On Improving the Availability of Replicated Files

Darrell D. E. Long
Jehan-François Pâris

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, San Diego
La Jolla, California 92093

Abstract

To improve the availability and reliability of files the data are often replicated at several sites. A scheme must then be chosen to maintain the consistency of the file contents in the presence of site failures. The most commonly used scheme is *voting*. Voting is popular because it is *simple* and *robust:* voting schemes do not depend on any sophisticated message passing scheme and are unaffected by network partitions.

When network partitions cannot occur, better availabilities and reliabilities can be achieved with the *available copy scheme*. This scheme is somewhat more complex than voting as the recovery algorithm invoked after a failure of all sites has to know which site failed last. We present in this paper a new method aimed at finding this site. It consists of recording those sites which received the most recent update; this information can then be used to determine which site holds the most recent version of the file upon site recovery. Our approach does not require any monitoring of site failures and so has a much lower overhead than other methods.

We also derive, under standard Markovian assumptions, closed-form expressions for the availability of replicated files managed by voting, available copy and a naïve scheme that does not keep track of the last copy to fail.

1. Introduction

To improve the *reliability* and *availability* of files, the data are often replicated at several sites. A *replicated file* is an abstract data object with the same semantics as an ordinary file; by storing copies of the file data at several sites the reliability and the availability of a replicated file object are increased.

The existence of several copies of the file data residing at distinct sites raises the issue of *file consistency*. It is unreasonable to require the users of a replicated file system to be responsible for the consistency of their files; therefore, to insure the consistency of the file contents, a policy for maintaining the replicated data must be selected. Several schemes have been proposed, including: *voting,*[2,3,4,5,6,12] *voting with witnesses,*[9] *dynamic voting*[1] and *available copy.*[8]

Voting schemes insure the consistency of replicated files by honoring read and write requests only when an appropriate quorum of the sites holding copies of the file can be accessed. This method requires that the file data be replicated on at least three sites before there is an improvement in the availability over that of an ordinary unreplicated file.

Other schemes, such as dynamic voting, improve on the availability provided by a simple voting algorithm. Unfortunately, these methods provide this increased availability at the cost of increased network traffic and computation.

In the absence of network partitions, an available copy scheme greatly increases availability at a low cost. When a total failure occurs, a naïve available copy scheme that requires all sites to recover exhibits poor performance because all sites are required to be repaired before the correct state of the file can be ascertained. If we want to improve on this worst-case performance it is important to detect the last site to fail.

This paper is presented in four sections: Section 2 discusses consistency algorithms including voting schemes and the available copy scheme, Section 3 presents our method for detecting the last site to fail and Section 4 presents an analvsis of voting, available copy and naïve available copy with some surprising results.

2. Consistency Algorithms

In its simplest form, *voting*[2,3,4,5,6,12] assumes that the correct state of a replicated file is the state of the majority of its copies. Ascertaining the state of a replicated file requires collecting a quorum of the copies. Should this be prevented by one or more site failures, the file is considered to be unavailable.

This scheme can be refined by introducing different quorums for read and write operations or by allocating different weights, including none, to each copy. Consistency is guaranteed so long as:

1. the write quorum is high enough to disallow simultaneous writes on two disjoint subsets of the copies, and
2. the read quorum is high enough to disallow simultaneous reads and writes on two disjoint subsets of the copies.

These conditions are simple to verify, and account for much of the conceptual simplicity and the robustness of voting schemes. A problem with voting is that it requires at least three copies to improve file availability over that of an ordinary unreplicated file. Since it disallows all file accesses when a majority of the copies are not accessible, it considerably reduces file availability in the presence of site failures. Several novel voting algorithms have been developed to overcome these limitations.

One of the authors has proposed elsewhere[9,10] to replace some copies by *witness copies* that contain all of the state information but none of the user specified data. Like ordinary copies, witnesses vote and maintain a record of the file status. Since they do not contain any information about the contents of the file, they only require a negligible, fixed amount of storage. The maintenance of two copies and one witness is easier than maintaining three copies and incurs less network traffic. Analysis of the availability of a file implemented using witness copies has shown this to be a very useful approach.

Dynamic voting[2] is another algorithm capable of functioning in the presence of network partitions and site failures. Dynamic voting adjusts the necessary quorums of copies required for update or read operations according to the changing states of the network. The scheme improves file availability, but in doing so can incur heavy network traffic.

When network partitions are known to be impossible, *available copy schemes* provide a simple means for maintaining file consistency. Available copy schemes are based on the observation that so long as at least one site has been continuously available it is known to hold the most recent version of the file data.

This work was supported in part by a grant from the NCR Coporation and the University of California MICRO program.

The update rule for an available copy scheme is extremely simple: *write to all available copies.* Since all available copies receive each update, they are kept in a consistent state: data can then be read from *any* available copy. If there is a copy of the file data on the local site, then the read operation can be done locally, avoiding any network traffic.

