# A Hybrid Replica Control Algorithm Combining Static and Dynamic Voting

SUSHIL JAJODIA, SENIOR MEMBER, IEEE, AND DAVID MUTCHLER

***Abstract*—There are several replica control algorithms for managing replicated files in the face of network partitioning due to site or communication link failures. In this paper, we propose a hybrid scheme that integrates the static voting protocol and dynamic voting with linearly ordered copies. We use a stochastic model to compare the file availability afforded by the proposed hybrid scheme against the availabilities of voting, dynamic voting, and dynamic voting with linearly ordered copies. The hybrid scheme has the most availability of these four algorithms for all reasonable repair/failure ratios tested.**

***Index Terms*—Availability, consistency, distributed file management, distributed systems, dynamic-linear, dynamic voting, network partitioning, pessimistic algorithms, replica control, serializability.**

## I. Introduction

There are several replica control algorithms for managing replicated data in the face of network partitioning due to site or communication link failures. (See [14] for an excellent survey of several such algorithms.) *Voting* [19], [32], [35] is the best known example of such a scheme. More recently, researchers have introduced algorithms that permit a dynamic reassignment of votes [6], [7], [13], [15], [16], [20], [21]–[24]. Of these, *dynamic voting with linearly ordered copies* [22], hereafter referred to as *dynamic-linear*, is particularly attractive. It shares two virtues with ordinary voting: it has a simple statement that permits a clear correctness proof, and it is easy to implement. In addition, it has excellent availability.

It was shown in [22] (see also [24]) that the file availability afforded by dynamic-linear is greater than the availability of voting if the file is replicated at four or more sites in the network, but ordinary voting is superior if the number of sites that have copies is exactly three. The present work introduces a *hybrid* algorithm. The algorithm is of interest for three reasons. First, it shows how one can merge static and dynamic voting algorithms. Second, it has greater availability than voting, dynamic voting, and dynamic-linear for all reasonable repair/failure ratios tested. Third, it suggests an extension that is the leading candidate for the *optimal* algorithm in the context of our homogeneous, stochastic model.

Manuscript received March 1, 1989; revised July 14, 1989.

S. Jajodia is with the Department of Information Systems and Systems Engineering, George Mason University, Fairfax, VA 22030-4444.
D. Mutchler is with the Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301.

IEEE Log Number 8931644.

In Section II, we state the problem more precisely. Section III presents informal descriptions of voting, dynamic voting, dynamic-linear, and our new hybrid algorithm. Section IV illustrates the hybrid algorithm by an example. In Section V, we give a careful description of the hybrid algorithm, with all the details of the protocol spelled out. Section VI contains a stochastic analysis of the availability of the hybrid algorithm.

In sum, this paper makes three original contributions. First, it introduces a hybrid algorithm that combines static and dynamic voting. Second the paper gives the details of the protocol by which the algorithm operates;[1] these details apply equally well to our other dynamic algorithms [21], [22]. Third, the availability of the hybrid algorithm is computed using a stochastic model. The results of this analysis show that for all reasonable repair/failure ratios, the hybrid algorithm has greater availability than dynamic-linear, which in turn has been shown to have greater availability than any static algorithm [24], in the context of our model.

## II. Specification of the Problem

The distributed database (DDB) system consists of a collection of independent computers, called *nodes* or *sites*, connected via communication links. We assume that site failures are clean, i.e., nodes stop executing without performing any incorrect actions and that node crashes are detectable by other nodes [31]. We do not include Byzantine failures [30] where sites may act in an arbitrary and malicious manner. Messages may be lost or delivered out of order but any modifications to messages while in transit are presumed to be detectable. Site or communication failures may separate the sites into more than one connected component of communicating sites. We call each connected component a *partition*.

There are several logical files in the DDB, and a physical copy of each logical file is stored at one or more sites. We assume that all sites run a *concurrency control protocol* which ensures that the execution of all transactions at any site is serializable [8], [25]. A *replica control protocol* ensures that the replicated data are managed correctly in the presence of failures. (An excellent survey of several of these strategies is given in [14].) In a *pessimistic* replica control protocol, mutual consistency of a rep-

[1]These details also appear in [24].

licated file is maintained by making sure that all reads and writes are made to the current copy and that files are updated in at most one partition at any given time. We will call such a partition the *distinguished* partition. For example, in simple voting [19], [32], [35], the distinguished partition is the partition, if any, that contains more than half of the sites. Different pessimistic protocols use different rules to define the distinguished partition. When the recovery of sites or communication links causes partitions to unite, those nodes whose copies are out-of-date may obtain missing updates from other sites in the partition. (Thus, there may exist brief times after a recovery when the copies at the various sites in a partition are not identical.) If there does not exist a distinguished partition, all sites in the system must wait until enough sites and communication links are repaired so that there is once again a distinguished partition in the system. Since this wait is unavoidable [34], the challenge is to come up with a pessimistic replica control algorithm which not only preserves mutual consistency of various copies of a file, but at the same time achieves high availability.

## III. Informal Description of the Algorithms

Since our protocol does not depend on the number of files that are replicated, we assume for ease of exposition that there is a single file $f$ which is replicated at $n$ sites $S_1, S_2, \cdots, S_n$.[2] In this section, we present the algorithms somewhat informally. The next section contains an example, after which we give a very precise explication of the hybrid algorithm.

For voting [19], [32], [35] in its simplest form, the distinguished partition is the partition, if any, that contains more than half of the sites. Clearly, this meets the second requirement of a pessimistic algorithm: there cannot exist more than a single such partition at any instant. The first requirement (fresh reads and writes) is met by maintaining a version number with each copy of the file.

Ordinary voting is a *static* algorithm, in that all possible distinguished partitions can be listed in advance [5], [18], [26]. Recently, several researchers have suggested *dynamic* algorithms that dynamically change the rules for defining the distinguished partition [6], [7], [13], [16], [17], [20]–[23], [27]–[29]. These algorithms use different mechanisms to accomplish similar ends. For example, the Davcev–Burkhard algorithm [13] requires that each site maintain a connection vector that continuously records the list of sites to which that site can talk. Barbara, Garcia-Molina, and Spauster [6], [7] present protocols for autonomous reassignment of votes.

Our own dynamic algorithm [21] (which we call simply "dynamic voting") operates by associating with each copy of the file $f$ two integer variables: *version number* and *update sites cardinality*. The update sites cardinality of copy $f_i$ is maintained in such a way that it is always the number of sites that participated in the most recent update to $f_i$. Thus, if the copy at site $A$ has version number 12 and update sites cardinality 7, then seven sites participated in the update to version 12. The distinguished partition is defined to be the partition, if any, that contains more than half of the *up-to-date* copies of the file. The details of this protocol can be distilled from the details we present for the hybrid algorithm in Section V. But the basic operation of dynamic voting is simple. A partition $P$ determines whether it is the distinguished partition by first determining the biggest (most recent) version number $VN$ among the copies in partition $P$, and the update sites cardinality $SC$ of the copies with that version number. Partition $P$ is the distinguished partition if and only if it contains more than half of the $SC$ sites with version number $VN$.

