliability provided by the protocol. How availability is changed by regenerating file copies also requires further analytical approaches. A variety of theoretical options should be considered before implementation begins.

Another extension that should also be the subject of statistical analysis is the partitioning case. An analytical model for the behavior of a replication control algorithm involving network partitions would certainly help to generalize and verify further the results presented in the dissertation. Unfortunately, this is a difficult task for the proposed algorithm because of the number of nodes required before a significant improvement over voting methods can be obtained. Partitioning is usually considered as a problem concerning the correctness of the scheme. Its analysis including the effect on performance is required in many respects as it cannot be ignored with the pace of technological advancement and application areas of computer networks.

The effects of network configuration and file placement on reliability require more further work. One approach would be to generate and simulate ``random'' configurations. While this is relatively simple for copy placement, the next advancement in this area would be the clarification of the notion of a randomly chosen topology. Perhaps by studying networks in the field, some criteria for probable topologies and node distributions can be determined so that suitable configurations can be randomly chosen and simulated. Such a study would make it possible to derive rules that would allow the reliability of a file to be calculated when the topology and location of copies are given as parameters.

A pilot implementation would certainly help to test the practicality of the algorithm, verify the estimated communication overhead involved and verify the availability and reliability analysis. Also, further research is needed to investigate the performance and requirements of the scheme in applications where concurrent updates are possible and must be serialized.

The first 81 entries are referred to directly in the text, the remaining works are representative of background material that has been studied.

References

1. L. Svobodova, "File Servers for Network-Based Distributed Systems," *ACM Computing Surveys*, vol. 16(4), pp. 353-398, December 1984.
2. M Hsu and A Chan, "Partitioned Two Phase Locking," *ACM Transactions on Database Systems*, vol. 11(4), pp. 431-446, December 1986.
3. J N Gray, "Granularity of Locks in a Shared Distributed Database," *Proceeding of Int. Conf. VLDB*, pp. 428-451, September 1975.
4. P A Bernstein and N Goodmann, "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases," *ACM Transactions on Database Systems*, vol. 9(4), pp. 596-615, December 1984.
5. R H Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Transactions on Database Systems*, vol. 4(2), pp. 180-209, June 1978.
6. D Gifford, "Weighted Voting For Replicated Data," *Proceeding of the 7th ACM Symp on Operating System Principles*, pp. 150-162, December 1979.
7. J J Bloch, D Daniels, and A Spector, "A Weighted Voting Algorithm for Replicated Directories," *Journal of ACM*, vol. 34(4), pp. 859-909, October 1987.
8. D Daniels and A Spector, "An Algorithm for Replicated Directories," *ACM Operating Systems Review*, pp. 24-43, 1983.
9. A El-Abbadi, D Skeen, and F Cristian, "An Efficient, Fault Tolerant Protocol for Replicated Data Management," *Proceedings of the 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pp. 215-229, New York, March 1985.
10. M Herlihy, "A Quorum-Consensus Replication Method for Abstract Data Types," *ACM Transactions on Computer Systems*, vol. 4(1), pp. 32-53, February 1986.
11. B S Bacarisse and S Bek-Baydere, "A Low Cost Replication Algorithm," *Proceedings of the IEEE COMPCON Spring'89,San Fransisco*, February 1989.
12. S Bek-Baydere and B S Bacarisse, "Reliability of Replicated Files in Partitioned Networks," *Proceedings of the 1st IEEE Workshop on Management of Replicated Data*, November 1990.
13. A S Tanenbaum and R V Renesse, "Distributed Operating Systems," *ACM Computing Surveys*, vol. 17(4), pp. 419-470, December 1985.
14. A S Tanenbaum and S J Mullender, "An Overview of the AMOEBA Distributed Operating System," *ACM Operating Systems Review*, pp. 51-64, 1981.
15. D R Brownbridge, L F Marshall, and B Randell, "The NewCastle Connection," *Software-Practice and Experience*, vol. 12, pp. 1147-1162, July 1982.

16. A Wambecq, ''NETIX: A Network Operating System Based on UNIX Software,'' *Proceedings of the NFWO-ENRS Contact Group.*

17. Sun MicroSystem Inc, *Network File System*, February 1986.

18. E J Berglund, ''An Introduction to The V System,'' *IEEE Micro*, pp. 35-52, August 1986.

19. M R Brown, K N Kolling, and E A Taft, ''The Alpine File System,'' *ACM Transactions on Computer Systems*, vol. 3(4), pp. 261-293, November 1985.

20. E C Cooper, ''Circus: A Replicated Procedure Call Facility,'' Proceeding of the 4th Symp on Reliability in Distributed Software and Database Systems, October 1984.

21. P Dasgupta, R J LeBlanc Jr, and W F Appelbe, ''The Clouds Distributed Operating System,'' *IEEE Proceeding of the 8th Int Conf on Distributed Computing Systems*, pp. 2-8, June 1988.

22. M Fridrich and W Older, ''Helix: The Arcitecture of the XMS Distributed File Sytems,'' *IEEE Software*, vol. 2(3), pp. 21-29, May 1985.

23. M Stonebraker, ''Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES,'' *IEEE Trans. on Software Engineering*, vol. SE-5(3), pp. 188-194, May 1979.

24. G Popek, B Walker, J Chow, D Edwards, C Kline, G Rudisin, and G Thiel, ''LOCUS: A network Transparent High Reliability Distributed System,'' *Proceeding of the 8th ACM Symp on Operating System Principles*, pp. 169-177, December 14-16,1981.

25. H Sturgis, J Mitchell, and J Israel, ''Issues in the Design of a Distributed File System,'' *ACM Operating Systems Review*, vol. 14(3), pp. 55-69, July 1980.

26. P A Bernstein and N Goodman, ''Concurrency Control in Distributed Database Systems,'' *ACM Computing Surveys*, vol. 13(2), pp. 185-221, June 1981.

27. W H Kohler, ''A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems,'' *ACM Computing Surveys*, vol. 13(2), pp. 149-183, June 1981.

28. D P Reed, ''Implementing Atomic Actions on Decentralized Data,'' *ACM Transactions on Computer Systems*, vol. 1(1), pp. 3-23, February 1983.

29. J N Gray, P Mcjones, M W Blasgen, R A Lorie, T G Price, G F Putzulu, and I L Traiger, ''The Recovery Manager of the System R Database Manager,'' *ACM Computing Survey*, vol. 13(2), pp. 223-242, June 1981.

30. M Hammer and D Shipman, ''Reliability Mechanisms for SDD-1: A System for istributed Databases,'' *ACM Transcations on Database Systems*, vol. 5(4), pp. 431-466, December 1980.

31. C H Papadimitriou, ''The Serializability of Concurrent Database Updates,'' *Journal of ACM*, vol. 26(4), pp. 631-653.

32. K P Eswaran, J N Gray, R A Lorie, and I L Traiger, ''The Notions of Consistency and Predicate Locks in a Database System,'' *Communications of ACM*, vol. 19(11), pp. 624-633, November 1976.

33. P A Bernstein and N Goodman, ''Timestamp Based Algorithms for Concurrency Control in Distributed Database Systems,'' *Proceedings of the 6th International Conference on Very Large Databases*, October 1980.

34. H T Kung and J T Robinson, ''On Optimistic Methods for Concurrency Control,'' *ACM Transcations on Database Systems*, vol. 6(2), pp. 213-226, June 1981.

35. M Herlihy, ''Optimistic Concurrency Control for Abstract Data Types,'' *ACM Operating Systems Review*, pp. 33-44, 1985.

36. K H Bennett, ''Mechanisms For Distributed Control,'' in *Distributed Comp.*, Academic Press, 1984.

37. E Gelenbe and K Sevcik, ''Analysis of Update Synchronization for Multiple Copy Databases,'' *Proceedings of the 3rd Berkeley Workshop Distributed Databases and Computer Networks*, August 1978.

38. L Lamport, ''Time, Clocks, and the Ordering of Events in a Distributed System,'' *Communications of ACM*, vol. 21(7), pp. 558-565, July 1978.

39. S Wilbur and B Bacarisse, ''Building Distributed Systems with Remote Procedure Call,'' *IEE Software Engineering Journal*, vol. 2(5), pp. 148-159, September 1987.

40. A D Birrell and B J Nelson, ''Implementing Remote Procedure Calls,'' *ACM Trans.Comp.Sys*, vol. 2(1), pp. 39-59, February 1984.

41. A Z Spector, D Daniels, D Duchamp, J L Eppinger, and R Paussch, ''Distributed Transactions for Reliable Systems,'' Technical Report CMU, 1985.

42. A Z Spector, ''Communication Support in Operating Systems for Distributed Transactions,'' Technical Report CMU, August 1986.

43. J Crowcroft and K Paliwoda, ''A Multicast Transport Protocol,'' Research Note, University College London, March 1988.

44. J L Carroll, D D E Long, and J F Paris, ''Block-Level Consistency of Replicated Files,'' *IEEE 7th Int. Con. on Distributed Computing*, pp. 146-153, September 1987.

45. J Ousterhout and H DaCosta, ''A Trace-Driven Analysis of the Unix 4.2 BSD File System,'' *ACM Operating Systems Review*, vol. 19(5), pp. 15-24, December 1985.

46. S Bek, ''Actual and Potential Performance of NFS,'' Indra Note 2193, UCL,Dept of Computer Science, September 1987.

47. J H Howard, M L Kazar, S G Menees, D A Nichols, M Satyanarayanan, R N Sidebotham, and M J West, ''Scale and Performance in a Distributed File System,'' *ACM Transactions on Computer Systems*, vol. 6(1), pp. 51-81, February 1988.

48. L Lamport, R Shostak, and M Pease, ''The Byzantine Generals Problem,'' *ACM Transactions on Computer Systems*, vol. 4(3), pp. 382-401, July 1982.

49. R D Schlichting and F B Schneider, ''Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems,'' *ACM Transactions on Computing Systems*, vol. 1(3) P 222-238, August 1983.

50. P A Bernstein and N Goodman, ''The Failure and Recovery Problem for Replicated Databases,'' *Proceedings, 2nd Annual Symphosium on Principles of Distributed Computing*, pp. 114-122, August 1983.

51. J S M Verhofstad, "Recovery Techniques for Databases," *ACM Computing Surveys*, vol. 10(2), pp. 167-195, June 1978.

52. B G Lindsay, "Single and Multi-Site Recovery Facilities," in *Distributed Databases-An Advanced Course*, ed. F Poole, Cambridge University Press, 1980.

53. B Bhargava and Z Ruan, "Site Recovery in Replicated Distributed Databases," *IEEE Proceedings ot 6th International Conference on Distributed Computing Systems*, pp. 621-627, 1986.

54. M L Powell and D L Presotto, "A Reliable Broadcast Communication Mechanism," *Operating System Review(ACM)*, vol. 17(5), pp. 100-109, 1983.

55. A M Johnson,Jr and M Malek, "Survey of Software Tools for Evaluating Reliability, Availability and Serviceability," *ACM Computing Surveys*, vol. 20(1), pp. 227-269, December 1988.

56. R Van Renesse and A S Tanenbaum, "Voting with Ghosts," *Proceeding of the 8th Int. Conf on Distributed Computing Systems*, pp. 456-461, June 1989.

57. R E Barlow and K D Heidtmann, "Computing k-out-of-N Reliability," *IEEE Trans. on Reliability*, vol. R-33 (4), pp. 322-323, October 1984.

58. D D E Long, J L Carroll, and K Stewart, "The Reliability of Regeneration-Based Replica Control Protocols," University of California, Computer Research Lab, Technical Report UCSC-CRL-88-18, October 1988.

59. D L Eager and K C Sevcik, "Achieving Robustness in Distributed Database Systems," *ACM Transactions on Database Systems*, vol. 8(3), pp. 354-381, September 1983.

60. J F Paris, "Voting with a Variable Number of Copies," *Proceedings of the 16th FDCS*, pp. 50-55, 1986.

61. P G Selinger, "Replicated Data," in *Distributed Databases-an advanced course*, ed. F Poole, Cambridge University Press, 1980.