When a site recovers following a failure, if there is another site which holds the most recent version of the file then the recovering site can repair immediately. A complication arises in the event of total failure: it is not known by the recovering sites which of them hold the most recent version of the file until the last site to fail can be found. In order to speed recovery, it is desirable to ascertain as quickly as possible the last site, or set of sites, that failed.

In the original proposal for the available copy scheme,[8] the goal of detecting the last site to fail was achieved by maintaining several sets of failure information, including: all sites participating in the replication of the file, those sites which have been specifically *included* and those which sites have been specifically *excluded.* An *included* site is one which is known to hold a copy of the most recent version of the file data, an *excluded* site is one which has failed and that failure has been detected by an operational site.

When a site s fails another site t must detect that failure and execute the transaction `exclude(s)`. A failure detection mechanism is assumed to exist and it is assumed to be *foolproof.* When a site t repairs following a failure, it attempts to locate another site s which is operational. If such a site can be found, then t will repair from s and then request s to execute the transaction `include(t)`. In the presence of total failure, the information maintained by the `include` and `exclude` transactions is used to compute the last site to fail; the most recent version of the file data can then be found by examining the version numbers of the sites.

The assumption was made that failures are easily detected and notification of their occurrence can be broadcast to all surviving sites. In general, failures are not easy to detect in a reliable manner. Time-outs are the method usually employed for detecting failures, but they are time consuming and unreliable in the case of a heavily loaded site. An alternative would be to have daemon processes which poll all of the sites to determine their status, but this introduces a window of vulnerability proportional to the time between polls and also increases network traffic.

The scheme that we propose is much simpler in that it only requires the information about site availability to be broadcast each time the file is modified. This leads to a simpler implementation and to decreased network traffic.

3. An Improved Scheme

We present a method which requires only that the availability information be brought up to date when the file is modified or when a repair operation occurs. Our scheme assumes a fixed set of sites connected via a network which is free of partitions.

Definition 3.1. *The* was-available *set, denoted W_s, for a site s is the set of all sites that received the most recent update and all of those sites which have repaired from site s.*

The *was-available* sets represent those sites which received the most recent change to the file data. We posit the existence of an atomic broadcast mechanism; this makes it possible to guarantee that those sites which received the most recent update will appear in the was-available set of each operational site. This condition can be relaxed by ascertaining which sites are operational when the file is first opened and by sending this information along with the first update; the second update will contain the set of sites which received the first update and so forth. By delaying the information in this way, communication costs are minimized at the expense of some increase in recovery time.

Definition 3.2. *Let $S = \{s_1, \ldots, s_n\}$ be the set of sites which hold copies of the file data; then the* closure *of a was-available set W_s, written $C^*(W_s)$, is given by:*

$$C^0(W_s) = W_s$$

$$\ldots$$

$$C^k(W_s) = \bigcup_{t \in W_s} C^{k-1}(W_t)$$

$$C^*(W_s) = \bigcup_{i=0}^{n} C^i(W_s)$$

Definition 3.3. *We say that a site t is a* successor *of a site s if t repaired from s and s subsequently failed.*

For the purposes of this paper, we consider the transitive closure of the successor relation; thus, for a set of sites $S = \{s_1, \ldots, s_n\}$, saying that s_j succeeds s_i means that there can be any number of intermediate successors up to $n - 2$.

Definition 3.4. *A site that has ceased to be operational and that has not been repaired is said to be a* failed *site.*

A *failed* site is one that has ceased to function due to hardware or software failure. We assume *clean* failure; if a site fails it simply halts, malevolent failures are not tolerated. This fail-stop[11] behavior can be simulated by an appropriate software layer and we will not consider it further.

Definition 3.5. *A site that has been repaired but is not yet known to hold the most recent version of the file is said to be* unavailable *or* comatose.

A *comatose* site is one that has been repaired but the current state of the file is not known. Sites enter this state following a total failure and remain there until the most recent version of the file data can be found by examining the version numbers of the other sites.

Definition 3.6. *A site that has been continuously operational or that has been repaired and holds the most recent version of the file is said to be* available.

We propose an algorithm that has a better worst case behavior than a naïve available copy scheme that does not attempt to detect the last site to fail. By modifying the availability information of the sites only when an update occurs we decrease the amount of the network traffic. Our algorithm behaves just as the original available copy algorithm except in the case of total failure: when a site s recovers following total failure it utilizes the availability information by waiting for only those sites in $C^*(W_s)$ to recover. The algorithm is given below as three cases, and more formally in Figure 1.

1. When a site s recovers from a failure it finds that $W_s = \{s\}$. This indicates that s was the last site to fail and can be made immediately available.
2. When a site t recovers from a failure it finds another site s already available. In this case t can repair from s; t is then included in W_s and W_t is set equal to the modified W_s.
3. When a site t recovers from a failure it finds no other sites available. In this case t must wait for all other sites in $C^*(W_t)$ to recover. A site s which holds the most recent version of the file data must be in $C^*(W_t)$. Site t can then repair from site s; t is then included in W_s and W_t is set equal to the modified W_s.