The hybrid algorithm we shall shortly describe merges ordinary voting with dynamic-linear [22]. Dynamic-linear extends dynamic voting by associating with each copy of the file a third variable, called the *distinguished site*. When there is an even number (say $SC$) of sites participating in an update (say to version $VN$), each site sets its distinguished site entry to name one of the participating sites. (They can select this site by any mechanism desired, but all the participating sites must select the same site. For example, one might order the sites linearly and select whichever participating site is biggest according to the linear order.) Suppose a subsequent partition finds that version number $VN$ is the most current version among its copies. If this partition contains exactly $SC/2$ sites with version $VN$, dynamic voting cannot accept the update, for nothing would then prohibit the other half of the $SC$ sites from at the same time doing an update themselves, if they too were grouped in a single partition. Dynamic-linear accepts an update when dynamic voting does, plus also in the case when the partition contains exactly $SC/2$ sites *including the distinguished site*. That is, the distinguished site is used to "break the tie" when a partition contains exactly half of the sites with the up-to-date version of the file.

The *hybrid* algorithm acts just like dynamic-linear, except as follows.

1) When a three-site partition makes an update, the distinguished site entry is expanded to *list* the three sites. That is, each copy of the file has associated with it a *distinguished sites list* (containing the names of exactly three sites) instead of the (single) distinguished site. This enables the switch from dynamic modification of the distinguished partition to a static, three-site distinguished partition.

2) When a partition $P$ wishes to determine whether it is the distinguished partition, it determines as usual a site $X$ that contains the biggest version number in partition $P$. Suppose the update sites cardinality of site $X$ is 3. In this special case, partition $P$ is the distinguished partition if and only if it contains two of the three sites on the distinguished sites list of site $X$. Furthermore, if partition $P$ contains only those two sites, the hybrid algorithm does *not* modify the distinguished partition by changing the up-

[2]Our work generalizes to the setting where transactions may update two or more files. Any such transaction $T$ will require a distinguished partition for *every* file in its read and write set.

date sites cardinality. Only the version number is changed; the hybrid algorithm is in its static phase. However, if partition $P$ contains sites in excess of the required two sites, the algorithm reverts to a mimic of dynamic-linear and dynamically installs $P$ as the new distinguished partition.

The next section presents an example that shows how the hybrid algorithm works. After that we give all the gory details of the protocol.

## IV. An Example of the Hybrid Algorithm

In this example, we assume that the distributed database consists of a single file $f$ which has been replicated at five sites A, B, C, D, and E. These sites are initially connected and form a single partition. Suppose the file $f$ has been updated nine times, so the initial state can be represented as follows where the symbol '—' means that we do not care about the actual value of this variable.

| | A | B | C | D | E |
|---|---|---|---|---|---|
| *VN*: | 9 | 9 | 9 | 9 | 9 |
| *SC*: | 5 | 5 | 5 | 5 | 5 |
| *DS*: | — | — | — | — | — |

At this point suppose site A receives an update, and it finds that it can communicate with sites B and C only. Site A determines that the most current version in its partition is version 9, that the update sites cardinality associated with that version is 5, and that site A's partition contains three of the five copies with version 9. Site A concludes that it belongs to the distinguished partition and processes the update. Since there are exactly three sites involved in this update, the distinguished site ($DS$) variables at those participating sites are set to contain their names, in this case ABC. The site cardinalities of the participating sites are each set to 3, since three sites perform the update to version 10. The state then changes to

| | A | B | C | D | E |
|---|---|---|---|---|---|
| *VN*: | 10 | 10 | 10 | 9 | 9 |
| *SC*: | 3 | 3 | 3 | 5 | 5 |
| *DS*: | ABC | ABC | ABC | — | — |

Suppose now that site A receives yet another update, and it discovers that it can communicate with site C only. The most current version in its partition is version 10, with associated update sites cardinality 3. The novelty here is that since the update sites cardinality is 3, ordinary voting is used to perform the update. Since sites A and C together are two of the three sites on the distinguished sites list, A can process the update. The database state changes to

| | A | C | B | D | E |
|---|---|---|---|---|---|
| *VN*: | 11 | 11 | 10 | 9 | 9 |
| *SC*: | 3 | 3 | 3 | 5 | 5 |
| *DS*: | ABC | ABC | ABC | — | — |

Note that sites A and C do *not* change their update sites cardinality to 2 (as they would under dynamic voting or dynamic-linear), even though only two sites participated in the update. The hybrid algorithm is in its static phase: the set of potential distinguished partitions is fixed (for the moment) to contain exactly AB, BC, AC, and their supersets. Continuing the example, suppose that now site D receives an update. Suppose it finds that it can communicate with sites B, C, and E. The most current version in this partition is version 11, with associated update sites cardinality 3, so a majority from the three sites on the distinguished sites list is sought. Sites B and C form the desired majority, so the update can take place. (Note that neither dynamic voting nor dynamic-linear would permit this update.) Since the number of sites in the partition (four) is more than two, the algorithm reenters its dynamic phase and modifies the $SC_i$ entries of the participating sites. Since there is an even number of sites in the distinguished partition, the value of the $DS_i$ variables will be modified as well. Supposing that the sites are ordered in lexicographic order with respect to the file $f$, and that the distinguished site is selected according to the linear order, $DS_i$ will be set to B at the four sites participating in the update.

| | A | B | C | D | E |
|---|---|---|---|---|---|
| *VN*: | 11 | 12 | 12 | 12 | 12 |
| *SC*: | 3 | 4 | 4 | 4 | 4 |
| *DS*: | ABC | B | B | B | B |

Finally, suppose that the site E were to get an update and finds that it can communicate with site B only. Sites B and E together form a distinguished partition because they contain exactly half of the sites with most-current version number, plus they contain the distinguished site (B). Sites B and E perform the update, leading to the following situation:

| | A | B | E | C | D |
|---|---|---|---|---|---|
| *VN*: | 11 | 13 | 13 | 12 | 12 |
| *SC*: | 3 | 2 | 2 | 4 | 4 |
| *DS*: | ABC | B | B | B | B |

## V. The Hybrid Protocol in Detail

### A. Data Structures

We assign an *a priori* total ordering to the sites, denoted by $>$, for the file $f$. (Different files may be replicated at different groups of sites, and sites in each group may be assigned different total orderings.)

We associate with each copy of the file $f$ three variables: version number, update sites cardinality, and distinguished sites list, defined as follows.

*Definition 1:* The *version number* of a copy $f_i$ at site $S_i$ is an integer $VN_i$ which counts the number of successful updates to $f_i$. We set $VN_i$ to zero initially and increment it by one each time an update to $f_i$ occurs.

*Definition 2:* Associated with each copy $f_i$ at site $S_i$ is another integer called the *update sites cardinality*, denoted by $SC_i$, which (almost) always reflects the number of sites participating in the most recent update to $f_i$. We let $SC_i = n$ (number of sites) initially, and whenever an update is made to $f_i$, then $SC_i$ is set to the total number of copies which were updated during that update. The hybrid nature of our algorithm creates a single exception to this rule, however. See the procedure **Do_Update** below.

*Definition 3:* We associate with each copy $f_i$ at site $S_i$ a variable called *distinguished sites list*, denoted by $DS_i$. The value of a $DS_i$ is important if and only if the corresponding $SC_i$ is even or is equal to 3. When $SC_i$ is even, $DS_i$ identifies the site which is greater (in the linear ordering for the file $f$) than all other sites that participated in the most recent update to $f_i$. When $SC_i = 3$, $DS_i$ lists those three sites from which a majority is neeed to form a distinguished partition.