62. P A Alsberg, "A Principle for Resilient Sharing of Distributed Resources," *Proc. 2nd Int Conf of Software Engineering*, pp. 562-570, October 1976.

63. J F Paris, "Voting with Witnesses: A Consistency Scheme for Replicated Files," *Proceeding of the 6th Int. Conf. on Distributed Computing Systems*, pp. 606-612, 1986.

64. S Jajodia and D Mutchler, "Dynamic Voting," *ACM SIGMOD International Conference on Data Management*, pp. 227-238, 1987.

65. M Herlihy, "Dynamic Quorum Adjustment for Partitioned Data," *ACM Transactions on Database Systems*, vol. 12(2), pp. 170-194, June 1987.

66. D Barbara, H Garcia-Molina, and A Spauster, "Policies for Dynamic Vote Reassignment," *IEEE Proceedings ot 6th International Conference on Distributed Computing Systems*, pp. 37-44, 1986.

67. D Barbara, H Garcia-Molina, and A Spauster, "Increasing Availability Under Mutual Exclusion Constraints with Dynamic Vote Reassignment," *ACM Transactions on Computer Systems*, vol. 7(4), pp. 394-426, November 1989.

68. T Mann, A Hisgen, and G Swart, ''An Algorithm for Data Replication,'' Technical Report:46, Digital Systems Research Center, June 1,1989.

69. A Huseyin, G Pavlou, and P T Kirstein, ''A Distributed Database Study,'' Technical Report 143, UCL, Computer Science, March 1988.

70. H Garcia-Molina, ''Elections in a Distributed Computing System,'' *IEEE Transactions on Computers*, vol. 31(1), pp. 48-59, January 1982.

71. H Garcia-Molina and D Barbara, ''How to assign votes in a Distributed System,'' *Journal of ACM*, vol. 32(4), pp. 841-860, October 1985.

72. S B Davidson, H Garcia-Molina, and D Skeen, ''Consistency in Partitioned Networks,'' *ACM Computing Surveys*, vol. 17(3), pp. 341-370, September 1985.

73. C Pu, J D Noe, and Proudfoot, ''Regeneration of Replicate Objects: a Technique and Its Eden Implementation,'' *IEEE Proceeding of the 2nd Int. Con. on Data Engineering*, February 1986.

74. L Donatello and B R Iyer, ''Analysis of a Composite Performance Reliability Measure for Fault-Tolerant Systems,'' *Journal of the ACM*, vol. 34(1), pp. 179-199, January 1987.

75. D D E Long and J F Paris, ''A Realistic Evaluation of Optimistic Dynamic Voting,'' *IEEE Proceedings of the 6th Symposium on Reliable Distributed Systems*, pp. 129-137, October 1988.

76. J D Noe and A Andreassian, ''Effectiveness of Replication in Distributed Computer Networks,'' *IEEE Proceeding of the 7th Int. Conf. on Distributed Computing*, pp. 508-513, September 1987.

77. R D Schlichting, G R Andrews, and T D M Purdin, ''Mechanisms to Enhance File Availability in Distributed Systems,'' *Proceeding of the 6th Int. Conf on Distributed Computing Systems*, pp. 44-49, June 1986.

78. S Ross, in *Introduction to Probability Models*, 1970.

79. Y C Tay, ''The Reliability of (k,n)-Resilient Distributed Systems,'' *IEEE 3rd Conf on Data Engineering*, pp. 119-122, 1984.

80. M Sahinoglu, ''Use of Marchkov Modeling in Power System Reliability Studies,'' Master of Science Dissertation, UMIST,Manchester, October 1975.

81. M Sahinoglu, ''Use of Marchkov Modeling and Statistical Data Analysis in Spare Plant Assesment-Its Economic Evaluation,'' *Proceedings of the 7th Annual Reliability Conference on Reliability for Electric Power Industry,Wisconsin,USA*, pp. 269-278, April 1980.

82. M Sahinoglu, ''The Evaluation of Reliability Indices for the Off-Site Electric Power System at the Akkuyu Nuclear Power Plant for the Asssesment and Planning of On-Site Plant Reliability,'' Final Report,Ankara,Turkey, October 1987.

83. T A Joseph and K P Birman, ''Low Cost Management of Replicated Data in Fault Tolerant Distributed Systems,'' *ACM Transactions on Computer Systems*, vol. 4(1), pp. 54-70, February 1986.

84. S Bek-Baydere, ''Synchronising Multi-Responses for Version Vectors,'' Research Note, UCL, Dept. of Computer Science, December 1989.

85. JG Mitchell and J Dijon, "A Comparison of Two Network File Servers," *Comm. ACM*, vol. 25(4), pp. 233-246, April 1982.

86. W E Boyce, "Exponential Models," in *Case Studies in Mathematical Modelling*, Pitman Publishing Inc, 1981.

87. J W Cohen, in *The Single Server Queue*, North-Holland, 1969.

88. I N Herstein, in *Topics in Algebra*, p. 209, Blaidell Publishing Company, 1964.

89. G H Golub and C F Van Loan, in *Matrix Computations*, Baltimore:The Johns Hopkins University Press, 1983.

90. C B Moler and C F Van Loan, "Nineteen Dubious Ways to Compute the Exponential of a Matrix," *SIAM Review 20*, pp. 801-836, 1978.

91. S Bek-Baydere and B S Bacarisse, "Reliable Histories for Replicated Files," *Proceedings of the ISCIS-IV, Cesme,Turkey*, October 1989.

92. O Babaoglu, "On the Reliability of Consensus-Based Fault Tolerant Distributed Computing Systems," *ACM Transactions on Computer Systems*, vol. 5(3), pp. 394-416, November 1987.

93. J Donnelly, "Components of a Network Operating System," *Computer Networks*, vol. 3, pp. 389-399, 1979.

94. A J Frank, L D Wittie, and A J Bernstein, "Multicast Communication on Network Computers," *IEEE Software*, pp. 49-61, May 1985.

95. A J Frank, L D Wittie, and A J Bernstein, "Maintaining Weakly-Consistent Replicated Data on Dynamic Groups of Computers," *Procceeding of IEEE Conf on Parallel Processing*, pp. 155-161, 1985.

96. H M Gladney, "Data Replicas in Distributed Information Services," *ACM Transactions on Database Systems*, vol. 14(1), pp. 75-97, March 1989.

97. L Kleinrock, in *Queing Systems Volume 1-2*, John Wiley & Sons, 1975.

98. L H Shampine and C W Gear, "A User's View of Solving Stiff Ordinary Differential Equations," *SIAM Review 21*, pp. 1-17, 1979.

99. G T J Wuu and A J Bernstein, "Efficient Solutions to the Replicated Log and Dictionary Problems," *ACM Operating Systems Review*, pp. 57-66, 1984.

100. M H MacDougall, in *Simulating Computer Systems*, MIT Press, 1987.

Glossary

AC: Abbreviation for Available Copies Algorithm

Atomic operation: An operation either completes and (possibly) modifies the state of the system or it does not complete and has no effect on the system state.

Availability: The probability that the file will be accessible at any random point of time as times goes to infinity.

Bridge: A device that connects two networks typically at the link level and makes them appear as a single network.

Broadcast: The mechanism whereby a signal from one node on a network is received by all other nodes.

Combinatorial: A statistical approach combining basic probability theory with *k*-out-of-*n* reliability theory.

Decay Constant: A value representing the ratio with which the reliability graph declines as a function of time.

DFS: Abbreviation for Distributed File System

Effectiveness: Accessibility of a replicated file together with the other abstract properties of the consistency scheme such as assumptions made for its operability and correctness.

Ideal Network: A network in which partition failures are clean and nodes can detect partition failures almost instantaneously.

Logical file: An object of the replicated file system which is implemented by a set of physical files each holding a complete copy of the file and residing at a distinct processing node.

Multicast: The mechanism whereby a signal from one node on a network is received by a group of nodes.

MV: Abbreviation for Majority Voting Algorithm

Network Partition: A state in which bridge failures divide the network into multi-self functioning group of nodes.

Partition-free: A term used to describe networks in which partitioning never occurs.

Regeneration: Creating new replicas on available nodes in response to node failures.

Reliability: A conditional probability at a given confidence level that the file system will perform its intended function (read/write access) properly without failure and satisfy the specified requirements of continuous availability during a given time interval $[0,t]$. given period of time.

RH: Abbreviation for Reliable Histories Algorithm

RPC: Abbreviation for Remote Procedure Call

Serializability: A property which guarantees that that a replicated objects functional behavior is equivalent to that of single copy.

STR diagram: State-transition-rate flow diagram representing the availability or reliability behavior of the algorithm.

Steady-state: Equilibrium state behavior of the system as time goes to infinity.

Subnet: The smallest component of the network consisting a number of nodes connected by a communication link that cannot be partitioned further.

VWW: Abbreviation for Voting With Witnesses Algorithm

Appendix A

m	Number of processing nodes operating in the system
n	Number of file copies
k	Number of history copies
p	Probability of an individual node being up
p_r	Probability of an individual bridge being up
λ	Failure rate of individual processing nodes
μ	Repair rate of individual processing nodes
ρ	Failure to repair ratio of the system
γ	Copy regeneration rate
$P(A)$:	Probability of a file being accessible (availability at steady-state)
$P(A_f)$:	Probability of at least one file copy being available
$P(A_t)$:	Probability of the history table being accessible
$P(A_d)$:	Availability of the file during the bridge failure
$P(A_u)$:	Availability of the file when there is no partitioning in the network
$ts(h_i)$:	Last history update time on node i

$R(t)$: Reliability at time t where $R(0)$ is the steady-state availability

$A \times B \equiv \{(a, b) \mid a \in A \wedge b \in B\}$

$A \cup B \equiv \{a \mid a \in A \vee a \in B\}$

$A \cap B \equiv \{a \mid a \in A \wedge a \in B\}$

$a \Rightarrow b$: if proposition a then proposition b

$A \subset B : a \in A \Rightarrow a \in B$

$A \supset B : a \in B \Rightarrow a \in A$

(h, d): History table entries consisting of location and version number of physical file copies and a boolean flag set to *true* when the file is deleted: $h = \{(l_1, v_1), (l_2, v_2), \ldots\}$, $d = \{true, false\}$

mtbf: Exponential mean time between failures

mttr: Mean of the normally distributed repairs periods

Appendix B

The following method is followed for the analysis of the replication control system whose states are represented (Section 4.2) by three parameters; number of available copies, number of available nodes and the status of the copies(same, different). This system represents a Markov process under the observation that the future (being in a state) is conditionally independent of the past. This is the Markovian property specified by the transition probability of the states:

$$\pi_{i,j}(t) = P\{S_{m+t} = j \mid S_m = i\}.$$

In words, given that the present state is S_t, the past $(S_i, i<t)$, the future $(S_j, j>t)$ are conditionally independent, or given the history of the process $(S_i, i\leq t)$, the future $(S_{i'}, i'>t)$ depends only the present S_t.[1]

Let's define the *transition probability matrix* as

$$\Pi(t) = (\pi_{i,j}(t))$$

transition probabilities, Π (t) together with an initial distribution (at t=0 the system is assumed to be fully operating e.g. all nodes are up) determine the state probabilities $(p_j(t))$, which satisfies the forward equation

[1] In general, the future depends on the past — it is only conditionally independent given the present.

$$(d/dt)p_j(t) = \sum_i p_i(t)q_{i,j}$$

The matrix $Q = (q_{i,j})$ is the *infinitesimal generator* of the Markov process where

$$\Pi(t) = I - Qt + o(t)$$

or in open form,

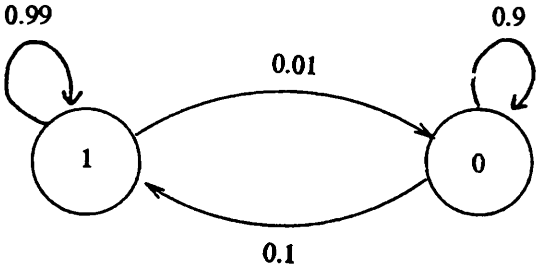
$$\pi_{i,i}(t) = 1 - q_{i,i}t + o(t) \quad as \quad t \to 0$$
$$\pi_{i,j}(t) = q_{i,j}t + o(t) \qquad as \quad t \to 0 \quad (if\ i \neq j)$$

The interpretation of the generator matrix is that, the system remains in a state for some random amount of time. The time in state i is exponentially distributed with

$$1 - exp(-q_{i,i}t)$$

When the system leaves state i, it makes a transition to state j with probability $-q_{i,j}/q_{i,i}$. The mathematical details can be found in reference (87).