The following theorem establishes the correctness of our update and recovery policies. The proof is divided into four parts, considering the cases of update, site failure and the three cases introduced by the recovery algorithm.

Theorem 3.1. *Following an update operation, a site failure or an application of the recovery algorithm the following invariant holds: $\forall s \in S, C^*(W_s)$ contains the name of a site that holds a copy of the most recent version of the file.*

```
procedure RECOVERY is
begin
  $state(s) \leftarrow comatose$
  select
    when all sites in $C^*(W_s)$ have recovered $\Rightarrow$
      let $t \in C^*(W_s) \ni \forall u \in C^*(W_s), version(t) \geq version(u)$
  or
    when $\exists u \in S \ni state(u) = available$ $\Rightarrow$
      let $t$ be any such available site
  end select
  repair $s$ from $t$
  $W_s \leftarrow W_t \cup \{s\}$
  $W_t \leftarrow W_s$
  $state(s) \leftarrow available$
end RECOVERY
```

Figure 1: Recovery Algorithm

Proof: Assume that the invariant holds. This is obviously true for the initial state where all sites are available and $\forall s \in S, W_s = S$. From this state there are four ways in which the state of the system can be changed: an update can occur, a site can fail, a site can recover and find an active copy already available, or a site s can recover and be required to wait for all sites in $C^*(W_s)$ to recover. We consider each case separately.

1. When an update occurs, all sites which are currently available receive the update. The effect is to make the was-available sets of all available sites consistent. The was-available set of each active site now contains the names of all available sites; the was-available sets of the failed sites remain unchanged.
2. When a site failure occurs, those sites which fail have as their was-available sets the names of the sites which received the most recent update. This is trivially true in the case of total failure since for each site s, $s \in W_s$ and no more updates can occur. If a site s fails then W_s contains the set of sites which received the last update. Let $t \in W_s$ be one of those sites. If t subsequently fails and an update occurs following its demise then some site u which is a successor of t will hold the most recent version of the file data, otherwise t holds the most recent version of the file data.
3. Suppose that when site s recovers there is a copy of the file at site t which is available. The copy at site s will be repaired from the copy at site t according to the recovery algorithm. The statements $W_s \leftarrow W_t \cup \{s\}$ and $W_t \leftarrow W_s$ insure that the invariant holds. If site t fails following the repair, site s or one of its successors will hold the most recent version of the file; in that case the invariant will be preserved due to the transitivity of the C^* relation.
4. Similarly, when all of the sites in $C^*(W_s)$ have recovered the site t that holds the most recent version of the file can be found. The repair is accomplished as in the previous case and the invariant is preserved. It should be noted that when a site s recovers and finds $W_s = \{s\}$ that s can be made available immediately since $C^*(W_s) = \{s\}$.

It should be noted that total failures do not often occur in practice, and when they do it is most often because of some catastrophic event such as a power failure. Following a power failure, it is almost always the case that all of the sites will recover within a few moments of one another.

A naïve available copy scheme does not attempt to detect the last site to fail. Because it does not maintain availability information about sites holding copies of the file data, network traffic is reduced at the cost of introducing poor worst-case behavior. The naïve scheme operates as the previous scheme would if the was-available sets were fixed so that $\forall s \in S, W_s = S$, where S is the set of all sites. The algorithm for such a scheme is given below as two cases, and more formally in Figure 2

```
procedure SIMPLE_RECOVERY is
begin
  $state(s) \leftarrow comatose$
  select
    when all sites have recovered $\Rightarrow$
      let $t \in S \ni \forall u \in S, version(t) \geq version(u)$
  or
    when $\exists u \in S \ni state(u) = available$ $\Rightarrow$
      let $t$ be any such available site
  end select
  repair $s$ from $t$
  $state(s) \leftarrow available$
end SIMPLE_RECOVERY
```

Figure 2: Naïve Recovery Algorithm

1. When a site t recovers from a failure it finds another site s already available. In this case t can repair from s.
2. When a site t recovers from a failure it finds no other sites available. In this case t must wait for all other sites to recover. The site s which holds the most recent version of the file data can then be found by examining the version numbers of all sites.

4. Availability Analysis

In this section, we compare the availabilities of replicated files managed by voting, available copy and a simplified version of the available copy scheme. In all three cases, we assume that the copies of the replicated file reside on distinct *sites* of a computer network. Sites are subject to failures; these failures may either involve the site itself or its communication interface. When a site fails, a repair process is immediately initiated. The repair process will never fail although it may take an arbitrary amount of time before completion. Should several sites fail, the repair process will be performed in parallel on these failed sites. We also assume that the repair process will attempt to bring up to date all the copies that might have become obsolete during the time the site under repair was not operational. Such attempts will not be always successful since they depend on the availability of up-to-date copies of the replicated file.