### B. Protocol under Normal Operation

In this section, we describe our three-phase protocol, assuming first that no site or communication link failures take place during the execution of these phases; we shall describe in the next section how our scheme adjusts in the face of failures.

Suppose a site $S$ receives an update to the file $f$. $S$ initiates execution of the protocol consisting of the following three phases.[3]

i) $S$ issues a LOCK_REQUEST to its local lock manager. When the lock request is granted, $S$ reads its copy of the file $f$, and performs the tentative update.

ii) Next, $S$ sends a VOTE_REQUEST message to all the sites.

iii) When a site $S_i$ receives a VOTE_REQUEST, it issues a LOCK_REQUEST to its local lock manager. When the lock request is granted, $S_i$ sends to $S$ the values $VN_i$, $SC_i$, and $DS_i$.

iv) The site $S$ collects the responses from the various sites and decides if it is a member of a distinguished partition (see routine **Is_Distinguished** below). Henceforth, we refer to $S$ as the *coordinator* and the respondents as *subordinates*.

v) If $S$ does not belong to a distinguished partition, it aborts the update, issues a RELEASE_LOCK request to its local lock manager, and sends ABORT messages to all the participants. When a participant receives the ABORT message, it too issues a RELEASE_LOCK message to its local lock manager. The remaining steps do *not* apply if step v) is invoked.

vi) If $S$ does belong to a distinguished partition, it determines if its copy is current; if it is not current, $S$ determines what subordinates have current copies of the file $f$. (See routine **Catch_Up** below.) If the copy at site $S$ is current, $S$ proceeds with the next step. Otherwise, $S$ acquires the missing updates from any subordinate that has a current copy, and then continues with the next step.

vii) Once the copy at $S$ is current, $S$ commits the update to the file $f$ together with the modification to its $VN_i$, $SC_i$, and $DS_i$ and sends to each $S_i$ the COMMIT message along with the missing updates (if necessary), the new update to $f$, and the new values for $VN_i$, $SC_i$, and $DS_i$. Moreover, $S$ issues a RELEASE_LOCK request to its lock manager. See the procedure **Do_Update** below.

viii) Each $S_i$ acts on the message received from $S$, and then issues a RELEASE_LOCK request to its lock manager.

The three phases of our protocol are: the voting phase [steps ii)–iv)], the catch-up phase [step vi)], and the commit phase [steps v), vii), and viii)]. Note that the catch-up phase is not necessary if the copy at $S$ is current.

We now describe in more detail each of the three phases in turn. To initiate the voting phase, the site $S$ executes the **Is_Distinguished** procedure described below.

**Is_Distinguished:**

1) The site $S$ asks all sites that have a copy of $f$ to send their values $VN_i$, $SC_i$, and $DS_i$. Let $P$ denote the set consisting of the coordinator $S$ and all the subordinates. Each site in $P$ locks its copy of the file $f$.

2) The site $S$ then calculates

$$\text{the value } M = \max \{VN_i : S_i \in P\},$$

$$\text{and the set } I = \{S_j \in P : VN_j = M\}.$$

$M$ denotes the largest version number which is in $P$, and the set $I$ consists of those sites in $P$ which have the version number $M$. $S$ then takes the update sites cardinality of *any* site in the set $I$. Denote this by $N$.

3) If card $(I)$[4] $> N/2$, then $S$ is a member of a distinguished partition.

4) Otherwise, if card$(I) = N/2$, then select any site $S_i$ in $I$. (Any site in $I$ will do; the choice is arbitrary.) If $DS_i \in I$, then $S$ belongs to a distinguished partition.

5) Otherwise, if $N = 3$, then the site $S$ examines the value of the distinguished sites list $DS_j$ of the site in $I$. (Since case 3) did not apply, there must be only a single site in $I$.) If $P$ contains two or all three of the sites listed in $DS_j$, then also does $S$ lie in a distinguished partition. (Note that we do *not* require that these sites be in $I$, but only that they be in $P$.)

6) Otherwise, $S$ does not belong to a distinguished partition. In this case, the site $S$ aborts the update, sends ABORT messages to the subordinates, and releases the lock on its copy. Upon receiving the abort message, the subordinates release all locks on their copies.

As we mentioned, a site $S$ initiates the second phase if it has determined that it belongs to a distinguished partition. We continue to use the same notation as in the **Is_Distinguished** procedure.

**Catch_Up:**

All the sites in the set $I$ possess the current copy of the file $f$, and the sites in the set $P - I$ have copies that are old. If the site $S$ is not a member of the set $I$, it requests the missing updates from some site in $I$.

[3]In the algorithm, we have chosen to use locking instead of time stamps. It is easy to come up with a time stamp analog also.

[4]*Notation:* For a set $X$, card$(X)$ denotes its cardinality.

We note that at the end of this phase, the copy of $f$ at the site $S$ is current. Once this is done, the third and the final phase commences.

**Do_Update:**

I) During this phase, the sites commit the update. The coordinator $S$ commits the update to the file $f$ and the new values for the three variables associated with each copy

$$VN_i = M + 1$$

$$SC_i = \text{card}(P)$$

$$DS_i = \begin{cases} S' & \text{if card}(P) \text{ is even and } S' > S'' \\ & \quad \text{for all other } S'' \text{ in } P \\ P & \text{if card}(P) = 3 \end{cases}$$

*except* if $N = 3$ and $\text{card}(P) = 2$,
then there is no change made to $SC_i$ and $DS_i$.

The site $S$ sends to each subordinate the following: the COMMIT message, the new update to the file, and the new values for $VN_i$, $SC_i$, and $DS_i$. Also, $S$ sends the missing updates to each subordinate in $P - I$. The site $S$ then releases the lock on its copy of the file.

II) Each subordinate upon receiving the message from $S$ commits the update and unlocks its copy of the file.

Each time an update succeeds at a site, it commits the update together with the three values. Thus, an update operation at a site is atomic in the sense that either all four operations—updates to the file, version number, update sites cardinality, and distinguished sites list—are performed in entirety or are not performed at all.

We should note that our protocol may cause deadlocks to occur. There are several ways to handle deadlocks, and we refer the reader to [9] or [10] for additional details.

### C. Protocol under Failures

In this section, we describe how our protocol adjusts to site and communication link failures. We will describe a *restart protocol* which is executed by a site when it recovers from a failure.

Our restart protocol works as follows. When a site recovers from failure, it takes steps to ensure that its copy is brought up-to-date. We next describe how a site $S$ can do this.

**Make_Current:**

a) The site $S$ locks its local copy and initiates execution of steps 1)-5) of the procedure **Is_Distinguished** to determine if it is a member of a distinguished partition. The value $M$ and the sets $I$ and $P$ are as described therein. Note that the subordinates in $P$ lock their copies as well.

b) If $S$ is not in distinguished partition, $S$ sends ABORT messages to all the subordinates in $P$ and releases the lock on its local copy. ($S$ cannot request missing updates from anyone; it may try again at a later time.) The subordinates upon receiving the ABORT message unlock their copies.

c) If $S$ does belong to a distinguished partition and the version number of the copy at $S$ is equal to $M$, then the copy at $S$ is current, and so $S$ may skip the next step (the catch_up phase).

d) If $S$ belongs to a distinguished partition and the version number of the copy at $S$ is less than $M$, then it requests the missing updates from any subordinate in the set $I$.

e) Finally, $S$ sends to all sites in $P$ the COMMIT message, the missing updates (if necessary), the new version number, the new update sites cardinality, and the new value for the distinguished sites list. All of these are computed exactly as in the **Do_Update** routine. Once this is done, $S$ unlocks its local copy.

f) The subordinates upon receiving the message from $S$ commit the updates and unlock their copies.