The following example employs the above method to a two state system whose transition rates are shown below.



This diagram represents a system with:

$$\Pi = \begin{bmatrix} 0.99 & 0.01 \\ 0.10 & 0.90 \end{bmatrix}$$

The generator matrix (Q) is found and its inverse is divided by the determinant. The resulting column vector, R, is the state probabilities as $t \to \infty$.

$$\Pi^T - I = \begin{bmatrix} -0.01 & 0.10 \\ 0.01 & -0.10 \end{bmatrix}$$

Following are the inverse of the generator matrix Q and state probability matrix R:

$$Q^{-1} = \begin{bmatrix} 1 & -0.10 \\ -1 & -0.01 \end{bmatrix}$$

$$\underset{j=1,2}{R_j(\infty)} = \frac{\begin{bmatrix} 1 & -0.10 \\ -1 & -0.01 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix}}{-0.11} = \begin{bmatrix} 0.91 \\ 0.09 \end{bmatrix}$$

Therefore this two-state system yields a solution at steady state:

$$P(1) = 0.91 \quad P(2) = 0.09 \quad \text{as } t \rightarrow \infty$$

The solutions to the system applying various consistency algorithms have been given in Chapter Four. The results have been presented in Figure 4.1 to Figure 4.4. The following diagrams represent the states of the system for available copies and majority voting algorithms with n copies.

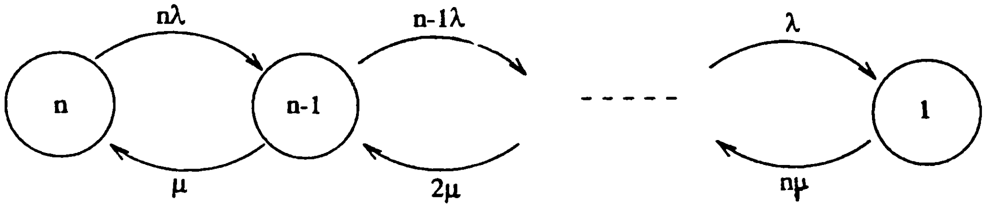


Figure B.1. *STR diagram for the availability with AC (n copies)*

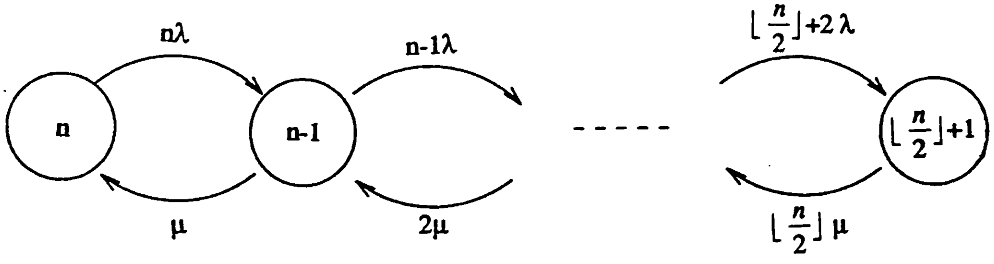

nλ
n-1λ
$\lfloor \frac{n}{2} \rfloor + 2\lambda$
n
n-1
- - - - -
$\lfloor \frac{n}{2} \rfloor + 1$
μ
2μ
$\lfloor \frac{n}{2} \rfloor \mu$

Figure B.2. *STR diagram for the availability with MV (n copies)*

Equilibrium State Conditions of the system when m=5

The following state-transition-rate diagram represents the *reliable histories* algorithm's availability behavior in a system of five processing nodes:

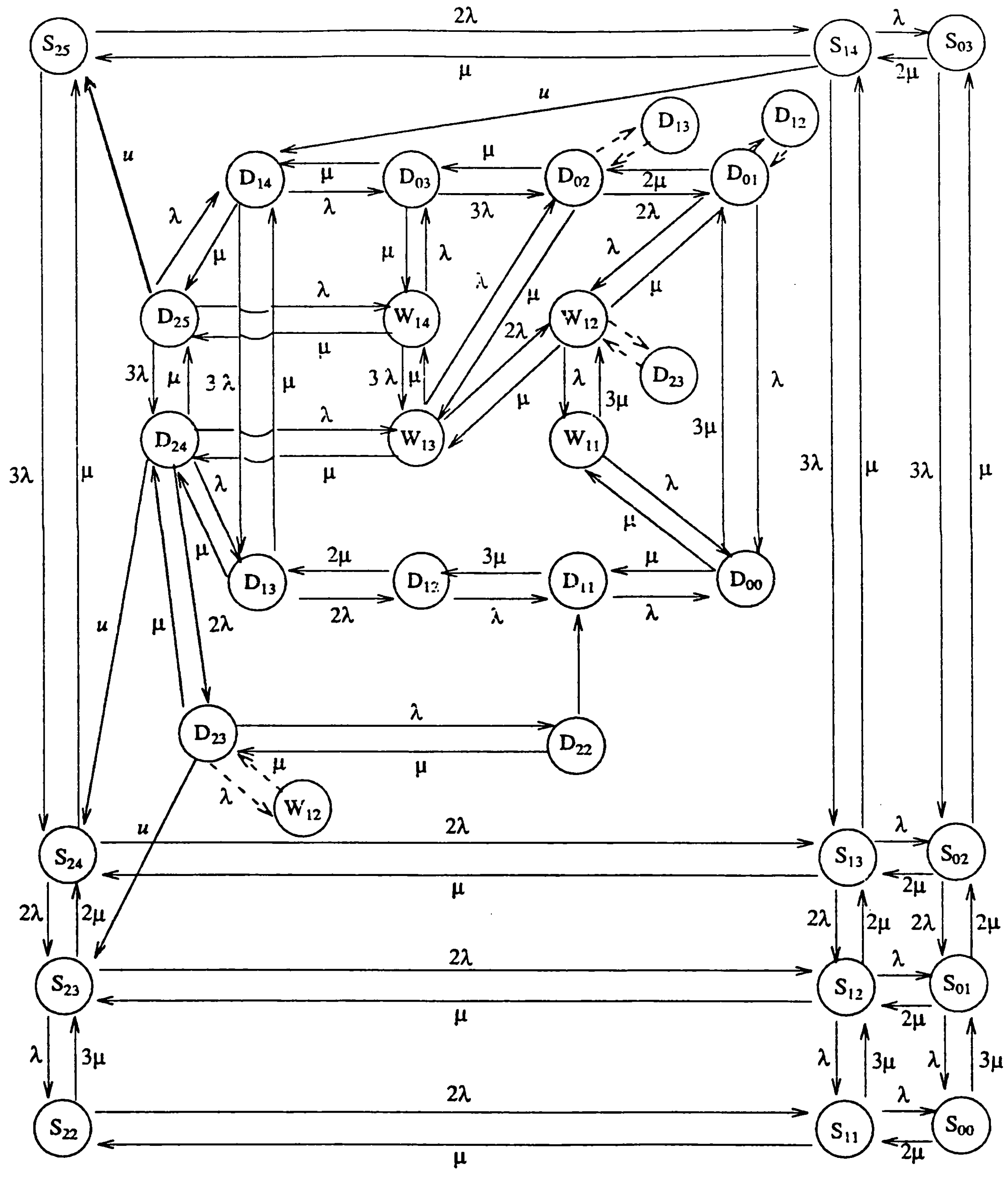


Figure B.3. *STR diagram for the availability with RH (m=5)*

There are a total of 28 states. Equilibrium state conditions representing some of these states are given below. These conditions are derived from the above diagram and example states show a group of transactions for different status conditions of file replicas (*S*, *D*, *W*). Each row of the transition probability matrix represents the equilibrium condition of an individual state under the observation that for any state *i*

$$\sum_{j=0}^{n} \Pi_{i,j} = 1$$

$$\begin{aligned}
\mathrm{P}_{S_{25}} 5\lambda &= (\mathrm{P}_{S_{24}} + \mathrm{P}_{S_{14}})\mu + \mathrm{P}_{D_{25}} u \\
\mathrm{P}_{S_{24}} (4\lambda + \mu) &= 3\mathrm{P}_{S_{25}} \lambda + (\mathrm{P}_{S_{13}} + (2\mathrm{P}_{S_{23}})\mu + \mathrm{P}_{D_{24}} u \\
\mathrm{P}_{S_{14}} (6\lambda + \mu + u) &= 2\mathrm{P}_{S_{25}} \lambda + (\mathrm{P}_{S_{13}} + 2\mathrm{P}_{S_{03}})\mu \\
\mathrm{P}_{S_{23}} (3\lambda + 2\mu) &= 2\mathrm{P}_{S_{24}} \lambda + (2\mathrm{P}_{S_{22}} + \mathrm{P}_{S12})\mu + \mathrm{P}_{D_{23}} u \\
\mathrm{P}_{S_{13}} (3\lambda + 2\mu) &= (3\mathrm{P}_{S_{14}} + 2\mathrm{P}_{S_{24}})\lambda + (2\mathrm{P}_{S_{12}} + 2\mathrm{P}_{S02})\mu
\end{aligned}$$

$$\begin{aligned}
\mathrm{P}_{D_{25}} (5\lambda + u) &= (\mathrm{P}_{D_{14}} + \mathrm{P}_{D_{24}} + \mathrm{P}_{W_{14}})\mu \\
\mathrm{P}_{D_{24}} (4\lambda + \mu + u) &= 3\mathrm{P}_{D_{25}} \lambda + (2\mathrm{P}_{D_{23}} + \mathrm{P}_{D_{13}} + \mathrm{P}_{W_{13}})\mu \\
\mathrm{P}_{D_{14}} (2\lambda + \mu) &= \mathrm{P}_{D_{25}} \lambda + (\mathrm{P}_{D_{13}} + \mathrm{P}_{D_{03}})\mu + \mathrm{P}_{S_{14}} u
\end{aligned}$$

$$\begin{aligned}
\mathrm{P}_{W_{14}} (4\lambda + \mu) &= \mathrm{P}_{D_{25}} \lambda + (\mathrm{P}_{D_{03}} + \mathrm{P}_{W_{13}})\mu \\
\mathrm{P}_{W_{13}} (3\lambda + 2\mu) &= (\mathrm{P}_{D_{24}} + 3\mathrm{P}_{W_{14}})\lambda + (\mathrm{P}_{D_{02}} + 2\mathrm{P}_{W_{12}})\mu
\end{aligned}$$

$$A = \sum (P_{S_{25}}, P_{S_{24}}, P_{S_{23}}, P_{S_{14}}, P_{S_{13}}, P_{D_{25}}, P_{D_{24}}, P_{D_{23}}, P_{D_{14}}, P_{D_{13}})$$

The above conditions generate a 28×28 *Q* matrix of which some entries are given below.

$$Q = \begin{bmatrix}
1-5\lambda & 3\lambda & 2\lambda & 0 & 0 & 0 & \cdot & \cdot \\
\mu & 1-(\mu+4\lambda) & 0 & 2\lambda & 2\lambda & 0 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{bmatrix}$$

Appendix C

The following model narrows the unlimited spare assumption to that in which the number of nodes in the system is finite. Two state-transition-flow-rate diagrams for the reliability of regenerative-*reliable histories* method with finite spares are given in Figure C.1 and Figure C.2. In the first model, the system has 2 copies of the file and 1 spare node (in other words, m=3 where 2 of these nodes contain a copy of the file). In the second model, the system has 2 copies of the file and 3 spare nodes (m=5).