We assume that individual site failures are independent events distributed according to the Poisson law. In other words, the probability that a given site will experience no failure during a time interval of duration t will be given by $e^{-\lambda t}$ where λ is the *failure rate*. Similarly, we will require that individual site repairs are independent events distributed according to the Poisson law. The probability that a given site will be repaired in less than t time units will be given by $1 - e^{-\mu t}$ where μ is the *repair rate*.

Although the assumption of a constant failure rate λ is usually reasonable, the assumption of exponential repair times is harder to defend on general grounds. However, both assumptions are necessary to represent our system by a Markov process[8] with a reasonable number of states.

The availability A of a system is the limiting value of the probability $p(t)$ that that system will be operating correctly at time t.

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

Since the available copy scheme does not operate correctly in the presence of partitions, we will assume that the communications network linking the several sites where the physical copies of the replicated files reside cannot fail.

4.1. Voting

We will restrict our analysis to the case where all sites containing copies have equal failure rates λ and equal repair rates μ. Under these conditions, it is common to assign equal weights to all the copies of the replicated file. Equal weights cause a particular problem for replicated files with an *even* number of copies. Draw conditions will occur every time an equal number of copies are up and down. To solve these ties, we will have to slightly adjust the weights of the

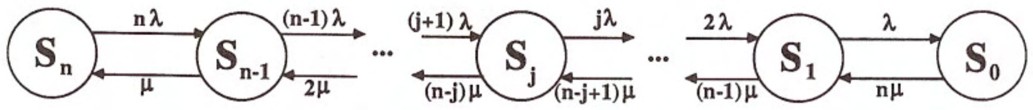

Figure 3: State-Transition-Rate Diagram for Voting

copies in such a way that all ties will be removed without reversing any existing quorum.

The *state* of this file can then be conveniently represented by the current number of copies that are available at any time. Failures and repairs of sites are the only events that can change the state of the system. Since processes resulting in these two events were assumed to be Markovian, the probability of two such events occurring simultaneously is *zero*.

Suppose that the system was initially in the state S_n where all current copies of the file were available. This state will be left for the state S_{n-1} if one of the n sites fails. Since each site failure is an independent event, the total rate at which state S_n will be left is equal to n times the individual failure rate λ of a single site. The state S_{n-1} will in turn be left if either:

1. the only site that was down is repaired and the system returns to the state S_n, or
2. one of the $n-1$ sites that were operational now fails.

The rate at which the first event may occur is the repair rate of a single site μ. The rate at which the second event may occur is equal to $n-1$ times the individual failure rate λ of a single site.

In general, the rate at which a state S_j, with $0 < j < n$, may be left will be given by $j\lambda + (n-j)\mu$, as shown in Figure 3.

The equilibrium conditions for our system are given by

$$\begin{aligned} n\lambda p_n &= \mu p_{n-1} \\ ((n-1)\lambda + \mu)p_{n-1} &= n\lambda p_n + 2\mu p_{n-2} \\ &\cdots \\ (j\lambda + (n-j)\mu)p_j &= (j+1)\lambda p_{j+1} + (n-j+1)\mu p_{j-1} \\ &\cdots \\ n\mu p_0 &= \lambda p_1 \end{aligned}$$

and

$$\sum_{i=1}^{n} p_i = 1,$$

where p_i denotes the probability that the file is in state S_i. One can easily derive from them the equilibrium state probabilities $p_n, \ldots, p_0$, which are given by

$$\begin{aligned} p_n &= \frac{1}{(1+\rho)^n} \\ &\cdots \\ p_j &= \frac{\binom{n}{n-j}\rho^{n-j}}{(1+\rho)^n} \\ &\cdots \\ p_0 &= \frac{\rho^n}{(1+\rho)^n} \end{aligned}$$

where $\rho = \lambda/\mu$ is the ratio of the failure rate over the repair rate. If we have an odd number of copies all with equal weights, the *availability* $A_V(n)$ of the file will be given by

$$A_V(n) = \sum_{j=n}^{\lceil n/2 \rceil} \frac{\binom{n}{n-j}\rho^{n-j}}{(1+\rho)^n} \qquad (n \text{ odd})$$

Should there be an even number of copies, we will adjust the weights of the copies in order to break up the ties. The best that we can do will be to allow access in one half of these ties. The availability of the file will then be given by

$$A_V(n) = \sum_{j=n}^{n/2+1} \frac{\binom{n}{n-j}\rho^{n-j}}{(1+\rho)^n} + \frac{\binom{n}{n/2}\rho^{n/2}}{2(1+\rho)^n} \qquad (n \text{ even})$$

In particular, we have

$$\begin{aligned} A_V(1) = A_V(2) &= \frac{1}{1+\rho} \\ A_V(3) = A_V(4) &= \frac{1+3\rho}{(1+\rho)^3} \\ A_V(5) = A_V(6) &= \frac{1+5\rho+10\rho^2}{(1+\rho)^5} \end{aligned}$$

4.2. Available Copy

Recall that the available copy scheme distinguishes between the copies that have fully recovered from a failure—the *available* copies—and those that have still to be brought up-to-date and remain *unavailable* or *comatose*. Comatose copies are only present after all copies of the replicated file have failed. They can fail and recover like available copies do. Once all the copies of the file have failed, the recovery algorithm will wait until the copy that failed last recovers, mark it as being available and use it to bring all other copies of the file up-to-date.

