Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database

SUSHIL JAJODIA
George Mason University
and
DAVID MUTCHLER
The University of Tennessee

There are several replica control algorithms for managing replicated files in the face of network partitioning due to site or communication link failures. Pessimistic algorithms ensure consistency at the price of reduced availability; they permit at most one (distinguished) partition to process updates at any given time. The best known pessimistic algorithm, *voting*, is a "static" algorithm, meaning that all potential distinguished partitions can be listed in advance. We present a dynamic extension of voting called *dynamic voting*. This algorithm permits updates in a partition provided it contains more than half of the *up-to-date* copies of the replicated file. We also present an extension of dynamic voting called *dynamic voting with linearly ordered copies* (abbreviated as *dynamic-linear*). These algorithms are dynamic because the order in which past distinguished partitions were created plays a role in the selection of the next distinguished partition. Our algorithms have all the virtues of ordinary voting, including its simplicity, and provide improved availability as well. We provide two stochastic models to support the latter claim. In the first (site) model, sites may fail but communication links are infallible; in the second (link) model the reverse is true. We prove that under the site model, dynamic-linear has greater availability than any static algorithm, including weighted voting, if there are four or more sites in the network. In the link model, we consider all biconnected five-site networks and a wide variety of failure and repair rates. In all cases considered, dynamic-linear had greater availability than any static algorithm.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems—*reliability, availability, and serviceability*; D.4.3 [**Operating Systems**]: File Systems Management—*distributed file systems*; H.2.4 [**Database Management**]: Systems—*distributed systems, transaction processing*

General Terms: Algorithms, Performance, Reliability

Additional Key Words and Phrases: Consistency, dynamic voting, network partitioning, pessimistic algorithms, replica control, replication, serializability

Preliminary versions of this paper appeared in the *Proceedings of the ACM International Conference on Management of Data* (San Francisco, Calif., May 1987) and the *Proceedings of the 13th International Conference on Very Large Data Bases* (Brighton, England, September 1987). Much of this work was performed while the authors were at the Naval Research Laboratory, Washington, D.C. 20375-5000.

Authors' addresses: S. Jajodia, Department of Information Systems and Systems Engineering, George Mason University, Fairfax, VA 22030-4444; D. Mutchler, Department of Computer Science, The University of Tennessee, Knoxville, TN 37996-1301.

1. INTRODUCTION

A *partitioning* of a distributed database (DDB) occurs when the sites in the network split into disjoint groups of communicating sites due to site or communication failures. The sites in each group can communicate with each other, but no site in one group is able to communicate with sites in other groups. We refer to each such group as a *partition.*

The files in a distributed database may be replicated at various sites. When partitioning occurs, a dangerous situation begins: sites in one partition might perform an update to a file, while at the same time sites in another partition do a different update to the same file. If these two updates conflict, it may be difficult or impossible to resolve the conflict satisfactorily. Thus, the partitioned database is faced with a choice: either it accepts updates in more than one partition, in which case conflicts among copies of the replicated files are inevitable, or it accepts updates in at most one partition, in which case the availability of the replicated file is diminished.

The algorithms that select the first choice, accepting transactions at more than one partition, are called *optimistic,* since it is hoped that the inevitable conflicts among transactions are rare [11, 16, 40, 42, 44, 53, 54]. These algorithms take the approach that the database must be available even when the network partitions. Since the data of the partition might then diverge, the algorithms require a strategy for conflict detection and resolution. Usually, rollbacks are used as a means for preserving consistency; conflicting transactions are rolled back when partitions are reunited. Since coordinating the undoing of transactions is a very difficult task, optimistic algorithms are most useful in a situation in which the number of files in a particular database is large and the probability of conflicts among transactions is small.

The second class of algorithms share the philosophy that mutual consistency is of considerably greater importance than availability. These algorithms are called *pessimistic* algorithms, because they maintain the consistency of the file even if transactions arrive at sites in different partitions at the worst possible moment [2, 6–8, 15, 19–22, 26, 27, 30–33, 35, 37–39, 46, 51]. Consistency is enforced by permitting files to be updated only in a single *distinguished* partition at any given time. (This partition is often called the *majority* partition, because the most common pessimistic algorithms use voting.) As a consequence, any updates that are permitted in a partition do not conflict with updates in other partitions, assuring mutual consistency of files when partitions are reunited.

There are algorithms (see [23, 29, 44, 49], for example) that do not belong to either of the preceding two classes; however, they require a priori knowledge of the kind of updates to be made to the file. We make no such assumption in this paper.

In this paper, we consider only the pessimistic algorithms. Since these methods permit updates only in the single distinguished partition, they must reject updates arriving elsewhere. Furthermore, all pessimistic algorithms operating in networks subject to partitioning share the drawback that failures can occur in such a way that no updates can be performed anywhere in the system until these failures are repaired [43, 48]. Thus, the challenge is to devise replica control algorithms that preserve mutual consistency of replicated files and that, at the same time, provide improvement in the availability of files over that of existing schemes.

Voting [27, 46, 50, 51] is the best known example of a pessimistic scheme. In its simplest form, a file can be updated in a partition if and only if the partition contains more than half of the sites where the file is replicated. Thus, if a file is replicated at 100 sites, a distinguished partition will have to contain at least 51 of these sites. In the event that there does not exist such a partition, no updates can occur anywhere in the system.

In this paper we propose two generalizations to the voting scheme: called *dynamic voting* and *dynamic voting with linearly ordered copies* (abbreviated to *dynamic-linear*). These algorithms have several advantages over those of ordinary voting:

(1) In our dynamic algorithms, unlike ordinary voting, the number of sites necessary for an update is a function of the number of *up-to-date* copies in existence at the time of the update.
(2) A file can be updated in a partition if it contains more than half of the up-to-date copies. As a consequence, a file may be replicated at 100 different sites, yet it is possible for dynamic-linear to allow updates with only a single copy accessible.
(3) Changes to the quorum occur dynamically without any manual intervention.
(4) Like ordinary voting, our algorithms are simple to state as well as implement. They require only slight modifications to the voting scheme.
(5) Our algorithms do not require a complicated message-based coordination mechanism. The messages required are only a fraction more than that required by ordinary voting, and no more complicated.
(6) Under two different stochastic models, dynamic-linear provides greater availability than that provided by *any* static algorithm, including ordinary voting, if there are four or more sites with copies of the replicated file.

This paper differs from our prior publications in this area [31, 32] in several respects:

(1) The algorithms presented improve upon those reported previously by distinguishing between the physical and logical versions of the file.
(2) Sections 6 and 7 contain many details not published previously.
(3) Theorems 2 and 4 in Section 8 have not appeared previously, nor have any of the proofs given in that section.

This paper is organized as follows. The first half of the paper presents our dynamic algorithms, while the second half gives a stochastic analysis of them. The contents of the sections are as follows. In Section 2, we list the assumptions we make about the network. In Section 3, we introduce our own algorithms, followed by a more detailed description in Section 4. Section 5 contains a proof of correctness of our algorithms. In Section 6, we discuss the relationship between our algorithms and previous pessimistic algorithms. Section 7 contains some extensions to our algorithms. In Section 8, we present a stochastic analysis of our algorithms and several versions of ordinary voting. The analysis is done twice, once for a model in which sites are subject to failure but links are infallible, and once for a model in which the reverse is true. Finally, in Section 9 we list our conclusions and suggestions for further work.

2. SPECIFICATION OF THE PROBLEM

A DDB system consists of a collection of independent computers, called *nodes* or *sites*, connected via communication links. Both the nodes and the communication links may fail, but Byzantine failures [41] are ruled out. The sites failures are clean, that is, nodes stop executing without performing any incorrect actions [45]. 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 a 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* that ensures that the execution of all transactions at any site is serializable [9, 34]. A *replica control protocol* ensures that the replicated files are managed correctly in the presence of failures. (An excellent survey of several of these strategies is given by Davidson et al. [17].) In a pessimistic replica control protocol, mutual consistency of a replicated file is maintained by making sure that all files are updated in at most one partition at any given time. We call such a partition the *distinguished* partition. Different pessimistic protocols use different rules and different mechanisms to define the distinguished partition. When site or communication link recoveries cause partitions to unite, the nodes form a new partition and obtain, if necessary, all updates that they have missed. 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 [43, 48], the challenge is to come up with a pessimistic replica control algorithm that not only preserves mutual consistency of various copies of a file, but at the same time achieves high availability.

We assume for ease of exposition that there is a single file f that is replicated at n sites. The extension to transactions that update multiple files is straightforward. Any such transaction will require that a distinguished partition, as described below, exists for *every* file in the read and write set of the transaction. Further, since read requests alone cannot damage the mutual consistency of the replicated file, we focus only on updates. In Section 7.5 we explain how our approach can be generalized to handle read requests with different quorums from write requests.