We should note that whenever the **Make_Current** procedure permits an old copy to catch up, we treat this operation like an update in that we increment the version numbers of the participating copies by one.

It is straightforward to formulate a *termination protocol* which is invoked to correctly terminate transactions when the three-phase update procedure is interrupted by failures. See, for example, [9, ch. 7], [10, sect. 9.2.2.2], or [24].

*Theorem 1:* The hybrid algorithm is correct, i.e., maintains the consistency of the replicated file.

*Proof:* To show correctness of a replica control algorithm, one must show that it produces one-copy serializable logs [3], [9]. However, every pessimistic algorithm with all reads from the current copy produces such logs [23], so it suffices to show that the hybrid algorithm is pessimistic and that reads are from the current copy. That is, we must show that the hybrid algorithm: permits at most one distinguished partition at any one time; keeps all copies within a distinguished partition identical; and ensures that any two consecutive distinguished partitions (say, from one containing version $M$ to another containing version $M + 1$) have a copy in common.

First note that the two-phase commit protocol which our algorithms embed ensures that the coordinator and its subordinates act in consensus. This keeps all copies within a distinguished partition identical, and permits us to focus on the coordinator in the rest of the proof. One must check that the termination protocols we suggest do not invalidate the coordination provided by two-phase commit; we leave this to the reader.

Consider an update from version $M$ to version $M + 1$. First note that all sites with version number $M$ will share the same update sites cardinality and distinguished sites entry/list (see the **Do_Update** routine). Let $SC$ and $DS$ denote the values of these variables, for version $M$, and let $I$ denote the set of version $M$ sites participating in the update to version $M + 1$. The update from version $M$ to version $M + 1$ is permitted only if one of the following conditions is met (see the **Is_Distinguished** routine):

- $\text{card}(I) > SC/2$;
- $\text{card}(I) = SC/2$ and $DS \in I$;
- $SC = 3$ and two or more of the three sites listed in $DS$ are participating in this update.

After the successful update, the participating sites change their version number to $M + 1$. None of the three conditions above will hold thereafter, no matter which of the remaining sites with version number $M$ join together. That is, the conditions needed for a second update from version $M$ cannot occur. We conclude that two updates in different partitions cannot be committed from version $M$. Thus, the file versions form a sequence. Any two successive versions will have at least one copy in common: see the **Catch_Up** routine. This suffices to show that the hybrid algorithm is a pessimistic algorithm with all reads from the current copy,[5] and hence is correct. □

## VI. Availability of the Hybrid Algorithm

### A. *How Should One Evaluate the Hybrid Algorithm?*

A sequence of failures, repairs, and update requests can occur in such a way that the hybrid algorithm can accommodate a request when other pessimistic algorithms cannot, and vice versa. To see this, consider the *partition graph* in Fig. 1. It shows a five-node network that fragments (via link failures) into two partitions ($ABC$ and $DE$) at time 1; partition $ABC$ further fragments into $AB$ and $C$ at time 2; site $C$ joins $DE$ at time 3, etc. Suppose that at least one update arrives at each partition shortly after each partition change. Consider the behavior of voting, dynamic voting, dynamic-linear, and the hybrid algorithm on the network.

At time 1, all four algorithms permit partition $ABC$ to accept updates. At time 2, all the dynamic algorithms accept updates arriving at the $AB$ partition. Voting, however, denies all updates at time 2. At time 3, $CDE$ and $A$ are the distinguished partitions for voting and dynamic-linear, respectively. (As before, we use lexicographic order to select the distinguished site for dynamic-linear and the hybrid algorithm.) The other algorithms deny all updates at time 3. Note that if updates are distributed roughly uniformly over the sites, and if our performance measure is the fraction of updates that succeed (see subsection C), then the performance of voting is strictly better than the performance of dynamic-linear at time 3, since voting's distinguished partition ($CDE$) is three times as large as dynamic-linear's distinguished partition ($A$). At time 4, only dynamic-linear and the hybrid algorithm permit updates. The hybrid algorithm performs better than dynamic-linear at time 4, since its distinguished partition ($BC$) is larger than the single-site distinguished partition for dynamic-linear.

The preceding example shows that at some times under some scenarios, one algorithm is best, while at other times under other scenarios another algorithm will be best. The real question is this: which algorithm is more *likely*, in the long run, to be able to handle any given update request? That is, which algorithm has greater *availability*? (There are other measures by which one might compare pessimistic algorithms, for example, the amount of communication required, or the ease of implementation. The algorithms considered in this paper are very similar when

[5]We have focused on updates. Read-only requests may be handled as if they were updates, except that the version number, update sites cardinality, and distinguished sites list need not be modified.

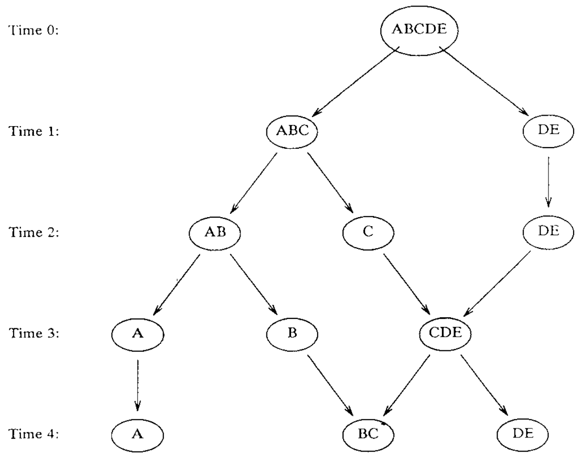


Fig. 1. A partition graph for a file replicated at sites $A$, $B$, $C$, $D$, and $E$.

compared under any of these other measures; the algorithms differ only in their availability.)

### B. *The Stochastic Model*

In this section we describe a stochastic model that makes precise what is meant by the phrase "more likely" in the preceding paragraph. The model is the same as that used previously to evaluate dynamic voting [21] and dynamic-linear [22], [24].

We make five assumptions to model stochastically the update availability of the network under dynamic algorithms. The first four assumptions duplicate assumptions that Pâris uses to analyze the availability of his *voting with witnesses* scheme [29]. The fifth assumption, however, causes our model to deviate from his.

- The communication links between sites are infallible. Only sites go up and down. Any site that is up can send a message to any other site that is up.
- The failures at the various sites form independent Poisson processes with *failure rate* $\lambda$. For any given site that is up (functioning), the probability that it goes down (fails) at or before the next $t$ time units is $1 - e^{-\lambda t}$.
- Similarly, the repairs at the various sites form independent Poisson processes with *repair rate* $\mu$.
- Updates are instantaneous. We ignore communication delays in the commit protocol.
- Updates are frequent: after any failure or repair, an update always arrives at a functioning site and is processed before the next failure or repair. An alternative assumption that yields the same model is frequent polling: after any failure or repair, the functioning sites communicate to determine the new status of the system before the next failure or repair.