The state-transition-rate diagram for a replicated file having n copies will have $2n$ states. The first n states labelled from S_1 to S_n will represent the states of the file when 1 to n copies are available; $n-1$ new states labelled from S'_1 to S'_{n-1} will represent the states of the file when all copies of the file have failed and 1 to $n-1$ copies not including the copy that failed last have recovered but remain unavailable. As for voting, state S_0 will represent the situations where all copies of the file have failed and none have yet recovered.

As seen in Figure 4, transitions between states obey the following rules:

1. State S_n has one outgoing transition with rate $n\lambda$ going to state S_{n-1}: this transition corresponds to the failure of one of the n available copies;
2. All states S_j with $j = 1, \ldots, n-1$ have two outgoing transitions: one with rate $(n-j)\mu$ that goes to state S_{j+1} and corresponds to the recovery of one failed copy while the other one with rate $j\lambda$ goes to state S_{j-1} and corresponds to the failure of one of the available copies;
3. State S_0 has two outgoing transitions: one with rate μ that goes to state S_1 and corresponds to the recovery of the copy that failed last while the other one with rate $(n-1)\mu$ goes to state S'_1 this transition corresponds to the recovery of one of the other $n-1$ failed copies;
4. All states S'_j with $j = 1, \ldots, n-2$ have three outgoing transitions: one with rate μ goes to state S_{j+1} and corresponds to the recovery of the copy that failed last, another one with rate $(n-j-1)\mu$ goes to state S'_{j+1} and corresponds to the recovery of one of the other $n-j-1$ failed copies while the last one goes to state S'_{j-1} and corresponds to the failure of one of the unavailable copies;

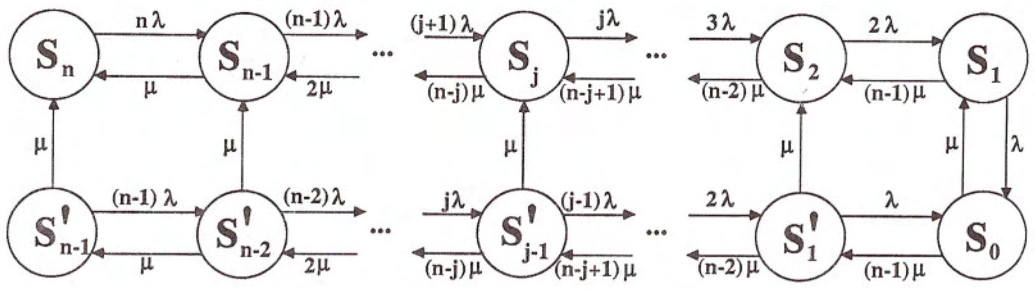

Figure 4: State-Transition-Rate Diagram for Available Copy

5. State S'_{n-1} has one outgoing transition with rate μ going to state S_n and another one with rate $(n-1)\lambda$ going to state S'_{n-2}: the first transition corresponds to the recovery of the last failed copy while the second one corresponds to the failure of one of the unavailable copies.

The equilibrium conditions for our system are given by

$$\begin{aligned}
n\lambda p_n &= \mu(p_{n-1} + p'_{n-1}) \\
((n-1)\lambda + \mu)p_{n-1} &= n\lambda p_n + \mu(2p_{n-2} + p'_{n-2}) \\
((n-1)\lambda + \mu)p'_{n-1} &= \mu p'_{n-2} \\
&\cdots \\
(j\lambda + (n-j)\mu)p_j &= (j+1)\lambda p_{j+1} + \mu((n-j+1)p_{j-1} + p'_{j-1}) \\
(j\lambda + (n-j)\mu)p'_j &= (j+1)\lambda p'_{j+1} + (n-j)\mu p'_{j-1} \\
&\cdots \\
(\lambda + (n-1)\mu)p_1 &= 2\lambda p_2 + \mu p_0 \\
(\lambda + (n-1)\mu)p'_1 &= 2\lambda p'_2 + (n-1)\mu p_0 \\
n\mu p_0 &= \lambda(p_1 + p'_1)
\end{aligned}$$