The differential equation describing the behavior of the system within the given parameters (number of spares, number of copies) managed by the method is derived from the state-transition-rate diagrams. The following model describes a general method for the system maintaining n copies with an additional m spare nodes. It is a costly model in terms of complexity. A similar model is derived by Long in reference (58) for the analysis of regenerative-voting and regenerative-available copies algorithms. The below model adopts the same technique for the RH algorithm and discusses the complexity of this analytic method.

The system is in state (j,k) if j copies are immediately accessible and k nodes are currently available as spares. The state 0 denotes the inaccessible state.

Definition C.1. The *reliability* $R(n,m,t)$ of an n*-copies* system with m spares managed by *reliable histories* algorithm is defined as the probability that the system will operate correctly over time interval of duration t given that an initial complement of n copies and m spares were operating correctly at time $t=0$.

As shown in the figures, finite-spares lead to more complex sets of equations. These systems can be represented with linear, constant coefficient ordinary differential equations(ODEs) of the form

$$P'(t) = AP(t)$$

with initial condition

$$P(0) = P_0$$

The solution is given analytically by

$$P(t) = e^{tA}P_0$$

where e^{tA} denotes the matrix exponential.

For simplicity of exposition, assume A has full geometric multiplicity. Its Jordan canonical form

$$A = T\Lambda T^{-1}$$

consists of the diagonal matrix Λ with eigenvalues, λ_i, of A on the diagonal and T, whose columns are the eigenvectors of A. The ODE can then be diagonalized:

$$P'(t) = T\Lambda T^{-1}P_0$$

Defining

$$Z(t) = T^{-1}P(t)$$

the differential equation is

$$Z'(t) = \Lambda Z_0$$

with solution

$$Z(t) = e^{t\Lambda}Z_0$$

where $e^{t\Lambda}$ is the diagonal matrix with entries $e^{t\lambda_i}$, $i = 1, \ldots, n$. The general solution is thus

$$P(t) = Te^{t\Lambda}T^{-1}P_0$$

which can be evaluated at any point t in time.

This procedure is costly. The vector $Z_0 = T^{-1}P_0$ need only be computed once, requiring approximately $\frac{1}{3}n^3$ flops, where a flop is a floating point add coupled with a floating point multiply. The n exponentials $e^{t\lambda_i}$ that comprise $e^{t\Lambda}$ would be formed for each value of t of interest. The cost would be reduced by computing the solution at equally spaced points $T_k = k.\Delta t$ using

$$e^{t_{k+1}\lambda_i} = e^{\Delta t\lambda_i} \times e^{t_k\lambda_i}.$$

The propagation matrix

$$e^{\Delta t\Lambda} = \begin{bmatrix} e^{\Delta t\lambda_1} & 0 & \cdots & 0 & 0 \\ 0 & e^{\Delta t\lambda_2} & \cdots & 0 & 0 \\ \cdot & \cdot & \cdots & \cdot & \cdot \\ \cdot & \cdot & \cdots & \cdot & \cdot \\ \cdot & \cdot & \cdots & \cdot & \cdot \\ 0 & 0 & \cdots & e^{\Delta t\lambda_{n-1}} & 0 \\ 0 & 0 & \cdots & 0 & e^{\Delta t\lambda_n} \end{bmatrix}$$

need only be formed once, and later time step values can then be formed recursively from the previous step beginning with P_0 and using

$$P_{k+1} = T.e^{\Delta t\Lambda}.P_k$$

at a cost of one matrix-vector multiply (n^2 flops) per step. The major cost, though, arises when the eigensystem of A is computed.

Obtaining the eigensystem of A is equivalent to finding the roots of its characteristic polynomial. It was shown by Evariste Galois in reference (89) that there is no direct method for computing the roots of a polynomial of degree higher than 4. This implies that, since the model has more than 5 states it requires an iterative process to obtain the eigensystem. The most effective method is the QR algorithm. Actual convergence of the iterative QR algorithm depends on the problem and the conditioning of the eigenvectors, but this one-time cost is estimated at $15n^3$ flops. This is shown in reference (58).

Thus, computing the matrix exponential directly is very costly and alternate methods of solution are desirable. For small configurations Runge-Kutta method is suggested. For larger configurations, the following inverse Taylor method is an effective technique.

The Taylor method can exploit the linear, constant coefficient nature of problem. The matrix exponential is defined by the convergent power series

$$e^{tA} = I + tA + \frac{(tA)^2}{2!} + \frac{(tA)^3}{3!} + \cdots$$

Truncating this series after 6 terms yields a fifth order numerical approximation to the propagation matrix, $e^{\Delta tA}$ above, Let

$$t_5 = I + \Delta tA + \frac{(\Delta tA)^2}{2!} + ... + \frac{(\Delta tA)^5}{5!}$$

which can be formed effectively using nested multiplications in 4 matrix-matrix multiplies costing about n^3 flops each. Using t_5 to approximate the solution at forward time steps,

$$P_{k+1} = t_5 P_k \simeq e^{\Delta tA} P_k = (e^{\Delta tA})^{k+1} P_0 = e^{t_{k+1}A} P_0$$

The cost per step is just the n^2 flops for the matrix-vector multiplication.

Unfortunately, the coefficient matrices that tend to arise in the reliability model have eigenvalues that are all real and non-positive. Summing these series leads to a loss of significance in forming t_5 since the terms will alternate in sign.

$$t_5 = T\{I + \Delta t\Lambda + \frac{\Delta t^2}{2!}\Lambda^2 + \cdots + \frac{\Delta t^5}{5!}\Lambda^5\}T^{-1}$$

A slightly more expensive approximation of the inverse of $e^{\Delta tA}$ is required. Just as an approximation of the scalar $e^x$for $x<0$ is accomplished more accurately in the presence of finite precision arithmetic using

$$e^x = \frac{1}{e^{-x}} \approx \frac{1}{1-x+\frac{x^2}{2!}-\frac{x^3}{3!}+\frac{x^4}{4!}-\frac{x^5}{5!}}$$

The matrix equivalent is

$$e^{-\Delta TA} \approx i_5 = I - \Delta tA + \frac{(\Delta tA)^2}{2!} - \frac{(\Delta tA)^3}{3!} + \frac{(\Delta tA)^4}{4!} - \frac{(\Delta tA)^5}{5!}$$

Besides the cost of forming the series using nested multiplication as before, there is also the one-time cost of about $\frac{1}{3}n^3$ to factor the matrix. The solution values are given by solving the linear system

$$i_5 P_{k+1} = P_{k,}$$

costing about n^2 flops per output point. This allows the flexibility to change the number of terms for the series depending on the solution behavior.

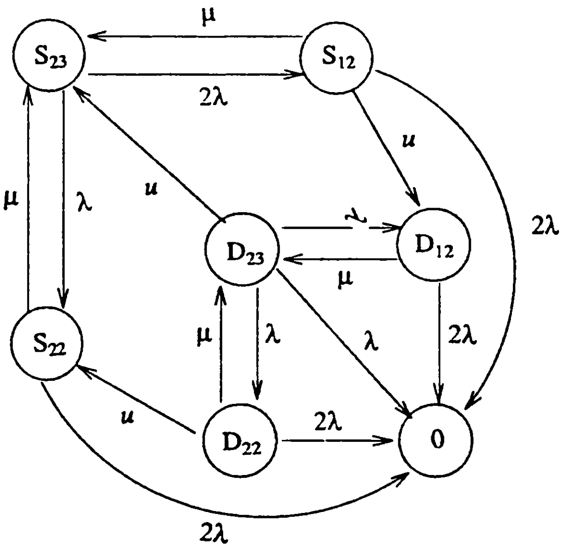


Figure C.1. *STR diagram for the reliability with RH (m=3)*

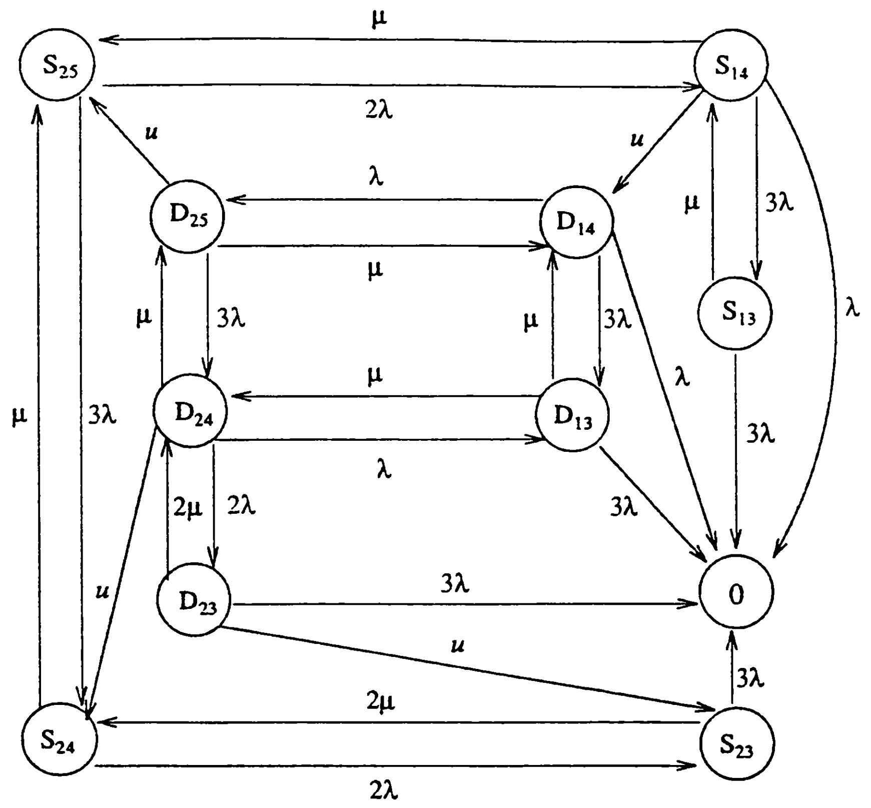


Figure C.2. *STR diagram for the reliability with RH (m=5)*

Appendix D

Simulation Activities

The dual approach of analysis and simulation is a common technique amongst system designers because it offers a degree of confidence through two sets of supporting results. Though the major performance measures for availability are gained from the two statistical analysis presented in Chapter Four, an analytical approach for reliability is far harder to develop in the same degree for the algorithm proposed because of the number of states involved in a model which gains advantage over the other algorithms. A suite of software simulation has been developed to support the analytical work on availability and to obtain the major performance measures for reliability of the consistency schemes while supporting a minor analytical work in this area.

The steady-state availability observations are made for two different failure models using several failure modes. In the first model, availability of individual processing nodes are obtained from a uniform number generator. This simulation is used to verify the results obtained from the combinatorial analysis in Section 4.1. In the second model, failure times and repair periods of individual nodes are generated from an exponential distribution with the means *mtbf* and *mttr* which produce the failure to repair ratio for the nodes corresponding to the same values of individual node availabilities used in the combinatorial analysis simulation. The results of simulation for both analysis have been given in Figure 4.4 and Figure 4.8 to support the upper and lower bounds of availability provided by the algorithm proposed and simple partitioning analysis. Note that the two sets of results digress more in the upper bound analysis. This is due increase in the confidence interval as the number of nodes are increased resulting from the behavior of random failure time generators. The results shown represent the average of around 20 simulations for 50,000 simulation time period.

The *C* programming language was used to implement simulation. The system is regarded as a set of processes, comprising a set of network operations initiated from a local node in order to access the replicated file. The nodes are operating in parallel and interacting with each other via communication links. Changes of state in the nodes can only take place in accordance with simulation clock pulses. The parallel processes are implemented sequentially before the clock ticks.