3. THE ESSENTIALS OF OUR DYNAMIC VOTING ALGORITHMS

3.1 High-Level Statement of the Algorithms

In this section, we present an informal description of two new pessimistic algorithms: dynamic voting and dynamic voting with linearly ordered copies (called dynamic-linear for short). These algorithms are presented in their simplest form in this subsection, with an example of their conduct appearing in the next subsection. Succeeding sections present the algorithms in full detail, provide a correctness proof, and describe extensions to the algorithms.

We assume throughout this section that for the purposes of voting each file is assigned an equal weight (of one). It is possible to permit copies having different weights; this generalization is explained in Sections 7.3 and 7.4. Also, 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 that is replicated at n sites $S_1, S_2, \ldots, S_n$.

For static voting [27, 46, 50, 51] in its simplest form, the distinguished partition is the partition, if any, that contains more than half of the sites. Clearly, this ensures that there cannot exist more than a single such partition at any instant. There is a *version number* associated with each copy of the file. This version number is set to zero initially and is incremented by one each time the copy is updated. Thus, a version number represents up-to-dateness of a copy. Any copy in the distinguished partition whose version number is maximal is a current copy of the file. Before a new update is committed, a current copy is used to propagate missing updates to each site in the distinguished partition whose copy is not current. (Alternatively, one can use the notions of physical and logical copies, as described for our dynamic algorithms in Section 3.3.)

Simple voting is a *static* algorithm. Such algorithms are those for which one could, in principle, provide a list of groups of sites such that a group of sites is a distinguished partition if and only if it is on the list [5, 26]. We convert simple voting into a *dynamic* algorithm by associating with each copy of the file f not only the copy's version number but also a second integer variable called the *update sites cardinality*. This new algorithm, which we call simply dynamic voting, maintains the update sites cardinality of copy f_i 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 updates sites cardinality 7, then 7 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 basic operation of dynamic voting is simple. When a site S receives an update request, the site sends a message to all other sites, requesting certain information. Those sites belonging to the partition P to which S currently belongs (that is, those sites with which S can communicate at the moment) lock their copies of the file and reply to the inquiry sent by S. From the replies, S learns 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 the partition contains more than half of the SC sites with version number VN. If partition P is the distinguished partition, then S commits the update and sends a message to the other sites in P, telling them to commit the update and unlock their copies of the file too. For each of these sites, at the same time that the site commits the update, the site increments its version number and changes its update sites cardinality to equal the number of sites in partition P, that is, the number of sites participating in the update. A two-phase commit protocol [25, 28, 36] is used to ensure that transactions are atomic. The details of the communication protocol for dynamic voting are given in Section 4.

Incrementing the version number has the effect of reassigning votes in such a way that sites not participating in the update receive no votes. At each other site (those with votes), the update sites cardinality is the total number of votes currently available, that is, the number of sites that participated in the most recent update. Thus the update sites cardinality is maintained as promised. A site with no vote (i.e., a site with an old version of the file) can regain its vote when it rejoins a distinguished partition and participates in an update within that partition.

Our next algorithm, dynamic-linear, extends dynamic voting by adding a third variable, the *distinguished site* (DS). When there are 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 [24], 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 doing an update themselves at the same time, 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 and includes 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.

3.2 An Example of Dynamic-Linear

Assume there are five sites A, B, C, D, and E that have copies of the file f, where these sites are initially connected and form a single partition. We linearly order the sites lexicographically: $A > B > C > D > E$ and use this order to determine the distinguished site. Now, suppose the file f has been updated nine times, so the initial state can be represented as follows. The DS entry has significance only when the SC value is even; we use the symbol "—" in a DS entry to indicate that the DS value is immaterial.

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

At this point suppose site C receives an update, and it finds that it can communicate with sites D and E only. Since C still belongs to a distinguished partition, it can process the update. The state then changes to

	A	B	C	D	E
VN:	9	9	10	10	10
SC:	5	5	3	3	3
DS:	—	—	—	—	—

Suppose now that site C receives yet another update, and it discovers that it can communicate with site E only. The novelty here is that since sites C and E together contain more than half of the current copies of the replicated file, they form a distinguished partition even though there are only two sites (out of five) in this partition. Thus site C can process the update, after which the database state will be

	A	B	C	E	D
VN:	9	9	11	11	10
SC:	5	5	2	2	3
DS:	—	—	C	C	—

Suppose sites C and E perform four additional update operations and subsequently become isolated from each other. The system state changes to the following:

	A	B	C	E	D
VN:	9	9	15	15	10
SC:	5	5	2	2	3
DS:	—	—	C	C	—

The novelty of dynamic-linear is that at this point the partition consisting of the single site C is a distinguished partition and can process updates.

This example shows that site and link failures can occur in such a way that dynamic voting and dynamic-linear can process updates that must be rejected by simple voting. The reverse is true also. Suppose that the above example continues like this: site C fails and the remaining four sites regroup into a single partition. Then updates are blocked under both of the dynamic algorithms, while simple voting permits updates in the four-site partition. The evaluation of which is better, a static voting algorithm or a dynamic voting algorithm, is a stochastic question to which Section 8 is devoted.

3.3 Physical and Logical Version Numbers

It is desirable that, within any single partition, all copies of the replicated file be identical. As presented so far, our algorithm lacks that property—when a site joins a partition that is *not* a distinguished partition, the site cannot "catch up" from a more current version within the partition. This drawback is easily removed by associating a *logical version number* and *physical version number* with each copy of the file, instead of just the single version number used earlier. Here we sketch how these variables are used with dynamic-linear and give an example to show their usefulness. The next section describes these variables, along with the protocols for dynamic-linear, in more detail.

Each copy of the file has associated with it four integers: logical version number, physical version number, update sites cardinality, and distinguished site entry. When a site S wishes to do an update, it communicates with other sites in its partition P. Site S learns the biggest (most recent) *logical* version number LN among copies in P as well as the update sites cardinality SC and distinguished site entry DS of the copies with that version. As before, for partition P to be a distinguished partition, it must contain more than half of the *SC* sites with logical version number *LN*, or exactly half of the *SC* sites including the site named by the *DS* entry. However, there is now an additional requirement: at least one of the *SC* sites must have a *physically* current copy of the file, for without this the update cannot be physically performed! The physical version number is incremented each time a physical update to the copy occurs. Thus partition *P* contains a physically current copy of the file if one of the sites in *P* has a copy whose physical version number equals the maximal logical version number *LN*.

To show why it is useful to distinguish the logical version number from the physical version number, we consider again the scenario in the previous section. The last database state pictured in the example will now be repesented as follows. (Note: we have renamed the sites from the earlier example in order to simplify the depiction of partitions in this example.)

	A	*B*	*C*	*D*	*E*
LN:	15	10	15	9	9
PN:	15	10	15	9	9
SC:	2	3	2	5	5
DS:	*A*	—	*A*	—	—

Now, suppose that sites *D* and *E* resume communication with site *C*. Because of the distinction between physical and logical copies, sites *D* and *E* may receive missing updates although *CDE* is not a distinguished partition. That is, sites *D* and *E* can safely change their physical version numbers to 15 after doing the six missing updates obtained from site *C*. This illustrates one benefit of the introduction of logical/physical copies: all sites within a partition will have the same physical version of the file, except for a brief period after each recovery. Note that although it is always safe to do updates and change physical version numbers, it is not safe to change logical version numbers without going through the protocol to verify that a distinguished partition exists. In particular, changing a logical version number without changing the corresponding update sites cardinality would destroy the invariant that all copies within a given logical version number have identical update sites cardinalities. We use this fact in our proof of correctness of our algorithms.

After *D* and *E* make their copies current (using routine **Make_Current** in Section 4.3), the new database state would become

	A	*B*	*C*	*D*	*E*
LN:	15	10	15	9	9
PN:	15	10	15	15	15
SC:	2	3	2	5	5
DS:	*A*	—	*A*	—	—

Suppose that at this point site *A* receives an update and *A* finds that it can communicate with sites *B* and *C*. Site *A* determines that it has both of the copies with maximal logical version number 15 and that both of these copies are physically current. Site *A* therefore performs the update and sends to *B* and *C* the new data structures: $LN = 16$, $SC = 3$, and *DS* is immaterial. Site *A* also sends the update to site *C*. All these messages are received properly. Finally, site *A* sends the five missing updates, plus the new update, to site *B*. Before this last (lengthy) message is delivered, site *A* becomes isolated from *B* and *C*. The database state is now as follows.

	A	*B*	*C*	*D*	*E*
LN:	16	16	16	9	9
PN:	16	10	16	15	15
SC:	3	3	3	5	5
DS:	—	—	—	—	—

The site *B* becomes a site with a *logically* current copy, and so updates may continue to occur in the distinguished partition *BC*. Note that under similar circumstances, but without the distinction between physical and logical copies, site *B* would still be waiting for the missing updates and hence could not participate in an attempt to form a new distinguished partition. Thus sites *B* and *C* would not form a distinguished partition, and no updates would take place anywhere in the system. This illustrates a second advantage of the introduction of logical/physical copies: it permits us to remove from the locking portion of our protocol certain time-consuming actions (like the transfer of missing updates) that can conveniently be placed outside of this portion.