We hasten to note that the above assumptions exist only for purposes of modeling; *the algorithm works correctly even when the above assumptions are violated.* The virtues and shortcomings of the model are discussed more fully in [21] and [24]. Briefly, the second and third assumptions are necessary to yield a Markov process. The first assumption keeps the number of states manageable. (We emphasize that our algorithm tolerates both site and link failures, although our modeling of its performance considers only the former.) The fourth assumption is justified because we are comparing algorithms whose communication delays are comparable, and because the communication delays are several orders of magnitude less than the typical time between failures or repairs. The fifth assumption is a reasonable approximation for many applications. In further defense of the model, we remark that no one has yet obtained analytic results for dynamic algorithms from models any stronger than the model we describe here.

### C. *Two Measures of Availability*

The standard measure of availability (see [1], [6], [29], and [33], for example) is the limit as $t$ goes to infinity of the probability that a distinguished partition exists at time $t$, where the definition of "distinguished" depends on the algorithm used. An alternative measure is the limit as $t$ goes to infinity of the probability that an update arriving at an arbitrary site at time $t$ will succeed. This alternative measure requires not only that a distinguished partition exist, but also that the update arrive at a functioning site in the distinguished partition. The traditional measure is appropriate if updates are viewed as arriving to the *system;* the alternative measure is more appropriate if updates arrive at an individual *site*. In this paper, we use the alternative measure, deeming it more appropriate for our application.

### D. *The State Diagram for the Hybrid Algorithm*

The action of the hybrid algorithm is initially the same as that of dynamic voting. The system begins with all $n$ sites in the distinguished partition. Eventually one site fails. Our fourth assumption ensures that before another failure occurs or the failed site is repaired, an update arrives at a functioning site. The distinguished partition finds that it now contains $n - 1$ of the $n$ sites with up-to-date copies of the file—still a majority. The update sites cardinalitiès are adjusted to $n - 1$ at the $n - 1$ functioning sites. If a second failure then occurs, the distinguished partition will soon thereafter discover that it contains $n - 2$ of the $n - 1$ sites with up-to-date copies of the file—still a majority, so the update sites cardinalities will be adjusted to $n - 2$ at the $n - 2$ functioning sites. The process continues, with update sites cardinalities always increasing or decreasing by one, until there are only three sites in the distinguished partition and a failure then occurs. Now the hybrid algorithm switches to its static phase. Although the subsequent update will be accepted, the update sites cardinality remains at three, with the distinguished sites list naming the three sites that form the distinguished partition, only two of which are currently functioning. Now if any site subsequently is repaired, the system will change the distinguished sites list to name the new trio of sites. Suppose instead that one of the two functioning sites fails, so that only a single site is now functioning. The subsequent update is blocked—one out of three sites is not a majority. Of these three sites in the most recent distinguished partition, call the one still up site $U$ and the two down sites $D_1$ and $D_2$. From this state, one of three events can occur.

- Site $D_1$ or $D_2$ might be repaired. The two-of-three distinguished partition is restored and the action of the network continues in the fashion described thus far.
- One or more of the other $n - 3$ failed sites might be repaired. If sometime later site $D_1$ or $D_2$ is repaired and an update arrives, the functioning sites include two of the three sites on the distinguished sites list, hence a majority. In this case, however, the newly-formed distinguished partition will also include the other sites that have meanwhile been repaired.
- Site $U$ might fail. Now two of the three sites $U$, $D_1$, and $D_2$ must be repaired before a new distinguished partition will be formed. Again any such newly-formed distinguished partition will also include any other sites that have meanwhile been repaired.

The state diagram we have just described is shown in Fig. 2. State $(X, Y, Z)$ is the state in which:

- the update sites cardinality of each up-to-date copy of the file is $Y$;
- $X$ of the $Y$ sites with update sites cardinality $Y$ are up;
- $Z$ of the $n - Y$ other sites are up.

Arcs in the state diagram indicate the rate at which the system moves from state to state. For example, the transition rate from state (3, 3, 0) to state (2, 3, 0) is $3\lambda$ because there are three functioning sites, each of which has failure rate $\lambda$.

An update request will be accepted if it arrives at a functioning site and the network is in any of the states on the top row of Fig. 2. Let $A_2, A_3, \cdots, A_n$ denote the steady-state probabilities of the top-row states, listed from left to right. Then the availability of the hybrid algorithm is

$$\sum_{k=2}^{n} \frac{k}{n} A_k.$$

The $k/n$ term reflects the fact that the site at which the update request arrives must be one of the $k$ functioning sites in state $A_k$.

To find the steady-state probabilities of the states in Fig. 2, one sets the flow-out of each state equal to the flow-in to obtain the so-called *balance equations*, one equation per state. One of the $3n - 5$ equations thus obtained is redundant and can be replaced by the equation that says the probabilities sum to 1. It appears difficult to find a product-form solution to this system of equations. However, for fixed $n$ and fixed repair/failure ratio $\mu/\lambda$, the system is easily solved by any numerical technique for systems of linear equations. Furthermore, for fixed small values of $n$, a symbolic manipulator like Maple [11] can solve the system in terms of $\mu/\lambda$.

### E. *Results*

*Theorem 2:* The availability of the hybrid algorithm is greater than the availability of dynamic voting.

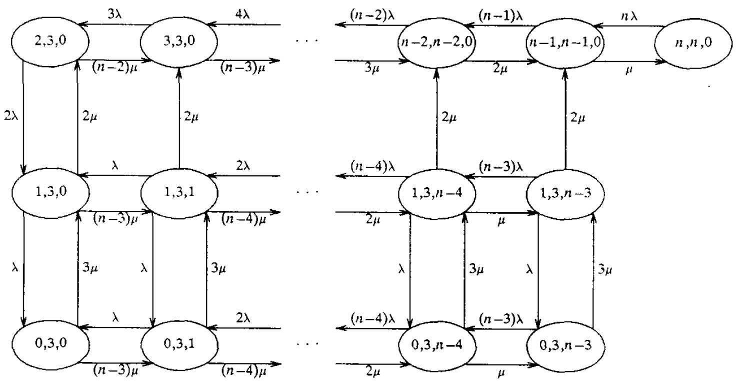


Fig. 2. The state diagram for the hybrid algorithm.

*Proof:* We describe an algorithm $X$ and show the following.

1) Algorithm $X$ has the same availability as the hybrid algorithm.

2) Every update accepted by dynamic voting is accepted by algorithm $X$.

The result then follows.

Algorithm $X$ proceeds exactly as the hybrid algorithm, except that it also changes the distinguished sites list at certain instants, namely, whenever a failure occurs that leaves only a single site still functional. Call such instants *critical* moments. To know when such failures are about to occur, algorithm $X$ requires foreknowledge. This makes the algorithm unimplementable, but does not preclude its use in this proof.

To specify how algorithm $X$ selects the distinguished sites list at critical moments, we simulate dynamic voting on the network of interest. Consider the state of this simulation at a critical moment. In the simulation, dynamic voting must have update sites cardinality equal to two. (It is never less than two. Nor can it be more: just before a critical moment, there are exactly two sites up, and the assumption that updates are frequent permits the update sites cardinality to be set to two.) Since the update sites cardinality is two, there are two sites with current copies of the file, in the simulation. Let $A$ and $B$ denote these sites.

*Case i):* One of these two sites is the single site up just after the critical moment. In this case, algorithm $X$ changes its distinguished sites list to $\{A, B, C\}$, where $C$ is any other site.

*Case ii):* Both sites $A$ and $B$ are down just after the critical moment. In this case, algorithm $X$ changes its distinguished sites list to $\{A, B, C\}$, where $C$ denotes the single site that is up.