and

$$\sum_{i=0}^{n} p_i + \sum_{k=1}^{n-1} p'_k = 1,$$

where p_i denotes the probability that the file is in state S_i and p'_k the probability that the file is in state S'_k. One can easily derive from them the availability of the replicated file and which is again given by

$$A_A(n) = \sum_{i=1}^{n} p_i$$

In particular, we have

$$\begin{aligned}
A_A(1) &= \frac{1}{1+\rho} \\
A_A(2) &= \frac{1+3\rho+\rho^2}{(1+\rho)^3} \\
A_A(3) &= \frac{2+9\rho+17\rho^2+11\rho^3+2\rho^4}{(1+\rho)^3(2+3\rho+2\rho^2)} \\
A_A(4) &= \frac{6+37\rho+99\rho^2+152\rho^3+124\rho^4+47\rho^5+6\rho^6}{(1+\rho)^4(6+13\rho+11\rho^2+6\rho^3)}
\end{aligned}$$

where $\rho = \lambda/\mu$. Note that $A_A(1)$ corresponds to the degenerate case of a replicated file which has one single copy and is only included for the sake of completeness.

4.3. Naïve Available Copy

In the naïve available copy algorithm, no record is kept of which copy failed last. Once all the copies of a file have failed, the recovery algorithm will then have to wait until *all* copies of the file have recovered. It will then select the copy with the highest version number, mark it as being available and use it to bring all other copies of the file up-to-date.

As seen in Figure 5, the state-transition-rate diagram for a repli cated file of n copies will still have $2n$ states. As before, n of these states labelled from S_1 to S_n will represent the states of the file when 1 to n copies are available and $n-1$ of these states labelled from S'_1 to S'_{n-1} will represent the states of the file when all copies of the file have failed and 1 to $n-1$ copies have recovered but remain unavailable. State S_0 will continue to represent the situations where all copies of the file have failed and none have yet recovered.

Transitions between states will be quite similar to those observed for the diagram for a conventional available copy algorithm with the exception that there will be no transitions from state S_0 to state S_1 nor any transition from a state S'_j with $j \le n-2$ to an available state. By examining Figure 5, it can be seen that the transitions between states obey the following rules:

1. State S_0 will have now only *one* outgoing transition with rate $n\mu$ that goes to state S'_1: this transition will correspond to the recovery of any of the n failed copies;
2. All states S'_j with $j = 1, \ldots, n-2$ will have only *two* outgoing transitions: one with rate $(n-j)\mu$ will go to state S_{j+1} and will correspond to the recovery of one of the $n-j$ failed copies while the other one will go to state S_{j-1} and will correspond to the failure of one of the unavailable copies.

The equilibrium conditions for our system are given by

$$\begin{aligned}
n\lambda p_n &= \mu(p_{n-1} + p'_{n-1}) \\
((n-1)\lambda + \mu)p_{n-1} &= n\lambda p_n + 2\mu p_{n-2} \\
((n-1)\lambda + \mu)p'_{n-1} &= 2\mu p'_{n-2} \\
&\cdots \\
(j\lambda + (n-j)\mu)p_j &= (j+1)\lambda p_{j+1} + (n-j+1)\mu p_{j-1} \\
(j\lambda + (n-j)\mu)p'_j &= (j+1)\lambda p'_{j+1} + (n-j+1)\mu p'_{j-1} \\
&\cdots \\
(\lambda + (n-1)\mu)p_1 &= 2\lambda p_2 \\
(\lambda + (n-1)\mu)p'_1 &= 2\lambda p'_2 + n\mu p_0 \\
n\mu p_0 &= \lambda(p_1 + p'_1)
\end{aligned}$$

and

$$\sum_{i=0}^{n} p_i + \sum_{k=1}^{n-1} p'_k = 1,$$

where p_i denotes the probability that the file is in state S_i and p'_k the probability that the file is in state S'_k. The availability of the replicated file is again given by

$$A_{NA}(n) = \sum_{i=1}^{n} p_i$$

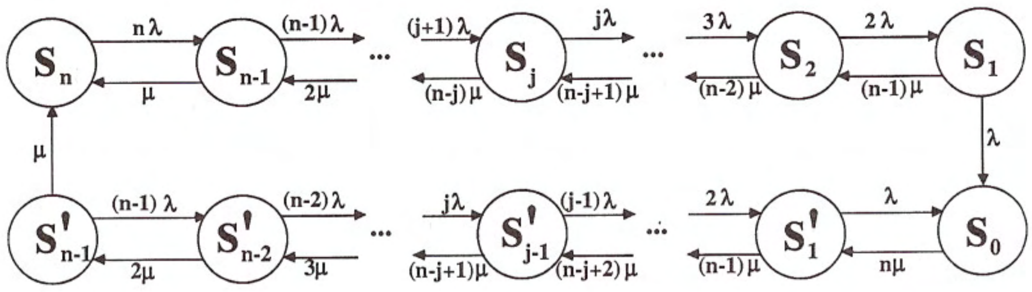

Figure 5: State-Transition-Rate Diagram for Simplified Available Copy

In particular, we have

$$A_{NA}(1) = \frac{1}{1+\rho}$$
$$A_{NA}(2) = \frac{1+3\rho}{(1+\rho)^3}$$
$$A_{NA}(3) = \frac{2+7\rho+11\rho^2}{(1+\rho)^3(2+\rho+2\rho^2)}$$
$$A_{NA}(4) = \frac{3+3\rho+23\rho^2+25\rho^3}{(1+\rho)^5(3-2\rho+3\rho^2)}$$