Suppose that there is one update in partition *BC*, but that *B* and *C* become isolated before *C* has a chance to bring the copy at *B* up-to-date, leading to the following state:

	A	*B*	*C*	*D*	*E*
LN:	16	17	17	9	9
PN:	16	10	17	15	15
SC:	3	2	2	5	5
DS:	—	*B*	*B*	—	—

Notice that the partition consisting of the single site *B* does not form a distinguished partition since even though it does not have a "logical majority," it does not contain a physically current copy. Now suppose that site *A* can communicate with site *C* and obtains its missing update from *C*, after which site *C* fails and sites *A* and *B* join together. Now the database state is

	A	*B*	*C*	*D*	*E*
LN:	16	17	17	9	9
PN:	17	10	17	15	15
SC:	3	2	2	5	5
DS:	—	*B*	*B*	—	—

At this point the partition *AB* is a distinguished partition since it has a "majority" of logically current copies and a copy that is physically current.

4. DETAILED DESCRIPTION OF OUR DYNAMIC VOTING ALGORITHMS

In the previous section, we said that the coordinating site sends a message to the other sites and awaits a reply. Because a distributed environment is subject to site and link failures, such message-passing requires more attention than would otherwise be necessary. Consider a site S that receives an update request. Our algorithms, like simple voting, must permit site S to inquire which sites are in its partition. The sites that admit to being in this partition must not respond to another such inquiry until they know that the first such inquiry has been concluded, lest two update protocols be occurring simultaneously. Whatever protocols simple voting might employ for such purposes, our dynamic algorithms can employ as well. The *only* extra communication cost incurred is the negligible cost caused by appending three integers to each message: the update sites cardinality, distinguished site entry, and the logical version number.

We now present one way to accomplish the communication protocol required by dynamic-linear. The protocol for dynamic voting can easily be gleaned from the protocol given for dynamic-linear.

4.1 Data Structures

We associate with each copy of the file f four variables[1]: logical version number, physical version number, update sites cardinality, and distinguished site, defined as follows. We continue to assume that sites are assigned a priori a linear ordering, denoted by $>$, and use this ordering to determine the distinguished sites.

Definition 1. The *logical version number* of a copy f_i at site S_i is an integer LN_i that counts the number of successful updates to the file f at or prior to the most recent time that site S_i participated in an update to f as a member of a distinguished partition. The site S_i sets LN_i to zero initially and updates it each time an update to f occurs with S_i as a member of a distinguished partition.

Definition 2. The *physical version number* of a copy f_i at site S_i is an integer PN_i that counts the number of successful updates to f_i. We set PN_i to zero initially and increment it by one each time an update to f_i occurs.

Thus the physical version number counts the actual updates performed to the copy, while the logical version number counts those updates plus any others to which the site has agreed but not yet actually performed. For example, suppose a site A has a copy of the replicated file with physical and logical version number 13. Suppose another site B has a copy with logical version number 17 and B informs site A that they are both permitted to do a new update. (The mechanism for this will be described shortly.) Then site A may change the logical version number of its copy to 18, but will not change the physical version number until the missing four updates and the new update show up.

Definition 3. Associated with each copy f_i at site S_i is another integer called the *update sites cardinality*, denoted by SC_i. This variable always reflects the

[1] These variables need not be physically attached to the file.

number of sites participating in the most recent update to f_i that site S_i participated in as a member of a distinguished partition. We set SC_i to n (number of sites) initially, and whenever site S_i participates in an update as a member of a distinguished partition, then SC_i is set to the total number of sites that participated in that update.

Definition 4. We associate with each copy f_i at site S_i a variable called *distinguished site*, denoted by DS_i. The value of DS_i is important only if the corresponding SC_i is even. When SC_i is even, DS_i identifies the site that is greater (in the linear ordering for the file f) than all other sites that participated in the most recent update to f_i.

Before we can state what we mean by a distinguished partition, we need additional terminology.

Definition 5. The *current version number* of a replicated file f is the maximum version number of all copies of f.

Definition 6. A copy is said to be *logically current* if its logical version number equals the current version number of the replicated file.

Definition 7. A copy is said to be *physically current* if its physical version number equals the current version number of the replicated file.

Definition 8. A copy is called *current* if it is both logically and physically current.

Definition 9. A partition P is said to be a *distinguished partition* if either of the following two conditions holds:

(a) the partition P contains more than half of the logically current copies and at least one physically current copy, or

(b) the partition P contains exactly half of the logically current copies and at least one copy that is physically current, and moreover, contains a site S such that (i) the copy of S is logically current, and (ii) $S > S'$, where S' is any other site containing a logically current copy of f.

4.2 Protocol under Normal Operation

In this subsection, we describe our protocol, assuming first that no sites or communication links fail during the execution of the protocol; in the next subsection, we describe 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 steps. All messages in this protocol (e.g., LOCK_REQUEST and VOTE_REQUEST) are with reference to file f, as are all data structures (e.g., LN_i).

(i) S issues a LOCK_REQUEST to its local lock manager. When the lock request is granted, S sends a VOTE_REQUEST message to all sites.

(ii) 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 LN_i, PN_i, SC_i, and DS_i.

(iii) The site S collects the responses from the various sites and decides whether it is a member of a distinguished partition (see routine **Is_Distinguished** in this subsection). Henceforth, we refer to S as the *coordinator* and the respondents as *subordinates*.

(iv) If S does not belong to a distinguished partition, it aborts the update, issues a RELEASE_LOCK request to its local manager, and sends ABORT messages to all the subordinates. When a subordinate receives the ABORT message, it too issues a RELEASE_LOCK message to its local lock manager. Steps (v) through (viii) do *not* apply if step (iv) is invoked.

(v) If S does belong to a distinguished partition, it determines whether its copy is current; if it is not current, S determines what subordinates have current copies of the file f (see routine **Catch_Up** in this subsection). 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.

(vi) Once the copy at S is current, S reads its copy of the file f and performs the update. Then S commits the update to the file f together with the modifications to its LN_i, PN_i, SC_i, and DS_i and sends to each S_i the COMMIT message and the new values for LN_i, SC_i, and DS_i (see the procedure **Do_Update** in this subsection). There is no need to send PN_i since it equals LN_i. Also, S sends the new update to each S_i whose copy f_i was current. Finally, S issues a RELEASE_LOCK request to its lock manager.

(vii) Each S_i acts on the message received from S (aborting or committing the update and associated data structures) and then issues a RELEASE_LOCK request to its lock manager.

(viii) S sends the missing updates to each subordinate whose copy f_i was not current (see the procedure **Send_Missing_Updates** that appears later in this subsection).

The three phases of our protocol are the voting phase (steps (i)–(iii)), the catch-up phase (step (v)), and the commit phase (steps (iv), (vi), and (vii)). Note that the collection and dissemination of votes are attached to the first and last phases of our protocol, respectively. These two phases also act as the two-phase commit protocol used to insure atomicity of transactions. Thus, collection and dissemination of votes require no extra rounds of communication. The middle (catch-up) phase is not necessary if the copy at S is current. Also note that the final step lies outside of the window of uncertainty [14] for each subordinate S_i. That is, each subordinate S_i is locked out from other updates during steps (i) through (vii) (its window of uncertainty), but may participate in other updates while step (viii) is being performed.

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

Is_Distinguished

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

(2) The site S then calculates: the value $M = \max\{LN_i \colon S_i \in P\}$, the set $Logical = \{S_j \in P \colon LN_j = M\}$, and the set $Physical = \{S_k \in P \colon PN_k = M\}$. M denotes the largest version number that is in P, the set *Logical* consists of those sites in P that have the logical version number M, and the set *Physical* contains those sites in P that have the physical version number M.

(3) If card(*Physical*) = 0, then S does not belong to a distinguished partition (since in this case P does not have any copy that is physically current).[2] Therefore, the site S

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

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.

(4) Otherwise, S takes the update sites cardinality of any site in the set *Logical.* (Any site in *Logical* will do; the choice is arbitrary, because all sites in *Logical* will have the same update sites cardinality.) Denote this by N. If card(*Logical*) $> N/2$, then S is a member of a distinguished partition.

(5) Otherwise, if card(*Logical*) $= N/2$, then select any site S_i in *Logical.* (Again the choice of S_i is arbitrary.) If $DS_i \in$ *Logical*, then S also belongs to a distinguished partition.

(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 *Logical* possess the logically current copy of the file f, and the sites in the set *Physical* have copies that are physically current. If the site S is not a member of the set *Physical,* it requests the missing updates from some site in *Physical* and processes those updates.