Note that in either case, the new distinguished sites list consists of the single site that is up and two sites that (after the failure) are down. In either case, the new distinguished sites list is stochastically identical to the original list, which also contains the single site that is up and two sites that (after the failure) are down. This follows from the facts that failures and repairs are memoryless (Poisson process), and rates are the same at all the sites. That is, the computation of the availability is the same whether the distinguished sites list is $\{A, B, C\}$ (as in algorithm $X$) or a different trio (as in the hybrid algorithm), as long as each trio contains one site that is up and two that are down. Thus, algorithm $X$ has the same availability as the hybrid algorithm.

Also note that algorithm $X$ is rigged so that whenever it forbids updates, its distinguished sites list includes the distinguished sites list of the simulated dynamic voting. Thus, dynamic voting must also deny updates, whenever algorithm $X$ does so. □

Dynamic-linear, dynamic voting, ordinary voting, and voting with a primary site have previously been compared under the stochastic model of this section [22], [24]. Dynamic-linear has the most availability of these four algorithms, except when there are three sites; then ordinary voting has the greatest availability, except when the repair/failure ratio is unreasonably small. The next theorem compares dynamic-linear against the hybrid algorithm.

*Theorem 3:* For $n$ from 3 to 20, there is crossover point $c$ such that if the repair/failure ratio $\mu/\lambda > c$, the availability of the hybrid algorithm is greater than the availability of dynamic linear, while the reverse is true for $\mu/\lambda < c$. The approximate values of the crossover points are as follows.

**Hybrid > Dynamic-linear IF:**

| | | |
|---|---|---|
| $n = 3$ | and | $\mu/\lambda \geq 0.82$ |
| $n = 4$ | and | $\mu/\lambda \geq 0.67$ |
| $n = 5$ | and | $\mu/\lambda \geq 0.63$ |
| $n = 6$ | and | $\mu/\lambda \geq 0.64$ |
| $n = 7$ | and | $\mu/\lambda \geq 0.66$ |
| $n = 8$ | and | $\mu/\lambda \geq 0.70$ |
| $n = 9$ | and | $\mu/\lambda \geq 0.75$ |
| $n = 10$ | and | $\mu/\lambda \geq 0.81$ |
| $n = 11$ | and | $\mu/\lambda \geq 0.86$ |
| $n = 12$ | and | $\mu/\lambda \geq 0.92$ |
| $n = 13$ | and | $\mu/\lambda \geq 0.97$ |
| $n = 14$ | and | $\mu/\lambda \geq 1.01$ |
| $n = 15$ | and | $\mu/\lambda \geq 1.05$ |
| $n = 16$ | and | $\mu/\lambda \geq 1.08$ |
| $n = 17$ | and | $\mu/\lambda \geq 1.11$ |
| $n = 18$ | and | $\mu/\lambda \geq 1.14$ |
| $n = 19$ | and | $\mu/\lambda \geq 1.16$ |
| $n = 20$ | and | $\mu/\lambda \geq 1.19$ |

In sum, for networks with three to twenty sites, *the hybrid algorithm has greater availability than the dynamic-linear algorithm*, and hence greater availability than voting (for four or more sites), for all reasonable repair/failure ratios.

*Proof:* We repeated the following steps for $n$ from 3 to 20, achieving a mechanically-aided proof for each of these values of $n$. First, we coded the *balance equations* for dynamic-linear and the hybrid algorithm in the mathematics-like language of the symbolic manipulator Maple [11]. The equations are obtained from setting flow-in (weighted by the source of the flow) equal to flow-out in the state diagrams for the algorithms. For example, the Maple equation associated with the top left state in Fig. 2 is

$$2 * \mathrm{mu} * \mathrm{B}[1] + 3 * \mathrm{lambda} * \mathrm{A}[3]$$
$$= ((\mathrm{n} - 2) * \mathrm{mu} + 2 * \mathrm{lambda}) * \mathrm{A}[2];$$

where the probabilities of the top row states are named (left to right) A[2], A[3], $\cdots$, A[n], and the probabilities of the middle row states are named B[1], B[2], $\cdots$, B[n − 2]. The state diagram for dynamic-linear appears in [22]; the associated equations appear in [24]. For both algorithms, one of the $3n - 5$ equations thus obtained is redundant and can be replaced by the equation that says the probabilities sum to one.

Next, for fixed $n$, we used the *solve* routine of the symbolic manipulator Maple to solve the balance equations for dynamic-linear and the hybrid algorithm. This computation was symbolic (the single variable was the repair/failure ratio $\mu/\lambda$) and exact (no roundoff-error). We then substituted the results into the expression given earlier for site availability. This yielded expressions for the availabilities of the two algorithms.

We then formed the difference between these two availabilities and applied Maple's *normal* routine. This routine converted the difference into an equivalent expression in the form of a fraction whose numerator and denominator were both polynomials in $\mu/\lambda$ with rational coefficients. To compare the availabilities of dynamic-linear and the hybrid algorithm, it suffices to find the zeros of the numerator of the difference polynomial, since the difference is a continuous and bounded function of $\mu/\lambda$.

We used Maple's *fsolve* routine to find the zeros of the difference polynomial, using floating point arithmetic. The results are the crossover points listed in the theorem. Results from *fsolve* are approximate, since the computation is subject to roundoff error. However, we truncated the result from *fsolve* to three decimal places and substituted this truncated, rational value into the exact, symbolic expression for the difference between the availabilities, obtained from *solve* as described above. This difference, computed exactly using rational arithmetic, proved to be positive. We then repeated the computation using the same number plus $1/1000$, and found the difference to be negative there. Since the availabilities are continuous bounded functions of $\mu/\lambda$, the crossing point is bracketed by these two numbers and the direction of the inequality is as stated in the theorem. This procedure verifies that the crossing points listed are accurate to the degree specified (two decimal places).

The *fsolve* routine may miss some zeros if the polynomial is particularly ill-conditioned. To show that there is only a single crossing point for each value of $n$, we used Descartes' Rule of Sign, as described in [12]. Descartes' rule states that if we list the nonzero coefficients of polynomial $p(x)$ from highest power of $x$ to lowest power, and then count the number $C$ of times this sequence changes sign, then this number $C$ exceeds the number of positive zeros of $p(x)$, multiplicities counted, by an even nonnegative integer. Thus, if the number $C$ of sign changes is 1, then there is only a single positive zero. The numerator polynomial of the difference in the availabilities, as described above, had only a single sign change. From Descartes' rule, there is only a single crossing point.

All computations in this proof are exact, except for the computation of the precise point at which the availability of the hybrid algorithm overtakes the availability of dynamic-linear. However, in any mechanically aided proof there is the possibility of an error because of a bug in the software. To increase our belief that no error has been introduced by a bug, we computed the availabilities of both algorithms numerically, for $\mu/\lambda$ from 0.1 to 20.0 at intervals of 0.1. These computations were through a different set of software, although still part of Maple. The 3600 data points thus obtained agreed with the theorem, thereby reinforcing our confidence in the software.