Graph-Walk Algorithm

The simulation of partitioned systems is rather more complicated. This analysis has been done by defining the topology as a graph whose nodes represents the vulnerable components; subnets, bridges, processing nodes. The *Graph–Walk* algorithm is used to check the state of individual nodes of the graph. This algorithm calculates the availability of replicated data in a network of vulnerable components with independent availabilities. The availability appears different from node to node. The network is described using a connection matrix where con_matrix[i][j] = 1 if there is a connection between node i and j, 0 otherwise. Node failures are exponential and the repairs are normally distributed. Failure[i][1] keeps the next failure time of node i and Failure[i][2] keeps the corresponding repair time for this particular failure.

collect (src, have)

/* This is a recursive function to walk the graph starting in node 'src'. It returns in 'have'

```
the number of available nodes at current_time. */

define case-1: Failure[src][1] > current_time    /* node failed */
define case-2: current_time = Failure[src][1] + Failure[src][2]    /* node repaired */
visited[src] = 1;   /*mark the visited nodes*/
if case-1
then have = have + 1;   /*if the node is reached acquire the votes*/
    if case-2   /*find next failure time */
    then Failure[src][1] = Failure[node][1] + Failure[node][2] + expntl(mean);
         Failure[src][2] = normal(mean,st_dev);
for (dst=1; dst<=nodes; dst++)   /*visit all reachable nodes */
    if ( (visited[dst] == 0 ) & (con_matrix[src][dst] == 1) )
    then have=collect(dst,have);
collect=have;
```

activator()

/* this function calls the *collect*() function to check the availability of the majority of nodes starting from a randomly chosen node; 'node'. The total number of nodes is stored in 'nodes' and the number of tables required for access in 'tables' */

```
            .
            .
            node=random(1,nodes);
            if ((t_avail = collect(node)) < tables)
            .
            .
```

Example Scenarios Employing the *Range* Algorithm

Steps of the *range* algorithm during consecutive calls of *ReadHistory* and *WriteHistory* operations are given below. Each scenario consists of an update request on the file therefore requires an independent read in the history, finds the *Read.Set*, returns the up-to-date history table and after the file operation is performed, writes back the new history on the available nodes. The file operations are excluded from scenarios as they are assumed to be always successful.

The following terms are used in the scenarios:

$ReadHistory_{(i,j)}$:

The function is initiated from node i with the request number j

$w=\{(R_{in}, R_{out}, t_x, table_x)_{i,j..}, (R_{in}, R_{out}, t_y, table_y)_{k,l..} \cdots \}$: $where x \neq y$

The members of the set w are the replies returned to the readHR() (see Chapter Six). Each reply is subscribed by its sender i.e. nodes i and j return the same reply with $table_x$ (last table update time is t_x) and, nodes k and l return the same reply with $table_y$ (last table update time is t_y).

$WriteHistory_{(i,j)}$:

The function is initiated from node i the corresponding table read request number is j. It writes the new table and the new *range* sets. The nodes which are accepted the new table become a member of the new R_{in} set.

return($(h,d) = table_i$):

returns the table from the reply of node i

$A_i = \{ \cdots \}$:

The set of available nodes during the scenario i

The scenarios are not related but each scenario takes up the system parameters from previous scenario. Initial parameters are: $m = 7$ (numbered 1..7), $R_{in} = \{1,2,3,4\}$ and $R_{out} = \{\}$.

$A_1 = \{ 1, 5, 6, 7 \}$, node(5): *write* (f)

$\mathit{ReadHistory}_{(5,1)}$:

$w = \{(\{1, 2, 3, 4\}, \{\}, t_0, \mathit{table}_0)_1\}$

$\mathit{Read.Set} = \{1, 2, 3, 4\}$

$\mathit{return}((h,d) = \mathit{table}_1)$

File operations are performed here (updateF , [copyF])

$\mathit{WriteHistory}_{(5,1)}$:$\{(\{1, 5, 6, 7\}, \{\}, t_1, \mathit{table}_1)_{1,5,6,7}\}$

$A_2 = \{1,2,3,4,7\}$, node(1): *write* (f)

$\mathit{ReadHistory}_{(1,2)}$:

$w = \{(\{1, 2, 3, 4\}, \{2\}, t_0, \mathit{table}_0)_{2,} (\{1, 2, 3, 4\}, \{3\}, t_0, \mathit{table}_0)_{3,}$

$(\{1, 2, 3, 4\}, \{4\}, t_0, \mathit{table}_0)_{4,} (\{1 ,5, 6, 7\}, \{\}, t_1, \mathit{table}_1)_7\}$

$\mathit{Read.Set} = \{1, 5, 6, 7\}$ since $\max(t) = t_1$

$\mathit{return}((h,d) = \mathit{table}_{1 \vee 7})$ — (both nodes (1,7) contain the up-to-date history. As 1 is the local node and 7 has replied to the history read request and returned the table as well, either table can be returned here)

File operations are performed here (updateF , [copyF])

$\mathit{WriteHistory}_{(1,2)}$:$\{(\{1, 2, 3, 4, 7\}, \{\}, t_2, \mathit{table}_2)_{1,2,3,4,7}\}$

$A_3 = \{5,6,7\}$, node(5): *write* (f)

$\mathit{ReadHistory}_{(5,3)}$:

$w = \{(\{1, 5, 6, 7\}, \{5\}, t_1, \mathit{table}_1)_5, (\{1, 5, 6, 7\}, \{6\}, t_1, \mathit{table}_1)_6,$

$(\{1, 2, 3, 4, 7\}, \{\}, t_2, \mathit{table}_2)_7\}$

$\mathit{Read.Set} = \{1, 2, 3, 4, 7\}$ since $\max(t) = t_2$

$\mathit{return}((h,d) = \mathit{table}_7)$

The *write* request does not succeed since the *WriteHistory* returns error — only three nodes are available.

$A_4 = \{3, 4, 5, 6, 7\}$, node(5): *write* (f)

$ReadHistory_{(5,4)}$:

$w = \{(\{1, 2, 3, 4, 7\}, \{3\}, t_2, table_2)_3, (\{1, 2, 3, 4, 7\}, \{4\}, t_2, table_2)_4,$

$(\{1, 5, 6, 7\}, \{5\}, t_1, table_1)_5\}, (\{1, 5, 6, 7\}, \{6\}, t_1, table_1)_6\}, (\{1, 2, 3, 4, 7\}, \{\}, t_2, table_2$

$Read.Set = \{1, 2, 3, 4, 7\} - \{3, 4, 5, 6\} = \{1, 2, 7\}$ since $\max(t) = t_2$

$return((h,d) = table_7)$

File operations are performed here (updateF , [copyF])

$WriteHistory_{(5,4)}: \{(\{3, 4, 5, 6, 7\}, \{\}, t_4, table_4)_{3,4,5,6,7}\}$

$A_4 = \{1, 2\}$

$ReadHistory_{(1,5)}$:

$w = \{(\{1, 2, 3, 4, 7\}, \{1\}, t_2, table_2)_1, (\{1, 2, 3, 4, 7\}, \{2\}, t_2, table_2)_2\}$

$Read.Set = \{1, 2, 3, 4, 7\} - \{1, 2\} = \{3, 4, 7\}$ since $\max(t) = t_2$

Since non of the nodes from the *Read.Set* returned a reply, *ReadHistory* returns *error* — table is not available.

Appendix E

The first paper was presented at the *IEEE COMPCON Spring'*89 conference in San Fransisco in February 1989. This paper discusses the RH algorithm and compares the steady-state availability for small number of copies. A similar paper was also published in the proceedings of the ISCIS-V International Conference held in Cesme, Turkey in November 1989.

The second is the position paper which will be presented at the *IEEE Workshop on Management of Replicated Data* in Houston in November 1990. This paper presents an original work on the effect of network partitions on reliability.

A Low Cost File Replication Algorithm

B S Bacarisse *S Bek Baydere*

Department of Computer Science
University College London
London WC1E 6BT.

ABSTRACT

An algorithm suitable for reading and writing replicated files is described. It provides high availability with very low replication factors by combining variations of existing replication control strategies. The algorithm is presented together with some statistical analysis, comparing the availability provided by this and other well known methods.

1. Introduction

Increased reliability is often quoted as one of the principal advantages of distributed systems. Unfortunately the potential for fault tolerance that is offered by distributed systems has only been realised in a few applications, such as real–time control of life–threatening processes and financial transaction processing, where the increased complexity of the system can be justified by the unacceptable cost of a failure. Commercially available distributed file systems have tended to concentrate on the problems providing efficient remote access, rather than offering increased reliability through replication. Ironically, system managers may then be tempted to distribute functionality across the network, thereby *decreasing* the overall reliability of applications.

The authors believe that there are three main design criteria that must be met before replicated file systems can become commonplace in general purpose computing environments. First, the communications overhead inherent in any replicated system must be brought within acceptable bounds. Secondly, the storage cost of replicating files must be kept down and, thirdly, mechanisms need to be provided to allow control over the level of reliability (or replication) required for particular sets of files.

This paper presents an algorithm for controlling access and updates to replicated files. It combines the advantages of available copy algorithms (high availability) and voting algorithms (consistency in the face of network partition) to provide fault tolerance with low levels of replication (even 2 copies) while keeping the communication overheads incurred by file operations down. The design is aimed at loosely–coupled systems; typically a collection of workstations linked by a local area network. It is not suitable for database applications which require concurrency control, stringent consistency constraints and support for atomic transactions. Our main concern was to provide fault tolerance efficiently as an extension to file systems that, typically, do not provide transactions nor ensure consistency in the face of concurrent updates.

The design contains some support for controling the level of replication and the placement of copies, although how these facilities are presented to users and system administrators needs, we believe, much more work. The rest of this paper consists of a summary of related work in section 2 and an overview of the file system design and a description of the algorithms in section 3. Section 4 presents some analytical results concerning file availability and communication overheads. The section concludes with a discussion of some possible implementation techniques and future work.

2. Relevant Work

Consistency schemes for replication control can be divided, broadly, into *voting* algorithms and *available copy* algorithms. The latter group are intellectual descendants of Alsberg's primary site algorithm.[1] Failed sites are dynamically detected by high priority status transactions and configured out of the system while newly recovered sites are configured back in. Recovered sites bring themselves up to date by copying from other sites before accepting any user transactions. Clients may read data from any available copy but must write to all available copies. This form of unanimous update[2] provides better availability than all other methods but does not prevent inconsistencies in the presence of communication failures such as network partition. Only clean, detectable, site crashes are handled correctly by this method.

El–Abbadi *et al*[3] extended this method to handle partitions. Two approaches were proposed. In the first, nodes maintain virtual partitions which are logical groups corresponding to actual partitions. Unanimous update being used within each virtual partition. Only a virtual partition containing a majority of the replicas may access the data. The second offered greater flexibility. In this system, nodes maintain views similar to virtual partitions but within each view weighted voting is used between sites.

The latest variation of the available copy algorithm is *regeneration*.[4] Here, the availability of the data is restored immediately after a node crash by regenerating the failed copy on a new available node. Again, this approach cannot maintain consistancy in the case of network partitions.

In voting algorithms, each site maintains a number of votes. Client requests must gather a quorum of votes before being accepted. In its simplest form, *majority* voting every copy, or site, has one read and one write vote. For a request to be accepted a majority need to approve it. The algorithm employs time–stamps both in the voting procedure and in the application of updates to data copies.

In *weighted* voting[6,7,8] sites may be assigned different numbers of votes. Read transactions collect a read quorum of r votes to read a file, and a write quorum of w votes to write a file, such that $r + w$ is greater than the total number of votes assigned to the file. There is then always an intersection between read and write quorums so every read quorum is guaranteed to include an up to date copy. Weighted voting provides serial consistency which means that it appears to each transaction that it is running alone.

Quorum consensus and primary copy methods work well if the number of copies is large. A number of variations have been proposed for reducing the storage cost of the algorithm by replacing some of the copies by so called *witnesses*[9] that record only the current status of the file and for increasing the write availability by replacing unavailable copies by so called *ghosts*[10] that vote but hold no actual data.

Most of the work in this area has grown out of database applications where the cost can be justified by the requirement for availability and consistency. Some distributed file systems[11,12] have realised the potential of replication for fault tolerance. Most, like LOCUS,[13] use a simple primary copy algorithm[2] because of its simplicity and relatively high reliability at low levels of replication. Voting algorithms have also been used. KUDOS[14] uses the majority consensus approach with a locking mechanism[15] for concurrency control and voting with ghosts is currently being implemented for the AMOEBA[16,17] file system.