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

Do_Update

(I) During this phase, the sites commit the update. First, the coordinator S reads its copy of the file f and performs the update. Next, S commits the update together with the new values for the four variables associated with each copy:

$$
\begin{aligned}
LN_i &= M + 1\\
PN_i &= M + 1\\
SC_i &= \text{card}(P)\\
DS_i &= S' \text{ if card}(P) \text{ is even and } S' - S'' \text{ for all } S'' \text{ in } P.
\end{aligned}
$$

The site S sends to each subordinate the following: the COMMIT message and the new values for LN_i, SC_i, and DS_i (if card(P) is even). There is no need to send PN_i since it equals LN_i. Also, S sends the new update to the file to each site in the set *Physical.* The site S then releases the lock on its copy of the file.

(II) Each subordinate S_i, upon receiving the message from S, processes it as follows: Each S_i commits the new values for LN_i, SC_i, and DS_i (if received). Moreover, if S_i receives the new update as well, it commits the update together with its new PN_i which is equal to the new LN_i. Once this is done, S_i unlocks its copy of the file.

Although the update has been committed by all participants, the coordinator S is aware that sites in *P–Physical* have copies that require updating. Therefore, S executes the following steps:

Send_Missing_Updates

(i) The site S sends each subordinate in *P–Physical* the physical version number PN_i and the missing updates.

(ii) When a site S_j receives the message from S, S_j compares its local PN_j with PN_i. If $PN_i > PN_j$, S_j applies the missing updates and sets PN_j equal to PN_i.

Finally, note that our protocol may cause deadlocks to occur. There are several ways to handle deadlocks, and we refer the reader to Bernstein et al. [10] or Ceri and Pelagatti [12] for additional details.

Here is an example to illustrate the routines described in this section. Suppose the current state is as follows. This state might arise naturally in a manner similar to the example of Section 3.3.

	A	B	C	D	E	F	G
LN:	16	17	9	17	17	17	9
PN:	17	10	15	12	17	10	15
SC:	3	4	5	4	4	4	5
DS:	—	B	—	B	B	B	—

At this point, site C initiates the protocol to process an update by locking its file and sending a VOTE_REQUEST message to all sites. Prompt replies arrive from sites A, B, and D with their logical and physical version numbers, update sites cardinalities, and distinguished site entries. However, no replies arrive from the other sites. After a while, site C decides (perhaps because a timeout has been exceeded) to proceed with the protocol. If late replies arrive after this point from any of sites E, F, or G, site C must send an ABORT message to those sites. In the next subsection we see how sites E, F, and G can acquire up-to-date copies of the file.

Site C executes the **Is_Distinguished** routine. Here the partition P is $\{A, B, C, D\}$. The maximal version number M that C receives is 17, and sites B and D form the set *Logical* of sites with that logical version number. The set *Physical* of sites with 17 as their physical version number is the set containing only site A. The update sites cardinality N of the sites in *Logical* is 4. Since set *Physical* is not empty, site C compares the cardinality of *Logical*, which is two, versus $N/2$, which is also two. Since these are equal, site C checks the distinguished site entry of sites in *Logical*; this is site B. Since B is a member of *Logical*, a distinguished partition exists.

Site C then notices that its own copy is not physically current, since it is not a member of the set *Physical*. Site C executes the **Catch_Up** routine and acquires the two updates it is missing from site A, the sole member in *Physical*. Then site C executes the **Do_Update** routine. It sets its logical and physical version numbers to $M + 1$, here 18. It sets its update sites cardinality to 4, the number of sites in the partition. Since this is an even number, it selects a new distinguished site entry, say A. It commits the update together with the new values of these variables. Then site C sends the COMMIT message and the new values of these variables to sites A, B, and D. Site C also sends the new update to site A, which is the only other site ready to perform that update. After these messages are sent, site C releases the lock on its copy of the file.

When sites A, B, and D receive the COMMIT message, they commit the values sent. Site A also receives the new update and so performs the update and increments its physical version number. At this point, the database state is as follows:

	A	*B*	*C*	*D*	*E*	*F*	*G*
LN:	18	18	18	18	17	17	9
PN:	18	10	18	12	17	10	15
SC:	4	4	4	4	4	4	5
DS:	*A*	*A*	*A*	*A*	*B*	*B*	—

Finally, site *C* executes the **Send_Missing_Updates** routine. Sites *B* and *D* receive the missing updates and the new update and commit those updates along with physical version number 18.

4.3 Protocol when Failures Occur

In this subsection, we describe how our protocol adjusts to site and communication link failures. We 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* contacts all sites in its partition, call it *P*, and asks for their physical version numbers PN_i.

(b) *S* calculates the value $V = \max\{PN_i\colon S_i \in P\}$. If the physical version number of the copy at *S* is less than *V*, then *S* can request missing updates from any site S_i in *P* such that $PN_i = V$. Upon receiving the missing updates from S_i, *S* applies them to its copy and modifies its physical version number to equal *V* as well.

Notice that a recovering site can obtain missing updates even it is not a member of a distinguished partition. it might be desirable, however, for the recovering site to also become a voting member of the distinguished partition, if it can. The site can wait for the next update to come along in its partition, at which time the site will be included in the new distinguished partition if one is formed. Alternatively, the site can instigate the normal-operation protocol by processing a null update. That is, the site can run the following procedure.

Rejoin_Distinguished_Partition

A recovered site *S* can attempt to regain voting status in the distinguished partition by running the three-phase normal-operation protocol, but using a special **null** update. This update does not change the contents of the file. If the **null** update is aborted (because *S* does not belong to a distinguished partition), then *S* cannot regain voting status at this time but can try again later. Otherwise, the **null** update causes the current logical and physical version numbers of the file to be incremented and a new distinguished partition (containing site *S*) to be formed.

Section 7.2 discusses whether a site should run the **Rejoin_Distinguished_Partition** procedure or simply wait for an update to occur in its partition.

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, [10, chap. 7] or [12, sect. 9.2.2.2]). Here is one way to do so.

Definition 10. We say a site S is *blocked* with respect to a file being updated if it has responded to a VOTE_REQUEST message for that file from the coordinator, but has not received a COMMIT or ABORT message in return.

Suppose a site S_i is blocked. There are three ways in which a blocked site may become unblocked (i.e., correctly terminate the pending transaction). First, suppose a blocked site S_i receives a VOTE_REQUEST from some site S'. Then S_i sends the following values for its data structures: $LN_i = -\infty$, the actual value for PN_i, $SC_i = -\infty$, and $DS_i = -\infty$, where $-\infty$ is some value much smaller than any actual values for each of these variables. If S_i later receives a COMMIT from S', then S_i may commit the update sites cardinality, distinguished site entry, and logical version number sent by S' (and S_i is no longer blocked).

The second way for a blocked site S_i to become unblocked is by running the following termination protocol.

Termination_Protocol

(1) The *initiator* S sends a DECISION_REQUEST message to all sites, along with the unique id and timestamp of the VOTE_REQUEST message that is blocking S.[3]

(2) A site S' upon receiving the DECISION_REQUEST message sends COMMIT to S if S' committed the update, sends ABORT to S if S' aborted the update, and otherwise does nothing.

(3) If S receives a COMMIT or ABORT from *any* responder, it too can do likewise; otherwise it must wait for some period of time and run this protocol again.

The above termination protocol can be modified into one that is more elaborate (see, for example, [12, Sect. 9.2.2.2]). The initiator S can further ask how many sites had responded to the VOTE_REQUEST message. If S determines that the sites that did not (and will not) respond to the VOTE_REQUEST message form a distinguished partition, it can go ahead and abort the update.

Note that a blocked site can obtain at any time missing updates from any site by running the procedure **Make_Current**. This provides a third way in which a site may become unblocked: it can unblock if it obtains any updates whose physical version number is greater than the current logical version number at the blocked site.

Continuing the example from the previous subsection, suppose that site E joins the partition containing sites F and G. Here is the current state of the database.

	A	B	C	D	E	F	G
LN:	18	18	18	18	17	17	9
PN:	18	18	18	18	17	10	15
SC:	4	4	4	4	4	4	5
DS:	A	A	A	A	B	B	—

Suppose that sites F and G now recover from a failure. They each execute the **Make_Current** routine, thereby acquiring missing updates from site E and changing their physical version numbers to 17. The three sites E, F, and G would then periodically try the **Rejoin_Distinguished_Partition** routine. When one of these succeeds, after the link separating the two partitions is repaired, then

[3] We assume that, as is typical in distributed message-passing, all messages are uniquely identified to permit subsequent references to them.

all 7 sites will change to logical version number 19 and update sites cardinality 7. The site which coordinated the update to logical version 19 will then execute **Send_Missing_Updates** and bring all sites to physical version number 19.