The Maple code that constitutes this proof is available from the authors, via anonymous ftp or U.S. mail. □

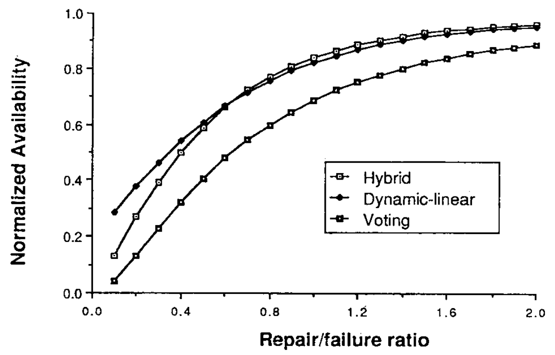


Fig. 3. Normalized availability for five sites; small repair/failure ratio.

Figs. 3 and 4 display the above comparison graphically for a typical case: five sites. The repair/failure ratio varies from 0.1 to 2.0 in Fig. 3, and from 2.0 to 10.0 in Fig. 4. Each figure has curves for the hybrid algorithm, dynamic-linear, and ordinary voting. Each curve is a graph of the *normalized availability* versus repair/failure ratio, where the normalized availability is

$$\text{availability}/\text{Prob (an arbitrary site is up)}.$$

No algorithm can have availability higher than the probability that an arbitrary site is up, given our nontraditional measure of availability. Hence, Figs. 3 and 4 graph the performance of the algorithms as a fraction of the best performance possible.

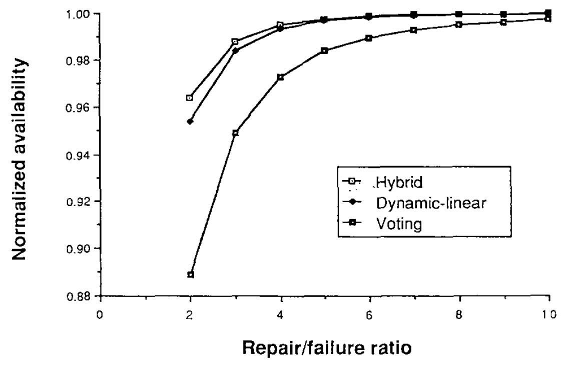


Fig. 4. Normalized availability for five sites; big repair/failure ratio.

## VII. Summary and Challenges

One can view the dynamic-linear algorithm as providing a practical means for dynamically reassigning votes: each participant in an update gets one vote, the distinguished site gets one extra vote (when the number of sites participating is even), and nonparticipants get no votes. Viewed this way, the dynamic-linear algorithm shows how very simple data structures (just three integers) can provide a practical means for accomplishing the group consensus method of dynamic vote reassignment suggested by Barbara, Garcia-Molina, and Spauster [6]. The hybrid algorithm goes one step further. It permits the sites to change algorithm in midstream: when three sites do an update, they halt the dynamic reassignment of votes until additional sites join the partition. If one of the three sites drops off, the two remaining sites still form a distinguished partition but choose to let the third site retain its vote.

The stochastic analysis of the hybrid algorithm shows that the hybrid algorithm has made a wise choice. For all reasonable repair/failure ratios, the availability of the hybrid algorithm exceeds that of its cousin algorithm, dynamic-linear. The price paid for this improvement is minimal: an integer variable must be converted to a triple of integer variables.

Indeed, even this price can be avoided, by making two changes to the hybrid algorithm.

*Change 1:* When exactly two sites perform an update, set the update sites cardinality to 2 and set the distinguished site entry to name one of the sites that is down, say, the site that most recently failed. (The original version of the hybrid algorithm would leave the update sites cardinality and distinguished sites list unchanged in this case.)

*Change 2:* The new rule for judging whether a partition $P$ is distinguished is as follows. Let $M$ be the largest version number among the sites participating in the update. Let $I$ be the set of sites with version number $M$, and let $SC$ and $DS$ be their common update sites cardinality and distinguished site entry.

*Case 1:* If $SC$ is 3 or more, use the same rule that the hybrid algorithm uses when $SC$ is 4 or more. That is, the partition is distinguished if it contains more than half of the $SC$ sites with version number $M$, or if it contains exactly half of those sites, including site $DS$.

*Case 2:* If $SC$ is 2, the partition is distinguished if it contains both of the sites with version number $M$, or if it contains exactly one of these two sites and (in addition) contains site $DS$.

It is not hard to see that the modified algorithm, which uses the same data structures that dynamic-linear uses, permits exactly the same updates as the unmodified hybrid algorithm does.

The modified version of the hybrid algorithm makes obvious the fact that it is but one of many hybrids possible. For example, one could permit $DS$ to be an arbitrary set of sites, with a majority of them required to "break the tie" when a partition contains exactly half of the sites with current copies of the file. Indeed, the members of a distinguished partition may convert to any vote reassignment they choose (or more generally, any valid coterie [5], [18], [26]), as long as adequate data structures are provided. The challenge is to determine the optimal algorithm. By the *optimal* algorithm is meant the algorithm that provides greatest availability, taken over the class of all algorithms that ensure the consistency of a replicated database in the face of network partitioning. The enormity of the class of algorithms considered makes this an ambitious task.

We once believed that the hybrid algorithm was this optimal algorithm, in the context of our homogeneous stochastic model, for sufficiently large repair/failure ratios. Preliminary evidence suggests that the hybrid algorithm is in turn bested by the algorithm that proceeds exactly as in the modified hybrid algorithm, except when exactly two sites perform an update. In that case instead of setting the distinguished site entry to a *single* site, set it to the set of *all* sites except the two sites that are doing the update.[6]

There has been much work recently to establish the optimal *static* assignment of votes or coteries in various heterogeneous models and to find heuristics that approach this optimum [1], [2], [4], [5], [18]. These are models which lack symmetry in communication links and uniformity in repair/failure ratios. The existence of practical dynamic algorithms provides a greater challenge: what is the optimal *dynamic* assignment of votes in such heterogeneous models, and what heuristics approach this optimum?

## References

[1] M. Ahamad and M. Ammar, "Performance characterization of quorum-consensus algorithms for replicated data," in *Proc. Symp. Reliability Distributed Software Database Syst.*, 1987, pp. 161-168.
[2] M. Ammar, M. Ahamad, and S. Cheung, "Reliability analysis of mutual exclusion using voting in a heterogeneous distributed sys-

[6]In practice, this new algorithm can be implemented without listing all these sites: when the update sites cardinality is two, updates are permitted if the partition includes both of the sites with current copies of the file, or if the partition contains one of them and more than half of the total sites.

tem," Dep. Comput. Sci., Georgia Inst. Technol., Atlanta, Tech. Rep.

[3] R. Attar, P. A. Bernstein, and N. Goodman, "Site initialization, recovery, and backup in a distributed database system," in *Proc. 6th Berkeley Workshop Distributed Data Management Comput. Networks*, 1982, pp. 185-202.

[4] D. Barbara and H. Garcia-Molina, "The reliability of voting mechanisms," Dep. Elec. Eng. Comput. Sci., Princeton Univ., Princeton, NJ, Tech. Rep. TR 330, 1984.

[5] —, "The vulnerability of vote assignments," *ACM Trans. Comput. Syst.*, vol. 4, no. 3, pp. 187-213, 1986.

[6] D. Barbara, H. Garcia-Molina, and A. Spauster, "Policies for dynamic vote reassignment," in *Proc. IEEE Conf. Distributed Comput.*, 1986, pp. 37-44.

[7] D. Barbara, H. Garcia-Molina, and A. Spauster, "Protocols for dynamic vote reassignment," in *Proc. 5th ACM Symp. Principles Distributed Comput.*, 1986, pp. 195-205.