where $\rho = \lambda/\mu$. Note that $A_{NA}(2) = A_V(3)$, which means that *two* copies managed by our naïve available copy scheme have the same availability as *three* copies managed by a voting algorithm.

4.4. Discussion

The figures 6, 7 and 8 contain the compared availabilities of replicated files with two, three and four copies respectively for the three consistency algorithms we have studied: voting, available copy and naïve available copy. In all three graphs, ρ varies between 0 and 0.20; the first value corresponding to perfectly reliable copies and the latter to copies that are repaired five times faster than they fail and thus have an individual availability of 83.33%.

These graphs call for a few comments: First, they clearly indicate that both the traditional and the naïve available copy schemes produce much higher availabilities than voting; Second, they fail to show any significant difference between the two available copy schemes under investigation for values of ρ less than 0.10.

Most of today's computers are characterized by availabilities well above 0.95 and by values of ρ well below 0.05. Within this range of values of ρ, the availabilities of the two schemes appear to be practically the same, which could lead us to the conclusion that the naïve recovery scheme would perform as well as the conventional algorithm. The real question is how this conclusion could apply to *real* replicated files whose behavior does not always conform to the hypotheses we introduced to build our stochastic model. Observed repair time distributions are characterized by coefficients of variation less than one. Under such conditions, sites will tend to recover in the same order as they failed. The last site to recover after a total failure will often be the last one that failed. When this happens, the conventional available copy scheme will be unable to recover faster than our naïve algorithm as it will have to wait for the last copy to recover in order to get the last copy that failed.

A more important limitation of our model lies in that we have totally discounted the possibility of *irreversible failures,* such as disk head crashes. Files managed by voting schemes can be protected against such failures by requiring that all write quorums include more than one copy. Available copy schemes are clearly more vulnerable since they allow updates as long as one copy remains available. Our naïve scheme will perform even worse than the traditional available