4.4 Addition and Deletion of Sites

In this subsection, we show how we may add or delete copies of the replicated file f dynamically, without any special intervention. Suppose a site S that does not have a copy of file f wishes to have one. The site S sets the logical and physical version numbers of its copy to $-\infty$, where $-\infty$ is some value much smaller than any actual values for the version numbers. Then, it may execute the **Make_Current** routine to obtain as many updates as are available, or the most recent version of the file if that is more convenient. Afterward, it will join the distinguished partition when its partition forms a new distinguished partition, or it can initiate said regrouping by executing the **Rejoin_Distinguished_Partition** routine. The site S should also notify all other sites having copies of f that S now has a copy too, so that S is included in the sites polled when an update arrives in those sites. Also, the linear ordering of sites used to select the distinguished site entry should be modified to include S.

It is equally easy for copies to be deleted from the system. Suppose a site S decides that it no longer wants to have its own copy of the file f. It executes the **Rejoin_Distinguished_Partition** routine (that is, it submits a null update to the normal-operation protocol), but with two changes.

Case 1. Suppose that after the first phase of the protocol, some copy in the partition to which S belongs is found to have a copy more logically current than the copy at S. In this case, site S can delete its copy any time it desires and send an ABORT message to the other sites in the partition, concluding the protocol.

Case 2. Otherwise, site S simply continues the protocol in the usual way, except that S omits itself from the count of sites in the partition, if a distinguished partition exists. If no distinguished partition exists, or if S is the sole site with a physically current copy of the file, then S should not delete its copy of the file yet and should try this procedure again later.

If site S succeeds under either case, it may delete its copy of the file. The site must not participate in further updates to the deleted file, of course. For this reason, it is desirable that S notify all other sites having the copies that it no longer maintains a copy and should be excluded from their future attempts at forming a distinguished partition.

5. PROOF OF CORRECTNESS

THEOREM 1. *Dynamic voting and dynamic-linear are correct, that is, they maintain consistency of a replicated file.*

PROOF. We give the proof for dynamic voting. The extension for dynamic-linear is trivial.

To show correctness of a replica control algorithm, one must show that it produces one-copy serializable logs [3, 10]. However, every pessimistic algorithm with all reads from the current copy produces such logs [30], so it suffices to show *that dynamic voting is pessimistic and that reads are from the current* copy. That is, we must show that dynamic voting (1) permits at most one distinguished partition at any one time; (2) keeps all copies within a distinguished partition logically identical; and (3) insures that any two consecutive distinguished partitions 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.

For dynamic voting, as for ordinary voting, the set of sites with the most current logical version forms the single current distinguished partition. Consider an update from logical version number M to logical version number $M + 1$. First note that all k sites with logical version number M will have k as their update sites cardinality (see the **Do_Update** routine). (A formal proof of this claim is by induction on current logical version number.) To permit the update from M to $M + 1$, more than half of these k sites must be participating in the update (see the **Is_Distinguished** routine). After the update, fewer than half of these k sites will remain with logical version number M. We conclude that two updates in different partitions cannot be committed from logical version M. Thus the file versions form a sequence. Any two successive logical versions will have at least one copy in common; see the **Catch_Up** routine and note that step (3) of the **Is_Distinguished** routine requires that at least one participating site be physically current. This suffices to show that dynamic voting is a pessimistic algorithm with all reads from the current copy, and hence is correct. □

6. RELATIONSHIP OF OUR ALGORITHM TO OTHER PESSIMISTIC ALGORITHMS

Perhaps the most common pessimistic algorithm is *weighted voting* [27]. Here site k is given v_k votes. The distinguished partition is the partition, if any, containing sites whose votes sum to more than half of the total votes. *Simple voting* [50, 51, 46] is the instance of weighted voting in which each site receives a single vote. The *primary site* algorithm [2] is another instance of weighted voting: the primary site gets all the votes. *Voting-primary* (see [32], for example) is an extension of simple voting, applicable when the number of sites is even. Voting-primary uses one vote per site, except that the primary site gets two votes, to "break the tie" and distinguish two partitions each of which contains exactly half of the sites.

Weighted voting provides a rule for determining whether a partition is distinguished. A more general way to describe the partitions that form potential distinguished partitions is simply to list them. For example, the list $\{\{A, B, C\}, \{A, D\}, \{B, D\}, \{C, D\}\}$ says that if a partition contains sites A, B, and C, then that partition is a distinguished partition and can perform an update. Likewise, site D plus any of sites A, B, or C would also form a distinguished partition according to this list, as would any superset of the four groups of sites in the list. The correctness of the pessimistic algorithm places only one restriction on such a list: any two groups in the list must have a site in common, for if they did not, then updates could occur in two different partitions at the same time. (This is called the *intersection property.*) Furthermore, since supersets of a distinguished partition are themselves distinguished partitions, we can omit from the list all sets that are supersets of another set on the list. (This is called the *minimality property.*) Lists with the intersection and minimality properties were labeled *coteries* by Garcia-Molina and Barbara [5, 26]; their use was first suggested by Lamport [35]. Surprisingly, there are coteries that cannot be specified by a weighted vote assignment [26].

Coteries (and the special case of weighted voting) need a commit mechanism to discover what sites are participating in the update and to coordinate the update among those sites. The most popular choice is distributed two-phase commit, as is used in Section 4. Whatever mechanism coteries might use, dynamic voting and dynamic-linear can use the same mechanism. The difference between coteries (or weighted voting) and our dynamic algorithms is that our algorithms react to changing conditions of the network. We show in Section 8 that, under two stochastic models, this yields increased availability of the replicated file. Our algorithms achieve this by adding only a few extra bytes to each message.[4]

Other dynamic pessimistic algorithms have been proposed. In the remainder of this section, we briefly describe these other algorithms, explain how they differ from our algorithms, and offer an informal discussion of their advantages and disadvantages.

In the *true-copy token* algorithm [37], there is a token asociated with each file, and a partition is a distinguished partition if it contains a site that has the file token. The token may be passed from site to site. This algorithm is the dynamic analog of the primary site algorithm. It shares the drawback of the primary site algorithm that the available choices for potential distinguished partitions are quite limited.

The *missing writes* algorithm [19] makes use of the fact that read-only transactions can sometimes "run in the past" without compromising correctness. It uses the read-one, write-all algorithm under normal operation, but converts to weighted voting when failures occur. The algorithm is particularly useful if reads are frequent and failures are rare. It has two drawbacks. First, the weights it uses for voting in failure mode are fixed, while all the vote reassignment algorithms (including ours) allow these weights to change dynamically. Second, it requires many more data structures than our algorithms do. Several files must be maintained to keep track of updates that were made while the network is partitioned—the so-called "missing write" information—so they may be propagated to other sites later. These files need not be maintained during normal operation, but can grow rapidly during failures [17, p. 355].

[4] Using the coteries or weighted voting, one can choose to update only as many sites as it takes to form a distinguished partition, even if there are additional sites with which one can communicate. For example, simple voting with 99 sites could choose to update 50 of the sites, even if all 99 are in a single partition. The advantage of this is that it reduces the number of messages needed to do an update. The disadvantage is that many sites will have out-of-date copies of the replicated file, which can degrade performance when they later need to be updated. We could do the same but choose not to. We assume in this section that all algorithms that use coteries or weighted voting choose to update all the sites with which they can communicate, if those sites form a distinguished partition.

Dynamic voting and the Davcev–Burkhard algorithm [15] both seek to reassign votes in such a way that sites with out-of-date copies receive no votes.[5] The algorithms are very similar in their selection of partitions in which updates can occur. In fact, if communication were instant and updates arrived frequently (see Section 8.3.1), then dynamic voting and the Davcev–Burkhard algorithm would permit updates in exactly the same partitions. In both algorithms, the reassignment occurs dynamically and automatically. However, the Davcev–Burkhard algorithm uses a special mechanism that monitors the network continuously. Whenever a failure or recovery of a node of communication link occurs, the new network status must be reflected instantaneously in special data structures at the sites. This mechanism is unrealistic and the data structures unduly complicated. Our algorithms do not require such monitoring. They detect partitioning simply by the failure of sites to respond promptly during the update itself. Inability to detect failures does not invalidate the correctness of our algorithms, although it may affect availability.

Group consensus vote reassignment [6–8] uses weighted voting to do the updates themselves. At various times (presumably upon noticing a failure or repair), the members of a distinguished partition (defined according to the current vote assignment) elect a coordinator [24]. The coordinator chooses a new vote assignment and installs it by using version numbers, just as our algorithms do. The choices for the new vote assignment are exactly the same as the choices available to a suitable generalization of our algorithms (see Section 7.4). There are two differences between our algorithms and group consensus. First, the simplest form of our algorithms (dynamic voting) uses just a single integer (the update sites cardinality) stored at each site to describe implicitly the new vote assignment. Second, we piggyback the vote reassignment onto the two-phase commit protocol already being used to perform the update itself, while group consensus uses a separate election protocol. Thus group consensus uses more messages than our algorithms do (because of its separate election protocol), while the messages of our algorithms are slightly longer than those of group consensus (because the messages contain the update sites cardinality, for instance). Also, group consensus does not automate the decision of when to do the vote assignment, while our algorithms do an implicit vote reassignment at each update. Finally, the use of an election algorithm to do the reassignment presents an additional opportunity for deadlock. The management of replicated data requires a two-phase commit protocol to perform the updates. So why not use two-phase commit to reassign votes at the same time the update is done? This is what our algorithms do.