[8] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surveys*, vol. 13, no. 2, pp. 185-221, 1981.

[9] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Reading, MA: Addison-Wesley, 1987.

[10] S. Ceri and G. Pelagatti, *Distributed Databases: Principles and Systems*. New York: McGraw-Hill, 1984.

[11] B. W. Char, G. J. Fee, K. O. Geddes, G. H. Gonnet, and M. B. Monagan, "A tutorial introduction to Maple," *J. Symbolic Computat.*, vol. 2, no. 2, pp. 179-200, 1986.

[12] G. E. Collins and R. Loos, "Real zeros of polynomials," in *Computer Algebra, Symbolic and Algebraic Computation*, B. Buchberger, G. E. Collins, R. Loos, and R. Albrecht, Eds. New York: Springer-Verlag, 1982 (Supplement 4 of *Computing*).

[13] D. Davcev and W. Burkhard, "Consistency and recovery control for replicated files," in *Proc. 10th ACM Symp. Operating Syst. Principles*, 1985, pp. 87-96.

[14] S. B. Davidson, H. Garcia-Molina, and D. Skeen, "Consistency in partitioned networks," *ACM Comput. Surveys*, vol. 17, no. 3, pp. 341-370, 1985.

[15] A. El Abbadi, D. Skeen, and F. Christian, "An efficient, fault-tolerant protocol for replicated data mangagement," in *Proc. 4th ACM Symp. Principles Database Syst.*, 1985, pp. 215-228.

[16] A. El Abbadi and S. Toueg, "Availability in partitioned replicated databases," in *Proc. 5th ACM Symp. Principles Database Syst.*, 1986, pp. 240-251.

[17] —, "Maintaining availabiity in partitioned replicated databases," Dep. Comput. Sci., Cornell Univ., Ithaca, NY, Tech. Rep. TR-87-857.

[18] H. Garcia-Molina and D. Barbara, "How to assign votes in a distributed system," *J. ACM*, vol. 32, no. 4, pp. 841-860, 1985.

[19] D. K. Gifford, "Weighted voting for replicated data," in *Proc. 7th Symp. Operating Syst. Principles*, 1979, pp. 150-162.

[20] S. Jajodia, "Managing replicated files in partitioned distributed database systems," in *Proc. IEEE 3rd Int. Conf. Data Eng.*, 1987, pp. 412-418.

[21] S. Jajodia and D. Mutchler, "Dynamic voting," in *Proc. ACM SIGMOD Int. Conf. Management Data*, 1987, pp. 227-238.

[22] —, "Enhancements to the voting algorithm," in *Proc. 13th Int. Conf. Very Large Data Bases*, 1987, pp. 399-406.

[23] —, "A pessimistic consistency control algorithm for replicated files which achieves high availability," *IEEE Trans. Software Eng.*, vol. 15, pp. 39-46, 1989.

[24] —, "Dynamic voting algorithms for maintaining the consistency of a replicated file," *ACM Trans. Database Syst.*, to be published.

[25] W. H. Kohler, "A survey of techniques for synchronization and recovery in decentralized computer systems," *ACM Comput. Surveys*, vol. 13, no. 2, pp. 149-183, 1981.

[26] L. Lamport, "The implementation of reliable distributed multiprocess systems," *Comput. Networks*, vol. 2, pp. 95-114, 1978.

[27] T. Minoura and G. Wiederhold, "Resilient extended true-copy token scheme for a distributed database system," *IEEE Trans. Software Eng.*, vol. SE-8, no. 3, pp. 173-189, 1982.

[28] J.-F. Paris, "Voting with a variable number of copies," in *Proc. IEEE Int. Symp. Fault-Tolerant Comput.*, 1986, pp. 50-55.

[29] —, "Voting with witnesses: A consistency scheme for replicated files," in *Proc. IEEE Int. Conf. Distributed Comput.*, 1986, pp. 606-612.

[30] M. Pease, R. Shostak, and L. Lamport, "Reaching agreements in the presence of faults," *J. ACM*, vol. 27, no. 2, pp. 228-234, 1980.

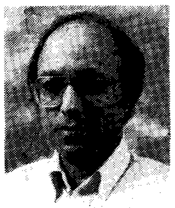
[31] R. Schlicting and F. Schneider, "Fail-stop processors: An approach to designing fault-tolerant distributed computing systems," *ACM Trans. Comput. Syst.*, vol. 1, no. 3, pp. 222-238, 1983.

[32] J. Seguin, G. Sergeant, and P. Wilms, "A majority consensus algorithm for the consistency of duplicated and distributed information," in *Proc. IEEE Int. Conf. Distributed Comput. Syst.*, 1979, pp. 617-624.

[33] P. G. Selinger, "Replicated data," in *Distributed Databases*, I. W. Draffen and F. Poole, Eds. Cambridge: Cambridge University Press, 1980, pp. 223-231.

[34] D. Skeen and M. Stonebraker, "A formal model of crash recovery in a distributed system," *IEEE Trans. Software Eng.*, vol. SE-9, no. 3, pp. 219-228 1983.

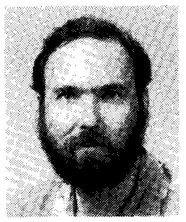
[35] R. H. Thomas, "A majority consensus approach to concurrency control," *ACM Trans. Database Syst.*, vol. 4, pp. 180-209, June 1979.



**Sushil Jajodia** (M'88-SM'88) received the Ph.D. degree from the University of Oregon, Eugene.

He is currently Associate Professor of Information Systems and Systems Engineering at the George Mason University, Fairfax, VA. He joined GMU after serving as the Director of the Database and Expert Systems Program within the Division of Information, Robotics, and Intelligent Systems at the National Science Foundation. Before that he was Head of the Database and Distributed Systems Section in the Computer Science and Systems Branch at the Naval Research Laboratory, Washington, DC, and Associate Professor of Computer Science and Director of Graduate Studies at the University of Missouri, Columbia. His research interests include database management systems, distributed systems, and parallel computing. He has published more than 50 technical papers in the refereed journals and conference proceedings and has coedited two books.

Dr. Jajodia has served in different capacities for various journals and conferences. He is on the Editorial Board of the IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING. He has edited special issues of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, the *Journal of Systems and Software*, and the *Bulletin on Data Engineering*. He chairs the IEEE Computer Society Magazine Advisory Committee and is serving as the general co-chair of the Second International Symposium on Databases in Parallel and Distributed Systems. He is a member of the IFIP Working Group 11.3 on Database Security and of the IEEE Computer Society Publication Planning Committee, and served as the chair of the IEEE Computer Society Technical Committee on Data Engineering for two years. He is a member of the IEEE Computer Society and the Association for Computing Machinery.



**David Mutchler** received the B.S. and M.S. degrees in mathematics from the University of Virginia, Charlottesville, in 1976 and 1980, respectively, and the Ph.D. degree in computer science from Duke University, Durham, NC, in 1986.

He is currently an Assistant Professor in Computer Science at the University of Tennessee, Knoxville, having joined the faculty there in the fall of 1987. Before that he was a member of the Computer Science and Systems Branch at the Naval Research Laboratory, Washington, DC. His research interests include the probabilistic analysis of heuristic search, the theory of game-playing, distributed databases, and cryptography.

Dr. Mutchler is a member of the IEEE Computer Society and the Association for Computing Machinery.