These voting schemes offer consistency even with serious communication failures, such as network partititions, but require at least three copies for practical use. The authors' approach can provide some of the advantages of both techniques: high availability with low levels of replication, combined with resilience to serious network and server failures.

3. System Model and Algorithms

We view a distributed computing system as a finite set, $M = \{1, 2, 3, ...,m\}$, of processing nodes connected by a data network. In the absence of failures the underlying network routes messages between these nodes. Nodes may crash, the network may fail and its failure may result in the system becoming partitioned. Failed nodes and links can recover spontaneously or because of system maintenance. It is assumed that a processing node can determine the status of another only by receiving a message from that node. A node is said to be available from another if both are running and the network can route packets between them (both ways). It is assumed that a transport level protocol will provide reliable error free communication between nodes, in fact a remote procedure call (RPC) mechanism[18,19] that offers at-most-once semantics is likely to be the best communication protocol for most of the algorithms.

The algorithms presented here are designed with this model of a distributed computing system in mind. The file system, as a whole, is configured in the following way.

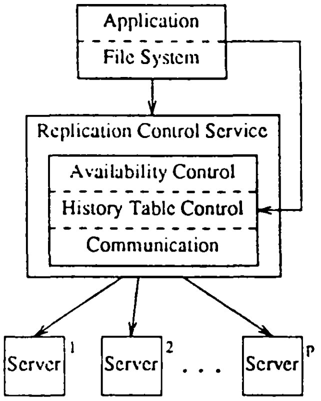


Figure 1 — Logical structure of the file system.

The file system presents applications with the abstraction of a logical file consisting of a sequence of bytes and identified by a unique identifier $f \in F$. They are considered to be sequences of bytes, any subsequence of which may be read or replaced by any other byte sequence. The file system provides four operations on these logical files: *create*, *read*, *write* and *delete*.

Logical files are implemented by a (possibly empty) set of physical files each holding a complete copy of the data in the logical file and each residing at a single, distinct processing node shown on the diagram as a file server. Two protocol layers within the Replication Control Service co–ordinate access to the physical copies so as to ensure that read requests return the most up–to–date version of the file. The *availability control* layer determines the appropriate update strategy for a file operation based on the file's history table. The history table records the location and version number of each copy along with a flag that marks the file as having been deleted. The history is maintained by the *history table control* layer and is discussed below.

The number and location of the copies of each file are controllable by its owner, and both may change during the file's lifetime. This is implemented as an interface into the lower control layers of the replication control system.

Notation

The algorithms in this paper are presented in a pseudocode based on set notation and predicate calculus mainly because the algorithms rely heavily on set manipulation and require very little in the way of conventional control structures. The following sets are used throughout.

$T = \{true, false\}$
Z the integers
M the set of processing nodes
F the set of file identifiers
$V = 2^{M} \times Z$ (sets of server and version number pairs)
Σ^* the set of byte strings

The notation 2^S denotes the power set of S, ie the set of all subsets of S.

4. The Availability Control Layer

Each logical file $f \in F$ has a history table $h(f) \in V \times T$ that records the version numbers and locations of every physical copy of the file along with a boolean flag used to mark the file as having been deleted. Two operations,

$$ReadHistory: F \rightarrow V \times T$$

$$WriteHistory: F \times (V \times T) \rightarrow \{success, error\}$$

are provided to manipulate this table. A read of the table for a particular file returns this flag together with the locations and corresponding version numbers of its physical copies. A write to the table records new machine and version number pairs and can be used to set the delete flag. This history information is itself replicated using a separate algorithm; the details of which are described in the next section.

At file creation time, all copies are in *equilibrium*, with all version numbers zero:

```
$create(f, S)$: $F \times 2^S \rightarrow \{success, error\}$
    let $h \leftarrow \{(m,0) \mid m \in S\}$
    return $WriteHistory(f, (h, false))$
```

No physical copies are accessed until the file is first written to.

The *delete* simply attempts to write a new history that records the file as having been deleted using the flag already mentioned.

```
$delete(f)$: $F \rightarrow \{success, error\}$
    return $WriteHistory(f, (empty, true))$
```

A file may be in any one of four availability states, determined by inspecting its reliable history table.

1) All copies are available and up to date.
2) All available copies are up to date but some copies are unavailable.
3) Some of the available copies are not up to date.
4) No up to date copy is available.

The availability control layer determines the appropriate access and update policy for each state. In order to read a file it must be in state 1, 2 or 3, ie at least one available copy must be up to date:

```
$read(f, posn, size)$: $F \times Z \times Z \rightarrow \Sigma^* \cup \{error\}$
    let $(h, d) \leftarrow ReadHistory(f)$
    if $d = true \vee h = \varnothing$
    return $error$
    let $latest \leftarrow \max(\{i \mid \exists (m, i) \in h\})$
    if $latest < 0$
    return $error$ — can't happen!
    let $U \leftarrow \{m \mid \exists (m, latest) \in h\}$
    return readF(U, $f$, $posn$, $size$)
```

If the file has been deleted ($d = true$) or the file's history is not available ($h = \varnothing$) then the operation fails. Negative version numbers are used by the *configure* operation to mark new copies that must be brought up to date at the next write, and must therefore be excluded from read operations, although the algorithm used ensures that there will always be on positive version number in any history. The set of servers holding copies with the highest version number is found and a read request is multicast to them using readF, defined as follows.

readF($U, f, posn,$: $2^M \times F \times Z \times Z \rightarrow \Sigma^* \cup \{error\}$
returns data (specified by size and position) obtained from any server in the set U, or an error indication if no server responds.

An alternative read operation can be provided for files in availability state 4 that will read the most up to date version that is available. We expect this operation to be used when all up to date copies have been lost forever (by disc failure, for example) or the file has very weak consistency requirements. For example it may be better to read an old version of a host address table than none at all.

The algorithm is similar to that for the read operation above, except that it iterates if the readF request fails, picking all servers that hold versions one less than *latest* until the lowest (positive) version number has been tried. We do not present the details here for reasons of space.

A *write* operation will succeed if at least one file copy can be updated and the file's new history can be recorded. The update is multicast to all up to date copies, and servers holding out of date versions are asked to copy the new file. The set of servers which accepted either the update, R, or a new copy, C, will hold up to date versions and this is recorded in the new history with an incremented version number for these sites. Untouched servers have their history table entries copied into the new table from the old one. If all servers either accepted the update or took a fresh copy, (availability state 1), then the file can be put back into equilibrium by recording all the version numbers as zero.

```
$write(f, data, posn)$: $F \times \Sigma^* \times Z \rightarrow \{success, error\}$
    let $(h, d) \leftarrow ReadHistory(f)$
    if $h = error \vee d = true$
    return $error$
    let $latest \leftarrow \max(\{i \mid \exists (m, i) \in h\}$
    if latest < 0
    return $error$ — can't happen!
    let $U \leftarrow \{m \mid \exists (m, latest) \in h\}$
    let $R \leftarrow$ updateF(U, $f$, $data$, $posn$)
    if $R = \varnothing$
    return $error$
    let $S \leftarrow \{m \mid \exists (m, i) \in h\}$
    let $C \leftarrow$ copyF(R, $f$, $(S - R)$)
    if $C \cup R = S$
    then let $h' \leftarrow \{(m, 0) \mid m \in S\}$
    else let $h' \leftarrow \{(m, latest+1) \mid m \in C \cup R\}$
              $\cup \{(m, i) \in h \mid m \notin C \cup R\}$
    return $WriteHistory(f, (h', false))$
```

The following support operations are invoked on physical file copies:

updateF(U, f, $data$, $posn$): $2^M \times F \times \Sigma^* \times Z \rightarrow 2^M$
Multicasts a request to write *data* to the file f to all the servers in the set U. It returns the set of servers that accepted the request.

copyF(R,f,X): $2^M \times F \times 2^M \rightarrow 2^M$

Copies the file f from any server in the set R to all the servers in the set X. Again, the result is the set of servers that accepted the operation.

The *configure* operation allows the set of servers that file copies of a file to be changed during the file's lifetime. The table is reconfigured so as to ensure that the new server set contains at least one up to date copy. If the intersection between the old and the new server sets is empty and none of the servers in the new set are available to take a copy of the file, the operation will fail. Copies are deleted simply by modifying the history table. New new sites that are unavailable to take a copy of the file are added with negative version numbers. The *write* algorithm will bring them up to date as soon as possible, and the *read* algorithm will ignore these copies. The details are a little messy and, again, have been omitted to save space.

configure (f, P): $F \times 2^M \rightarrow \{success, error\}$

ensure that file f is replicated on each of the servers in the set P.

5. History Table Control Layer

The modified available copy algorithm presented above, requires a highly reliable, consistent history table to be maintained for each file. This layer provides the operations

$$ReadHistory: F \rightarrow V \times T$$

$$WriteHistory: F \times (V \times T) \rightarrow \{success, error\}$$

The history records whether the file has been deleted (the truth value is interpreted as a "deleted" flag) and a set of machine and version number pairs ($V = 2^{M \times Z}$).

The history table control layer supports these operations by replicating the table using a variation of the basic majority voting algorithm so that the file histories are consistent in the face of network partitions. The history tables are made highly available by replicating them on k sites, where $k \gg n$, the number of file copies.

In its simplest form, $k = \lfloor m/2 \rfloor$, where m is the total number of processing nodes. Each node is assigned one read and one write vote, regardless of whether or not it holds a copy of any file's history table. The algorithm will allow a read to succeed even if only one copy of the table is available, so long as the majority of nodes is available. Writes to the table require a majority of nodes to accept the new table version. In the case of random node crashes the method will offer a high degree of read availability. Random network partitions will reduce availability more seriously but the table will still be consistent. More statistical analysis of partitioning is still required.

In order to reduce the communication cost, weighted voting can be used to reduce the numbers of responses required to complete an operation. This will reduce table availablility will unless the nodes with high weights are highly available themselves. Fortunately, the history table information is quite small — eight bytes per copy per file is quite sufficient, so high levels of replication are not costly in terms of storage.

A simple locking scheme is required to ensure that the file state and the history table are kept in step — the table being locked when it is read and unlocked when it is written back. A more subtle scheme is possible, but from our studies of active file stores we believe concurrent update of replicated files is likely to be very rare in practice.

In many distributed systems the number of messages, rather than their size, is the predominant factor in determining the cost of network protocols.[20] Many of the low-level operations required to support this algorithm would benefit from a multicast request response mechanism. If the underlying communication system uses a broadcast link level protocol, the cost of such a mechanism is a factor of the number of *replies* required from a request, not the number of servers to which the request was sent, nor the size of the request parameters. Many studies, including our own, have shown that read operations predominate in most general purpose file systems, and *ReadHistory* is the most costly part of our logical file read request. If the problem of concurrent write operations is ignored (as it very often is in non-replicated file systems), then it is possible to increase the performance of this algorithm by adding a file *open* operation, that caches the file's history locally, writing it back only when a corresponding *close* operation is performed.

6. Analysis and Conclusions

In this section we present a simple combinatorial analysis of this modified available copy method (MACM). We will derive expressions for P(A_u) — the probability that a file is available for update. In this analysis we assume that machines fail independently with a probability p and that update and read requests originate at random from machines not in the set of file servers M. Relaxing these assumptions severely complicates a combinatorial analysis, but we hope that a stochastic process model may provide more realistic formula.

In its simplest form, the available copy algorithm has a read and write availability of $P(A_f) = 1 - (1 - p)^n$ for a file with n replicas. As m, the number of file server nodes, increases the update availability of our MACM approaches this. To show this we must demonstrate that:

a) A_t, the availability of the history table for a file, and A_f, the availability of the file, are asymptotically independent events:

With k table and n file copies chosen from m nodes, the probability that a node holds a copy of both the table and the file is $\frac{k}{m} \times \frac{n}{m}$ which tends to zero as m tends to infinity.

b) The probability that a file is available for update, P(A_u), is asymptotically equal to the probability that the file is available:

For an update to succeed, both the file history and at least one copy of the file must be available. The table, replicated using majority voting, is available with probability

$$P(A_t) = \sum_{k>m/2}^{m} p^k (1-p)^{m-k} \begin{bmatrix} m \\ k \end{bmatrix}$$

which tends to 1 as m tends to infinity.[21,22] Since A_t and A_f are (asymptotically) independent,

$$P(A_u) \approx P(A_f)P(A_t) \approx 1 - (1 - p)^n$$

for large m.