In autonomous vote reassignment [6–8], each site can increase its own votes. Site S does this by first notifying other sites of its vote increase. These other sites record the new vote assignment for S as soon as they hear about it and send an acknowledgement back to S. Site S records its new vote assignment after it receives acknowledgements from a majority of sites, where the "majority" is determined according to the vote assignment currently in force at S. As with group consensus, the failure to piggyback the vote reassignment to the two-phase commit already being used for updates is the source of several drawbacks. First, extra messages are required for the vote reassignment. If some sites that form a

[5] Before our use of the phrase, Davcev and Burkhard labeled their algorithm, "dynamic voting."

distinguished partition all wish to change their votes, they must each communicate with each other. Thus autonomous vote reassignment adds $O(n^2)$ messages per vote reassignment. Contrast this with the $O(n)$ messages that accomplish an update (and thus also a vote reassignment under our algorithms). The second drawback of autonomous vote reassignment occurs because the sites reassign votes autonomously, instead of using the information available from the two-phase commit used to do the update. This means that the vote assignments probably will not reflect the status of the network as clearly as they can in our dynamic voting. We emphasize that these drawbacks exist only because, for management of replicated data, there already is a need for two-phase commit to do the updates. For other applications, the autonomous nature of autonomous vote reassignment may prove very useful.

The *views* approach of El Abbadi and Toueg [22] is a generalization of the *virtual partitions* algorithm [20] and the algorithm in [21]. (The algorithm in [20] was perhaps the first to suggest vote reassignment. However, it required a two-phase protocol for the reassignment, a flaw corrected in the later versions.) The views algorithm provides dynamic read quorums as well as write quorums. (See the discussion in Section 7.5 of how our algorithm can be extended to include read quorums.) The notable advantage of the views algorithm is that it sometimes permits reads to "run in the past" without compromising correctness. For example, the following is possible under the views algorithm but not under our dynamic voting: a group of sites has all the votes; that group is using read-one, write-all; yet a vote reassignment can occur even if some of the sites in that group fail. The disadvantages of the views algorithm are as follows. First, each vote reassignment involving a fixed fraction of the sites generates $O(n^2)$ messages that our algorithms avoid by piggybacking them onto the updates. Second, the vote reassignment provides additional opportunity for deadlock. Finally, the views algorithm will never allow an update in a partition that contains fewer than one-fourth of the sites. This stands in contrast to dynamic-linear, which will permit an update in a partition that contains only a single site, if updates arrive frequently and sites fail one by one.

Voting with witnesses [38, 39] is an interesting idea that can be applied to all of the vote reassignment algorithms, including our own. Here certain sites are *witnesses*: they have full voting rights but do not maintain a physical copy of the replicated file. Used with simple voting, this means that a distinguished partition requires a majority of votes plus a nonwitness whose copy is as current as any of those in the partition. The hope is that inexpensive witnesses can replace ordinary sites with only a small degradation in availability [18]. To incorporate witnesses into our algorithms, one simply creates sites whose physical version number (as described in Section 4.1) is fixed at $-\infty$.

7. OPTIONS AND EXTENSIONS

Dynamic voting has two essential features. First, it uses the version number to make obsolete the old assignment of votes. In particular, it assigns no votes to sites with old copies of the file. Second, it uses the update sites cardinality to count the number of sites that have votes, that is, those sites whose copy of the file is up-to-date. Dynamic-linear adds a third feature: the distinguished site entry to break the tie when the distinguished partition splits into two equal-sized pieces. The detailed description of the algorithm in Section 4 specified certain facets of the algorithms that could easily have been handled in other ways. One of these is our choice to distinguish the physical state of a copy, which reflects the updates actually processed, from the logical state of the copy, which reflects the updates that the copy has agreed to process. This section makes explicit certain other options we have chosen and offers several extensions to the algorithms.

7.1 When to Do New and Missing Updates

We could have easily incorporated the **Send_Missing_Updates** procedure inside the **Do_Update** procedure. However, we have chosen not to do so in order to make the window of uncertainty shorter. The example in Section 3.3 provides an instance in which the shortened window of uncertainty permitted updating to continue that would otherwise have been blocked. Indeed, we could have narrowed the window of uncertainty even further by treating the new update as a missing update. That is, the transmission of the new update to subordinates in *Physical* could have been removed from the **Do_Update** procedure and placed in the **Send_Missing_Updates** procedure instead. This seems unlikely to be an efficient choice, however, since it fails to piggyback the update to the vote reassignment. We could also have removed the **Catch_Up** procedure from the window of uncertainty. Or we could have chosen to have the coordinator abort and release all locks, do a **Catch_Up**, and then start the protocol again. The optimal choice for all of these concerns depends on the details of the local installation.

7.2 When Should a Recovered Site Instigate a New Distinguished Partition?

When a site recovers from a failure, it certainly should run the **Make_Current** procedure to obtain missing updates. This procedure uses no locking and hence is not bothersome. To regain its vote in the distinguished partition, however, the site must either wait for an update to come along in its partition, or run the **Rejoin_Distinguished_Partition** procedure (that is, create a null update and run the normal-operation protocol). The advantage of doing the latter is that the new distinguished partition will more accurately reflect the actual state of the network, which may increase availability. The disadvantage of running this procedure is its expense. It requires locking and thereby increases the likelihood of deadlock. If the recovered site obtains the missing updates but regains its vote by simply waiting for a new update to arrive in the partition, no extra locking occurs.

7.3 Weighted Dynamic Voting

In the simplest form of static voting, each site has a single vote. In weighted voting, site S_k has v_k votes and a partition P is the distinguished partition if the sum of the votes in P is more than half of the sum of the votes of all sites. The idea of weighted voting extends directly to dynamic voting and dynamic-linear, with no additional data structures or messages, as follows. Each site S_k is assigned v_k votes. Each site knows how many votes each other site is assigned. When a new distinguished partition is formed, the update sites cardinality is set to the sum of the preassigned votes of the sites in the new partition (instead of just the number of such sites). To determine if a certain group of sites (all with the same logical version number) forms a distinguished partition, one sums the preassigned votes of the members of the group (instead of just counting how many sites are in the group). The group forms a distinguished partition if this sum is more than half of the update sites cardinality of these sites. If the update sites cardinality is even, one can use a distinguished site entry to "break the tie" when the sum is exactly half of the update sites cardinality, just as in dynamic-linear.

The advantage of weighted static voting over simple static voting is that one can tailor the votes to suit characteristics of individual sites and links in the network. For example, one might want to assign more votes to sites that are particularly reliable, or to sites that have a high degree of connectivity. The advantage of weighted dynamic voting over dynamic voting lies likewise in the additional flexibility of the former algorithm. What vote assignment is optimal in a heterogeneous network, and how much advantage one obtains by such optimization are unresolved questions even for weighted static voting, much less for weighted dynamic voting.

Note that although the preceding scheme permits an arbitrary initial assignment of votes, it does *not* permit arbitrary reassignment of votes. If a site begins with, say, five votes, then it will always have either five votes (if its copy is logically up-to-date) or none (if its copy is old). The voting power of a site with an up-to-date copy of the file, relative to the other sites, depends on which other sites have up-to-date copies.

7.4 Dynamic Coteries

The weighted dynamic voting algorithm described in the preceding subsection is the best one can do with the data structures given (logical version number, physical version number, update sites cardinality, and distinguished site entry). The role of the logical version number is to make obsolete the old vote assignment when a new distinguished partition forms. By having more elaboration data structures, one can obtain any vote reassignment desired (and simplify the algorithm as well). As an extreme, suppose one carries with each file in addition to its version number, the entire coterie, that is, the list of potential distinguished partitions.[6] By doing so, we need not have two version numbers—physical and logical. When an update arrives at a site, the site uses the normal operation protocol to learn which sites are in its partition P, as usual, plus the coterie of the sites whose version number is maximal. The partition P is a distinguished partition if some set in the coterie is a subset of P. When an update is committed, the coordinator of the update selects a new coterie and sends the new coterie to each site in the partition processing the update. The coordinator can use whatever reasoning it wishes and whatever information it has available to select the new coterie. In particular, the coordinator can give sites outside its partition voting power or not, as it sees fit. Whenever a site S finds that another site S' has a copy with a larger version number, S asks S' for the missing updates, and S replaces its version number and coterie by those of S'.

[6] Alternatively, each site can maintain a table with all possible coteries and pass around the coterie index in place of the coterie.