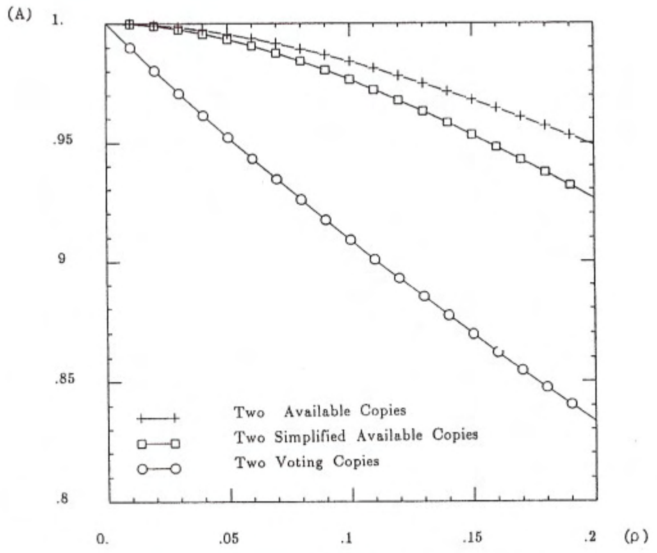

Figure 6: Compared Availabilities for Two Copies

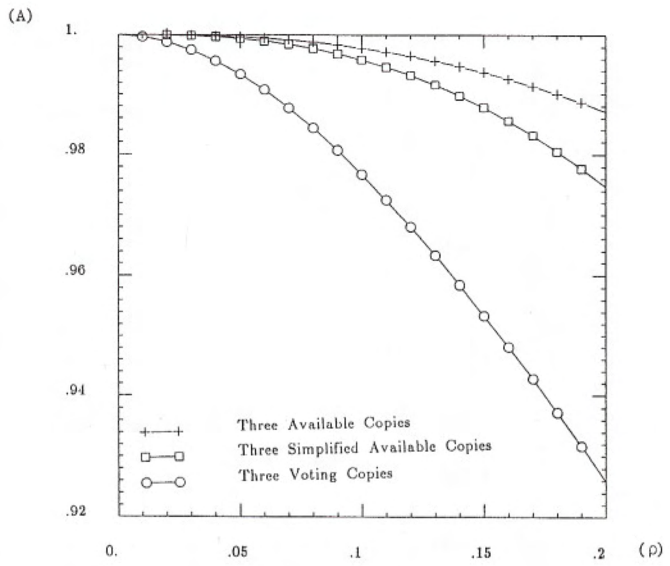

Figure 7: Compared Availabilities for Three Copies

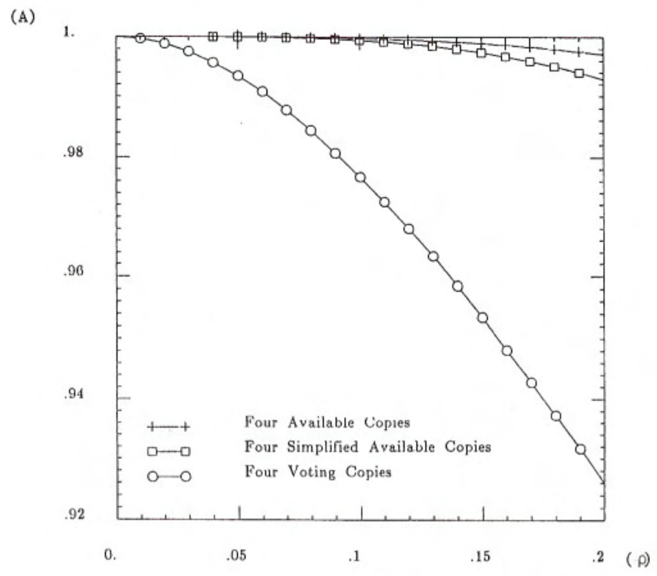

Figure 8: Compared Availabilities for Four Copies

copy algorithm since a manual intervention will be required to recover from any total failure in which one of the failed sites cannot be repaired.

5. Conclusion

In this paper we have presented a new method aimed at improving the performance of replicated files managed by an available copy consistency scheme. Our method consists of recording those sites which received the most recent update; this information can be used to determine which site holds the most recent version of the file upon site recovery. Unlike the original method, our approach does not require any monitoring of site failures and has a much lower overhead.

We have also derived, under standard Markovian assumptions, closed-form expressions for the availability of replicated files managed by voting, available copy and a naïve scheme that does not keep track of the last copy to fail. These data clearly indicate that the two latter schemes perform better than voting.

Acknowledgements

We wish to thank Walter Burkhard, Bruce Martin and all the members of the Gemini group for their help and their encouragement. We are also grateful to Laurette Bradley for her helpful suggestions, and to John Carroll for his help in producing the state diagrams. The first author is especially indebted to Ernestine M^{c}Kinney for her assistance and encouragement.

This work has been done with the aid of MACSYMA, a large symbolic manipulation program developed at the Massachusetts Institute of Technology. MACSYMA is a trademark of Symbolics, Inc.

Bibliography

[1] Davčev, D. and W. A. Burkhard, "Consistency and Recovery Control for Replicated Files," *Proceedings of the ACM Tenth Symposium on Operating System Principles,* (December 1985), 87–96.

[2] Ellis, C. A., "Consistency and Correctness of Duplicate Database Systems," *Operating Systems Review,* 11, 1977.

[3] Ellis, C. S., R. A. Floyd, "The Roe File Systems," *Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems,* 1983.

[4] Garcia-Molina, H., "Elections in a Distributed Computer Systems," *IEEE Transactions on Computers,* C-31, 1982, 48–59.

[5] Garcia-Molina, H. and D. Barbara, "Optimizing the Reliability Provided by Voting Mechanisms," *Proceedings of the Fourth International Conference on Distributed Computing Systems,* San Francisco, California (May 1984), 340–346.

[6] Gifford, D. K. "Weighted Voting for Replicated Data," *Proceedings of the Seventh ACM Symposium on Operating System Principles,* Pacific Grove, California, (December 1979), 150–161.

[7] Gnedenko, B. V., *Mathematical Methods in Reliability Theory,* Moscow, English Translation, New York, Academic Press, 1968.

[8] Goodman, N., D. Skeen, A. Chan, U. Dayal, R. Fox and D. Ries, "A Recovery Algorithm for a Distributed Database System," *Proceedings of the Second ACM Symposium on Principles of Database Systems,* Atlanta, Georgia (March 1983), 8–15.

[9] Pâris, J.-F., "Voting with Witnesses: A Consistency Scheme for Replicated Files," *Proceedings of the Sixth International Conference on Distributed Computing Systems,* Cambridge, Massachusetts (May 1986), 606–612.

[10] Pâris, J.-F., "Voting with a Variable Number of Copies," *Proceedings of the Sixteenth Fault-Tolerant Computing Symposium,* Vienna, Austria, (July 1986), 50–55.

[11] Schlichting, R. D. and F. B. Schneider, "Fail Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Transactions on Computer Systems,* 1983, 222–238.

[12] Thomas, R. H., "A Majority Consensus Approach to Concurrency Control," *ACM Transactions on Database Systems* 4, 1979, 180–209.