Table 3 shows file availabilities when p=0.7 and p=0.9 for five common replication strategies: unanimous update, single primary, moving primary, majority voting and available copies under the assumptions presented above. Figures for n = 2, 3, 4 and 5 are shown.

method	*n=2*	*n=3*	*n=4*	*n=5*
U Update	*0.49*	*0.34*	*0.24*	*0.16*
S Primary	*0.70*	*0.70*	*0.70*	*0.70*
M Primary	*0.49*	*0.78*	*0.91*	*0.96*
M Voting	*0.49*	*0.78*	*0.65*	*0.83*
A Copies	*0.91*	*0.97*	*0.99*	*0.99*

*Table 3a — Availability when p =*0.7

method	*n=2*	*n=3*	*n=4*	*n=5*
U Update	*0.81*	*0.73*	*0.66*	*0.59*
S Primary	*0.90*	*0.90*	*0.90*	*0.90*
M Primary	*0.81*	*0.97*	*0.95*	*0.99*
M Voting	*0.81*	*0.97*	*0.95*	*0.99*
A Copies	*0.99*	*0.99*	*0.99*	*0.99*

*Table 3b — Availability when p =*0.9

In order to derive a more realistic formula, it is necessary to consider conditional probabilities.

$$P(A_u) = 1 - P(\sim A_f \vee \sim A_t)$$
$$= 1 - P(\sim A_f) - P(\sim A_t) + P(\sim A_f \mid \sim A_t)P(\sim A_t)$$

In the first step, we will show that $P(\sim A_t) = (1-p)^k$ where in the following expression $P(N_i)$ is the probabaility that i nodes are down (not necessarily holding a table copy) and $P(T_i)$ is the probability that all tables are on those i nodes.

$$P(\sim A_t) = \sum_{i=k}^{m} P(N_i)P(T_i)$$
$$= \sum_{i=k}^{m} \frac{(m-k)!}{(m-i)!(i-k)!} p^{m-i}(1-p)^i$$

put $j=i-k$ and $i=j+k$ and we get

$$= (1-p)^k \sum_{j=0}^{m-k} \frac{(m-k)!}{(m-k-j)!j!} p^{m-k-j}(1-p)^j$$

and since the second part is a binomial expansion we can write,

$$P(\sim A_t) = (1-p)^k$$

Now it is simple to show that $P(\sim A_f) = (1-p)^n$. By working from the observation that

$$P(\sim A_f \mid \sim A_t) = \sum_{i=0}^{n} P(S_i)P(R_n)$$

where $P(S_i)$ is the probability that $n-i$ copies are on the servers that hold tables, and $P(R_n)$ is the probability that none of the remaining servers hold available copies, we can derive

$$P(A_u) = 1 - (1-p)^n - (1-p)^k +$$
$$\sum_{i=0}^{n} \frac{k!}{(k-n+i)!} \frac{(m-n+i)!}{m!} (1-p)^{i+k}$$

This formula gives more realistic availability statistics for the modified algorithm. Table 4 shows the update availability computed from this formula for two values of p. A value of m = 50 was used throughout.

	n=2	*n=3*	*n=4*	*n=5*
p=0.7	*0.90*	*0.97*	*0.99*	*0.99*
p=0.9	*0.99*	*0.99*	*0.99*	*0.99*

Table 4 — Update and read availability for MACM.

Conclusions

These figures show an expected availability very close to those obtained from the unmodified available copy method and significantly higher than those available from majority voting unless high replication factors are used. The history table enables the method to give consistent update in the presence of network partition which is not possible with an unmodified available copy approach. Although the history table must be replicated using a voting strategy and requires a high level of replication in order to give the degree of fault tolerance required, it is relatively small compared to the size of the file itself.

We are pursuing this work by simulating this and other algorithms in order to estimate the likely communication overhead involved and to verify the combinatorial analysis above. The simulations will also allow us to investigate file availability in the presence of failures that are harder to study analytically, such as network partition. An analysis based on stochastic processes is also being conducted.

Many areas require further study. In particular there are several interesting systems administration questions that arise only with replicated files. Who may alter the file's replication? How does the user or system administrator specify the replication — explicitly or by asking for a given level of fault tolerance? Should the positioning of files be decided automatically, by users, or by administrators? If the results from the simulation are encouraging, we hope to use a pilot implementation of this work to explore these and other issues.

ACKNOWLEDGEMENTS

We would like to acknowledge the advice and assistance of our collegues Karen Paliwoda, now at the Programming Research Group, Oxford, and Steve Wilbur.

REFERENCES

[1] PA Alsberg, "A Principle for Resilient Sharing of Distributed Resources," *Proceedings of the 2nd International Conference of Software Engineering*, pp. 562-570, October 1976.

[2] PG Selinger, "Replicated Data," in *Distributed Databases - An Advanced Course*, ed. F Poole, Cambridge University Press, 1980.

[3] A El-Abbadi, D Skeen, and F Cristian, "An Efficient, Fault Tolerant Protocol for Replicated Data Management," *Proceedings of the 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pp. 215-229, New York, NY, March 1985.

[4] C Pu, JD Noe, and A Proudfoot, "Regeneration of Replicate Objects: a Technique and Its Eden Implementation," *IEEE Proceedings of the 2nd International Conference on Data Engineering*, February 1986.

[5] RH Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Tran-*

sactions on Database Systems, vol. 4, no. 2, pp. 180-209, June 1979.

[6] D Gifford, ''Weighted Voting For Replicated Data,'' *Proceedings of the 7th ACM Symposium on Operating System Principles*, pp. 150-162, December 1979.

[7] M Herlihy, ''A Quorum-Consensus Replication Method for Abstract Data Types,'' *ACM Transactions on Computer Systems*, vol. 4, no. 1, pp. 32-53, February 1986.

[8] JJ Bloch, D Daniels, and A Spector, ''A Weighted Voting Algorithm for Replicated Directories,'' *Journal of the ACM*, vol. 34, no. 4, pp. 859-909, October 1987.

[9] JF Paris, ''Voting with Witnesses: A Consistency Scheme for Replicated Files,'' *IEEE 6th International Conference on Distributed Computing Systems*, pp. 606-612, 1986.

[10] R Renesse and AS Tanenbaum, ''Voting with Ghosts,'' *IEEE 8th International Conference on Distributed Computing Systems*, pp. 456-461, June 1988.

[11] L Svobodova, ''File Servers for Network-Based Distributed Systems,'' *ACM Computing Surveys*, vol. 16, no. 4, pp. 353-398, December 1984.

[12] H Sturgis, J Mitchell, and J Israel, ''Issues in the Design of a Distributed File System,'' *ACM Operating Systems Review*, vol. 14, no. 3, pp. 55-69, July 1980.

[13] G Popek, B Walker, J Chow, D Edwards, C Kline, G Rudisin, and G Thiel, ''LOCUS: A network Transparent High Reliability Distributed System,'' *Proceedings of the 8th ACM Symposium on Operating System Principles*, pp. 169-177, December 1981.

[14] KH Bennett, ''Distributed Filestores,'' in *Distributed Computing*, ed. FB Chambers, Academic Press, 1984. ISBN 0-12-167350-2

[15] JN Gray, ''Granularity of Locks in a Shared Distributed Database,'' *Proceedings of International Conference on Very Large Databases*, pp. 428-451, September 1975.

[16] AS Tanenbaum and SJ Mullender, ''An Overview of the Amoeba Distributed Operating System,'' *ACM Operating Systems Review*, vol. 15, pp. 51-64, July 1981.

[17] SJ Mullender and AS Tanenbaum, ''A Distributed File Service Based on Optimistic Concurrency Control,'' *ACM Operating Systems Review*, vol. 19, no. 5, pp. 51-62, December 1985.

[18] A Birrell and BJ Nelson, ''Implementing Remote Procedure Calls,'' *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 39-59, February 1984.

[19] S Wilbur and B Bacarisse, ''Building Distributed Systems with Remote Procedure Call,'' *IEE Software Engineering Journal*, vol. 2, no. 5, pp. 148-159, September 1987.

[20] TA Joseph and KP Birman, ''Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems,'' *ACM Transactions on Computer Systems*, vol. 4, no. 1, pp. 54-70, February 1986.

[21] RE Barlow and KD Heidtmann, ''Computing k-out-of-N Reliability,'' *IEEE Transactions on Reliability*, vol. 4, no. 33, pp. 322-323, October 84.

[22] Y C Tay, ''The Reliability of (k, n)-Resilient Distributed Systems,'' *IEEE Proceedings of the 3rd International Conference on Data Engineering*, pp. 119-122, 1984.

Reliability of Replicated Files in Partitioned Networks

S Bek Baydere *B S Bacarisse*

Department of Computer Science
University College London
London WC1E 6BT.

1. Introduction

The potential for increased reliability is often given as one of the benefits of a distributed system. The hardware is, by definition, replicated so hardware and software problems are more likely to cause only a partial failure rather than affect the whole system. As networks grow and evolve, subnets can become bridged together and machines moved from subnet to subnet; more often than not as required by geographical constraints. As a result, it is reasonable to assume that network partitioning is a relatively likely event.

We believe that three central problems must be solved before the benefits of file replication can be realized in general purpose distributed systems:

1. High reliability must be provided with minimum storage cost (say, 2 file copies).
2. The system should provide simple mechanisms to alter the reliability of files as users' requirements change.
3. The reliability should not be adversely affected by changes to the network topology and therefore to the failure modes of the network.

This position paper concentrates on the first and third of these and outlines our work estimating the comparative reliability of files replicated using a variety of strategies under the realistic assumption the network may become partitioned. Of particular interest is the *reliable history* strategy (presented in an earlier paper)[1] which, we believe goes some way towards addressing these three points.

Most of the algorithms used to control updates to replicated data fall into one of two families: voting[2-4] and available copy methods.[5] Available copy algorithms with two copies offer better availability than voting with three; but require all the nodes in the system to have the same understanding of which nodes are available and which are not. This requirement is at odds with the failure model of distributed systems that we outlined above and.

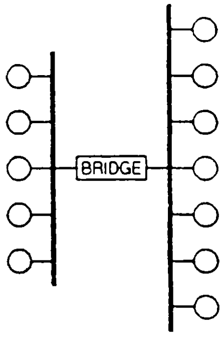
We propose providing a highly available file replication history by recording a version number for each file copy. The table of version numbers is replicated many times and updates to it are controlled using a regenerative-majority consensus voting algorithm.[6] Any client wishing to read a file consults the file's history to determine which file copies are up to date. A read in the file is allowed as long as one up to date copy is available. Write operations proceed in the same way with the update being performed on all the up-to-date copies. Copies that are out of date but whose servers are available are brought up to date by copying.

Because the file's history is small compared to the size of the file itself the high degree of replication involved does not add greatly to the storage cost. It has a communication cost associated with the history access operations but, we believe a multi-cast transport protocol can reduce this cost into acceptable bounds.

The algorithm is a hybrid of voting and available copies techniques but it doesn't require a mechanism to distinguish network failures from node crashes. For obvious reasons we call this the *reliable histories* method.