The above algorithm is the dynamic analogue of coteries. Just as coteries permit maximal flexibility in the static selection of potential distinguished partitions, dynamic coteries permits maximal flexibility in the dynamic selection of potential distinguished partitions.

7.5 Read Requests

We have chosen to focus on update requests in this paper instead of distinguishing between read and write requests. There are two ways to augment our algorithms to include separate read and write quorums. First, each site can maintain a static table that indicates the read and write quorums for each possible value of the update sites cardinality. These tables (which are bounded in size by the number of sites) would be maintained along with the tables that implement weighted dynamic voting, as described in Section 7.3. For instance, suppose the update sites cardinality is 9 and each site has 1 vote. The read quorum for this situation might be 3, with a write quorum of 7. As usual, the read and write quorums must sum to more than the number of sites involved. Here, that means the read and write quorums for update sites cardinality SC must sum to more than SC.

The second way to incorporate read and write quorums is more dynamic. Each site maintains not only the update sites cardinality (or whatever structure maintains the current vote assignments), but also two more integers: the current read and write quorums. At each update, the coordinator sets these however it chooses, but with the sum of the quorums greater than the update sites cardinality.

8. STOCHASTIC ANALYSIS OF THE DYNAMIC AND STATIC ALGORITHMS

8.1 Why Do a Stochastic Analysis?

Recall that simple voting is weighted voting with each site receiving a single vote and that voting-primary is the same as simple voting except that one site (the primary site) gets two votes if there are an even number of sites. It is clear that voting-primary *dominates* simple voting in this sense: If an update request can be accommodated by simple voting, then that same request can also be accommodated by voting-primary. Dynamic-linear dominates dynamic voting in the same way. However, no single algorithm dominates, in the above sense of the word, the class of all static algorithms. Further, no single algorithm dominates the class of all dynamic algorithms. That is to say, a sequence of failures, repairs, and update requests can occur in such a way that dynamic voting (for instance) can accommodate requests that simple voting (for instance) cannot, and vice versa. For example, suppose a five-site network splits into two partitions,

$$A \;\; B \;\; C \mid D \;\; E$$

and an update appears at site A. Simple voting, dynamic voting, and dynamic-linear all permit the update to be processed. If site C thereafter forms a third partition by itself,

$$A \;\; B \mid C \mid D \;\; E$$

the two dynamic algorithms permit subsequent updates arriving at sites A or B to be processed, while simple voting rejects all update requests. On the other hand, if the two-site A–B partition now splinters into two single-site partitions, while site C joins sites D and E,

$$A \mid B \mid C \; D \; E$$

simple voting would permit subsequent updates arriving at sites C, D, or E to be processed, while dynamic-linear would permit only those updates arriving at the single site A, and dynamic voting would reject all update requests in this state. If site A then fails, only simple voting permits updates to be processed.

The preceding example shows that at some times under some scenarios one algorithm is better, while at other times under other scenarios another algorithm is better. 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*?

In Sections 8.3 and 8.4 we develop two stochastic models to make precise what is meant by the phrase "more likely" in the preceding paragraph. In the first of these two models, sites are subject to failure and repair but communication links are infallible, while the reverse is true in the second model. The first model will permit proof of a stronger result. In both models we compare dynamic voting and dynamic-linear to the class of all static algorithms. Both models demonstrate the superiority of dynamic-linear over all static algorithms.

8.2 Two Measures of Availability

There are many measures by which one might evaluate and compare the virtues of pessimistic algorithms for maintaining the consistency of a replicated database. Deterministic measures like *node vulnerability* and *edge vulnerability* [5] are useful as worst case indicators. The *reliability* of a pessimistic algorithm tells how long, on average, the algorithm will permit updates before it reaches a halted state in which no updates are permitted. Note that all pessimistic algorithms must reach such a state if sites are subject to failure and partitioning is possible [43,48]. However, reliability does not indicate how long the algorithm will remain in the halted state. A more meaningful stochastic measure is the *availability* of the algorithm: the steady-state probability that an update request will be satisfied. The higher the availability, the better the algorithm.

The standard measure of availability (see [1], [7], [39], and [47], 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 within the distinguished partition. For example, suppose that we are using simple voting in a system of 11 sites, of which 3 sites are down during a certain time period. Under the traditional definition, the availability of the system during this time period is 1.0, since more than half of the sites are available throughout the time period. Under the alternative measure, with the further assumption that any given update has probability $\frac{1}{11}$ of arriving at any given site, the availability during this period is $\frac{8}{11}$, since only this fraction of the updates succeed (by arriving at a functioning site).

The traditional measure is appropriate if updates are viewed as arriving at the *system*; the alternative measure is more appropriate if updates arrive at an individual *site*. We term these two measures *system availability* and *site availability*, respectively.

It is our view that the nontraditional measure, site availability, is by far the more meaningful measure of availability for a replicated file in a network subject to partitioning. To see this, imagine first a network whose links fail (so that partitioning is possible), but whose sites do not fail. In such a network, the simple primary-site algorithm has system availability 1.0, since the primary site is always available for updates. Yet no one would claim that this trivial fact has established that the primary-site algorithm is an optimal (indeed, perfect) algorithm in such a network! For networks whose sites can fail, the argument is less strong but still persuasive. Given a choice between a single-site distinguished partition and a multisite distinguished partition, we would all choose the latter, if updates arrive uniformly.

We report results for site availability, but also mention in passing how matters change if one uses the traditional measure (system availability) instead. Each formula in this paper requires only a trivial modification to use the traditional measure. Interestingly, the two measures do not coincide in their advice, as we will see in Section 8.3.5. However, use of the traditional measure only *increases* the superiority of our two dynamic voting algorithms over static algorithms. That is, the measure of availability that we use in this paper shows our dynamic voting algorithms in their least favorable light. This is a consequence of the following theorem.

THEOREM 2. *Assume that the update requests arrive at sites uniformly, that is, any given update request has probability $1/n$ of arriving at any given site, independently of whatever partitioning is occurring, where n is the number of sites. Let X be any pessimistic algorithm, static or dynamic. If X has greater* site *availability than simple voting, then X also has greater* system *availability than simple voting. The same statement remains true if "simple voting" is replaced by "voting-primary."*

PROOF. Suppose that X has greater site availability than simple voting. This means that the difference between the two algorithms' site availabilities is positive. Another way to express this statement is to sum over disjoint cases whose union is the entire sample space:

$$\sum_{j=1}^{n} \frac{j}{n} \Pr(X \text{ has a size } j \text{ dist. part. and Voting has no dist. part.})$$

$$- \sum_{k=0}^{n} \frac{k}{n} \Pr(X \text{ has no dist. part. and Voting has a size } k \text{ dist. part.})$$

$$+ \sum_{j=1}^{n} \sum_{k=1}^{n} \frac{j-k}{n} \Pr(X \text{ has a size } j \text{ dist. part. and Voting has a size } k \text{ dist. part.}) > 0$$

where "Voting" means "simple voting" and "dist. part." stands for "distinguished partition." The first assumption in the statement of the theorem (uniform arrival process) justifies the j/n and k/n factors that reflect the probabilities the update arrives at the given size j and size k partitions, respectively. Site availability is differentiated from system availability precisely by the presence of these factors.

Simple voting has a distinguished partition if and only if there is a partition that contains more than half of the sites. Thus the first sum above, in which simple voting has no distinguished partition, runs only up to $\lfloor n/2 \rfloor$. Similarly, the second sum above runs from $\lfloor (n/2) + 1 \rfloor$. In the third term above (the double sum), both X and simple voting have a distinguished partition. Again, the distinguished partition for simple voting is at least half of the sites, so the distinguished partition for X cannot be any larger than simple voting's partition. Thus each entry in the third term above (the double sum) is nonpositive and hence can be discarded. All this yields

$$\sum_{j=0}^{\lfloor n/2 \rfloor} \frac{j}{n} \Pr(X \text{ has a size } j \text{ dist. part. and Voting has no dist. part.})$$
$$> \sum_{k=\lfloor n/2 \rfloor+1}^{n} \frac{k}{n} \Pr(X \text{ has no dist. part. and Voting has a size } k \text{ dist. part.})$$

Now replace the coefficients of each of the left-hand-side terms by their upper bound and replace the coefficients of each of the right-hand-side terms by their lower bound to see that

$$\left\lfloor \frac{n}{2} \right\rfloor \Pr(X \text{ has a dist. part. and Voting has no dist. part.})$$
$$> \left\lfloor \frac{n}{2} + 1 \right\rfloor \Pr(X \text{ has no dist. part. and Voting has a dist. part.})$$

from which the result follows. (The difference in the system availabilities is just the unweighted difference between the above two probabilities.) The proof for voting-primary is the same as the above proof for simple voting, except that $\lfloor (n/2) + 1 \rfloor$ is replaced by $\lfloor n/2 \rfloor$. □

One other measure for pessimistic algorithms deserves mention. We have been assuming that rejected updates are aborted; in this case, availability is the most meaningful measure. Ahamad and Ammar [1] have suggested a different measure to be used if rejected updates wait until the site at which they arrive joins a distinguished partition. They suggest that, for this case, one can use the mean response time for updates. Although their measure is reasonable, it appears difficult to obtain analytic expressions for it. We consider only availability in this paper.