2. Analysis

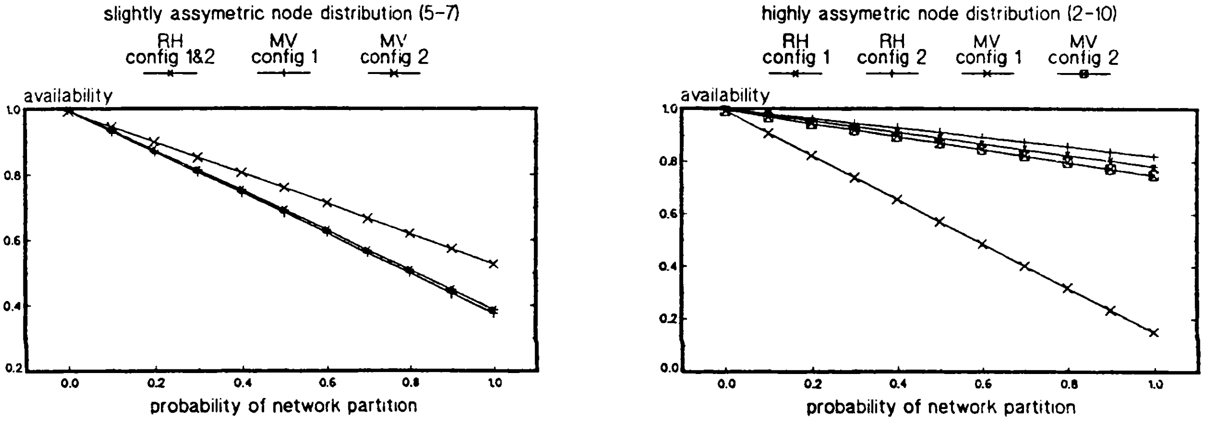
It is a relatively simple matter to analyse the availability offered by various algorithms. We have carried out a combinatorial analysis of the behaviour of majority voting (MV), available copy (AC) and the reliably history (RH) techniques under the assumption the network partitions do not occur.[1] When partitions are possible, the analysis becomes much more involved and only simple network topologies can be studies. For example the simple network shown on the right was studied and the file availability obtained as a function of the availability of the bridge for both MV (three copies) and RH (two copies). A graph of the results is shown below. The two graphs show the effect of allocating different numbers of nodes between the two subnets. In each case, two configurations were assumed. One in which all copies are on the same subnet and the other in which they are split.



Topology 1

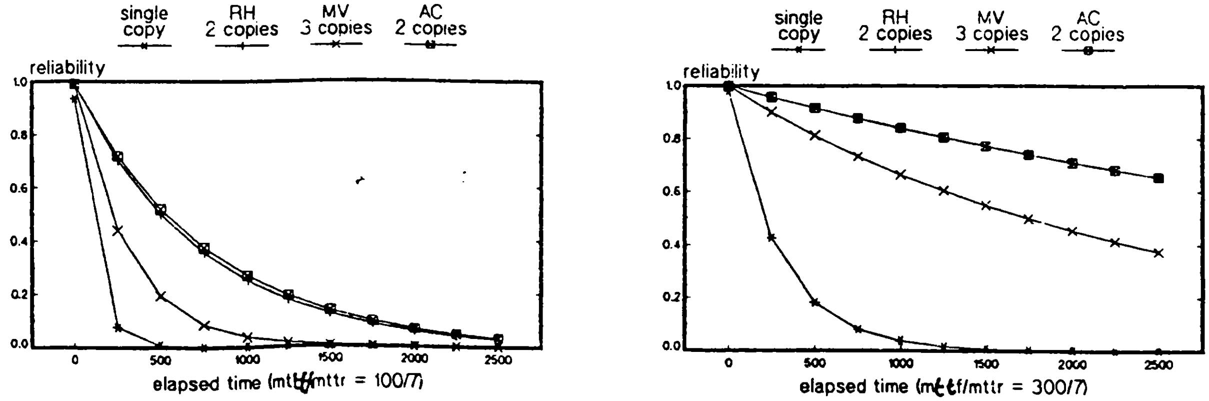
Not surprisingly, MV is much more sensitive to the topology and the placement of copies than RH. We believe this is an important property of the algorithm.

Topology 1: The Effect of Bridge Availability



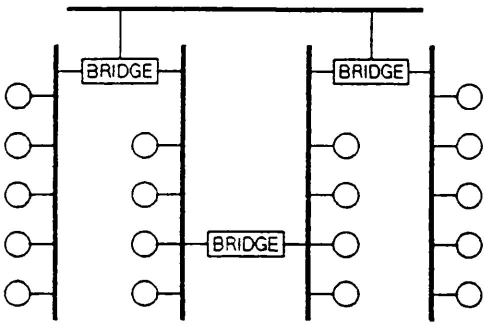
Availability is only one measure of an algorithm's behaviour. Reliability (the probability that the file is continuously available for a given period of time) is probably more important. To study reliability we turned to simulation. The graphs below shows the reliability offered by the MV, AC and RH algorithms in a partition free network of 50 nodes with the reliability of a single copy is given as a reference. An exponential distribution of the mean time between failures was used, whereas the mean time to repair were normally distributed. RH and AV with two copies perform considerably better than MV with three copies.

Reliability in a Partition Free System

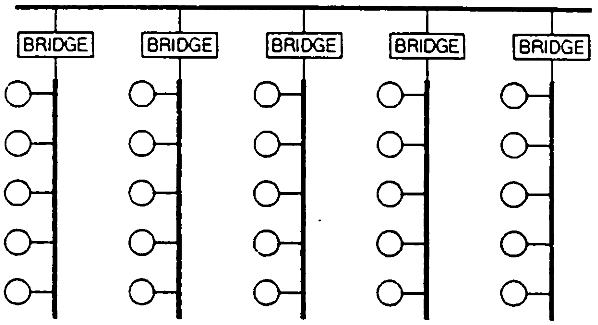


3. Effect of Partitioning on Reliability

The position becomes more interesting when partitioning is considered. A number of network topologies and file configurations have been simulated to study the effect these have on reliability. Since the available copies algorithm is not applicable in this environment, only majority voting is compared with RH. For example, given the two networks show here,



Topology 2



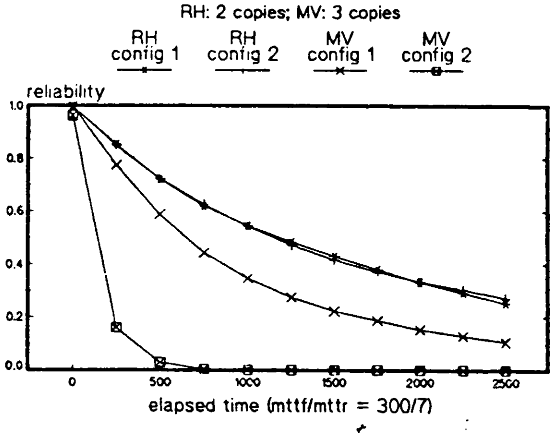
Topology 3

different configurations can be represented by the distribution of the nodes on the links and the link numbers where copies reside. For example, in topology 2, the configuration (10, 15, 5, 12; 2, 4) has 10 nodes on subnet 1, 15 on subnet, etc. The file has two copies: one on subnet 2 and the other on subnet 4.

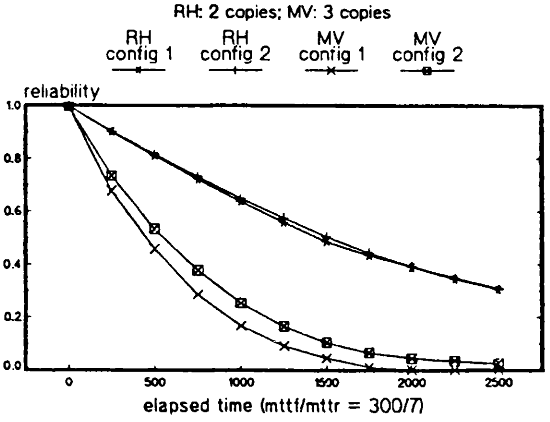
The graphs below compare MV between three copies and RH with two copies in these two network topologies. In each topology, two configurations were used. In the first (given above), all copies were on different links and in the second, all copies were on the same link: (10, 15, 5, 12; 4, 4).

For topology 3, the two configurations were (5, 7, 7, 10, 10; 3, 4, 5) and (5, 7, 7, 10, 10; 5, 5, 5). A failure/repair time ratio of 300/7 was used in all cases.

Topology 2: The Effect of Copy Placement



Topology 3: The Effect of Copy Placement



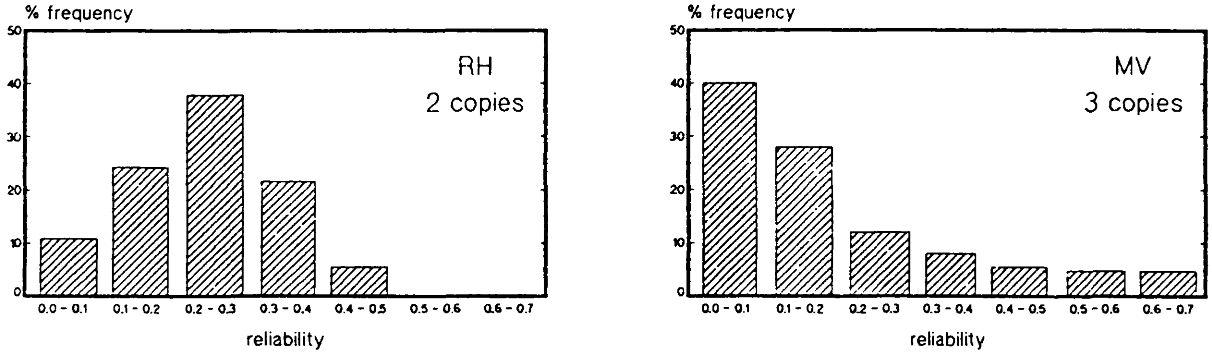
3.1. Resiliency to Network Topologies and Copy Placement

The above results show that there is great variation in the the reliability offered by voting algorithms. In these particular cases the same is not true for RH though in other studies greater variation was observed. In order to learn more, we studied the distribution of reliabilities obtained for several, randomly chosen, configurations. The probability of continuous availability for 1000 time units was used, since there is considerable variation for this value.

The results, shown below, are rather interesting. Majority voting gave exponentially distributed reliabilities with a mean less than 0.1, where the reliable histories algorithm gave more normally distributed reliabilities with a mean of about 0.25.

Distribution of Reliability at 1000 time units

(randomly chosen copy placements)



4. Conclusions

In terms of availability and reliability, the proposed reliable histories algorithm has performs very similarly to available copy methods whilst maintaining consistency in the face of network partition. In many configurations, RH with two copies gives reliability than voting algorithms using three copies (or two copies and one witness). There is also some evidence that the algorithm is less sensitive to the topology of the network and to the placement of file copies. This is likely to be an important property in systems where the configuration can not be planned to maximize file reliability, say, because it is dictated by physical considerations.

The effect of network configuration and file placement needs much further work. One plan of attack is to generate and simulate "random" configurations. While this is relatively simple for copy placement, the notion of a random topology must first be clarified. Perhaps by studying networks in the field some criteria for probable topologies and node distributions can be determined so that suitable configurations can be randomly chosen and simulated. Such a study may make it possible to derive rules that would allow the reliability of a file to be calculated when the topology and location of the copies are given as parameters.

An analytical model for reliability in partitioned systems would certainly help to generalize and verify the results obtained above. Unfortunately this is a difficult task for reliable histories approach because of the number of nodes required before a significant improvement over voting methods can be obtained. We plan to produce pilot implementation to test the practicality of this algorithm and to investigate how multicast protocols could be used to reduce the communication overhead involved in the history table operations.

References

1. B Bacarisse and S Bek Baydere, "A Low Cost File Replication Algorithm," *Proceedings of IEEE COMPCON Spring '89*, pp. 191-196, IEEE Computer Society Press, San Francisco, February 1989.
2. D Gifford, "Weighted Voting For Replicated Data," *Proceedings of the 7th ACM Symposium on Operating System Principles*, pp. 150-162, December 1979.
3. JF Paris, "Voting with Witnesses: A Consistency Scheme for Replicated Files," *IEEE 6th International Conference on Distributed Computing Systems*, pp. 606-612, 1986.
4. R Renesse and AS Tanenbaum, "Voting with Ghosts," *IEEE 8th International Conference on Distributed Computing Systems*, pp. 456-461, June 1988.
5. PA Bernstein and N Goodman, "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases," *ACM Transactions on Database Systems*, vol. 9, no. 4, pp. 596-615, December 1984.
6. RH Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Transactions on Database Systems*, vol. 4, no. 2, pp. 180-209, June 1979.