8.3 The Site Model

8.3.1 *Specification of the Model.* In this subsection we state the first of our two stochastic models, the *site model.* The following are key aspects of the site model: no attempt is made to model changes to the network topology (sites fail and are repaired but links are infallible); updates are assumed to be frequent relative to site failures and repairs; and communication delays in the commit protocol are ignored. The next three subsections show how to compute the site availabilities of simple voting/voting-primary, dynamic voting, and dynamic-linear. We then state and prove two theorems that compare the site availabilities of our two dynamic algorithms against that of static algorithms. The main result is this: *under the assumptions of the site model, dynamic-linear provides greater site availability than any static algorithm if there are four or more sites.*

We now introduce the assumptions of the site model. The first four assumptions duplicate assumptions that Pâris uses to analyze the availability of his *voting with witnesses* scheme [38, 39]. The fifth assumption, however, causes our model to deviate from his. Below are the assumptions; their justification comes next.

(1) 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.

(2) The failures at the various sites form independent Poisson processes with failure rate λ. For any given site that is up (functioning), the probability that it does down (fails) at or before the next t time units is $1 - e^{-\lambda t}$.

(3) Similarly, the repairs at the various sites form independent Poisson processes with repair rate μ.

(4) Updates are instantaneous. We ignore communication delays in the commit protocol.

(5) Updates arrive frequently: 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 remark that the algorithms require none of these assumptions to operate properly. The assumptions are made only to provide a model whose analysis is tractable.

The first assumption seems rather odd: the model prohibits precisely the phenomenon—partitioning—that the algorithms in this paper are designed to tolerate! We make this assumption in order to sidestep the countless network topologies that might occur as links fail. Lest our results appear to hinge on this admittedly odd assumption, we present in Section 8.4 a second stochastic model that does allow partitioning.

As Pâris notes [39, p. 608], the third assumption is less reasonable than the second, but both are necessary if we wish to model the network's behavior by a Markov process. The fourth assumption (instantaneous updates) is another simplification necessary to maintain the Markovian model. Because we are interested in a comparison of static versus dynamic voting algorithms, and because both classes of algorithms face similar communication delays in the commit protocol, the fourth assumption does not seem unreasonable.

On the other hand, the fifth assumption is not necessary to force a Markovian model. Indeed, the assumption presents a bias in favor of our dynamic algorithms, since it permits them to prepare themselves for the next failure. In either of its forms, the fifth assumption permits great reduction in the number of states in the Markov process that describes the network's behavior. This simplifies the analysis and led to the discovery of Theorem 3 in Section 8.3.5. Further, in many applications updates arrive frequently with respect to both failures and repairs.

We justify the utility of the site model on three accounts. First, the site model properly emphasizes the features of the network most important to a comparison of the availabilities of static and dynamic voting algorithms, namely, failures and repairs. For purposes of comparison, we believe that ignoring link failures is reasonable. This belief is supported by the fact that the results under the link model, in which link failures are permitted, are qualitatively the same as the results under the site model. Second, all the assumptions of the site model are either reasonably true-to-life (e.g., the frequent update assumption), or treat static and dynamic algorithms equally. For example, the communication needs of the static and dynamic voting algorithms are quite similar, so treating communication as instantaneous is fair for purposes of comparison. Finally, we note that no other analytic results that compare static and dynamic algorithms with more than five sites have been obtained under any model even as sophisticated as the site model.

Because the site model treats updates as if they were instantaneous, a site's physical version number will always equal its logical version number. Thus we can ignore the distinction between logical and physical version numbers. This is the case for simple voting and voting-primary as well as for our dynamic algorithms. Throughout the analysis of the site model, we will speak simply of "the version number."

Under the assumptions of the site model, our dynamic voting algorithm is available for updates exactly when the Davcev–Burkhard algorithm is available. Hence the analysis of dynamic voting in this section applies equally well to the Davcev–Burkhard algorithm.

8.3.2 *Analysis of Simple Voting and Voting-Primary.* The mean time to failure of a functioning site is $1/\lambda$. The mean time to repair of a failed site is $1/\mu$. It follows that for the Poisson process describing the behavior of the sites, the probability that any given site is up to any particular time is

$$\frac{1/\lambda}{1/\lambda + 1/\mu}, \quad \text{that is,} \quad \frac{\mu}{\lambda + \mu}.$$

The well-known site availability of simple voting is

$$\sum_{k=\lfloor n/2 \rfloor + 1}^{n} \frac{k}{n} \binom{n}{k} \left[\frac{\mu}{\lambda + \mu}\right]^{k} \left[\frac{\lambda}{\lambda + \mu}\right]^{n-k}$$

where n is the number of sites in the network. The k/n term in the summation reflects the fact that an update request can be processed only if the site at which the request arrives is one of the k sites in the distinguished partition. Note that we need not assume that update arrivals are uniformly distributed over the sites, since voting and the site model treat the sites uniformly.

The voting-primary algorithm retains a majority when exactly half of the sites are functioning, if the functioning sites include the primary site. Thus the site availability of voting-primary is exactly the same as the site availability of simple voting when there are an odd number of sites and contains the additional term

$$\frac{1}{4}\binom{n}{n/2}\left[\frac{\mu}{\lambda+\mu}\right]^{n/2}\left[\frac{\lambda}{\lambda+\mu}\right]^{n/2}$$

if n is even. Here we are assuming that the primary site is selected at random, thereby avoiding any reliance upon assumptions about the sites to which updates arrive.

These same formulas could also be obtained by drawing the state diagram for the birth–death process that describes the number of failed sites and solving the resulting balance equations. We use just such a procedure to analyze the dynamic voting algorithms.

8.3.3 *Analysis of Dynamic Voting.* The system begins with all n sites in the distinguished partition. Eventually one site fails. Our fifth assumption insures 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 cardinalities 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 two sites in the distinguished partition and a failure then occurs. The subsequent update is rejected—one out of two sites is not a majority. Of these two sites in the most recent distinguished partition, call the one still up site U and the one now down site D. From this state, one of three events can occur.

(1) Site D might be repaired. The two-site distinguished partition is restored and the action of the network continues in the fashion described thus far.

(2) One or more of the other $n - 2$ failed sites might be repaired. If sometime later site D is repaired and an update arrives, the functioning sites include both (hence a majority) of the sites with up-to-date copies of the file. In this case, however, the newly formed distinguished partition will also include the other sites that have meanwhile been repaired.

(3) Site U might fail. Now both site U and site D 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 Figure 1. State (X, Y, Z) is the state in which

(1) the update sites cardinality of each up-to-date copy of the file is Y;

(2) X of the Y sites with update sites cardinality Y are up;

(3) 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.

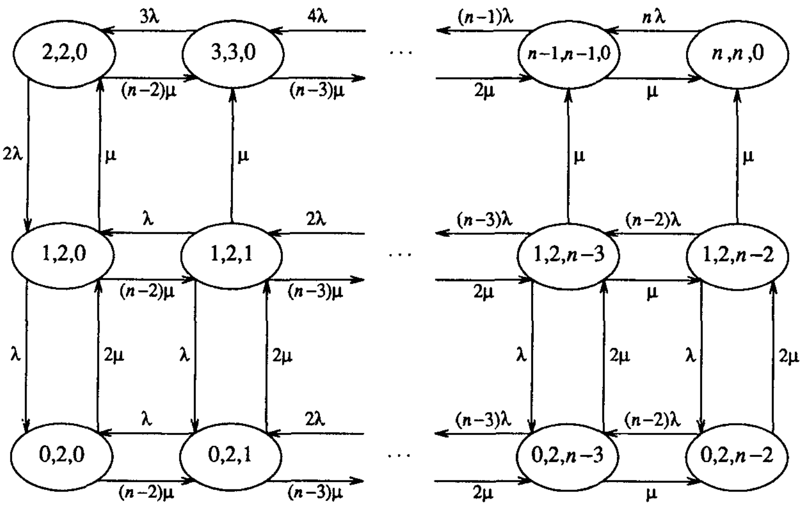

Fig. 1. The state diagram for dynamic voting.

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 Figure 1. Label the top-row states $A_0, A_1, \ldots, A_{n-2}$, from left to right. Label the middle-row and bottom-row states B_0 through B_{n-2} and C_0 through C_{n-2}, respectively, again left to right. In an abuse of notation, we also let A_k denote the steady-state *probability* of state A_k, and similarly for the B_k and C_k. The site availability of the network is

$$\sum_{k=0}^{n-2} \frac{k+2}{n} A_k.$$

The $(k+2)/n$ term reflects the fact that the site at which the update request arrives must be one of the $k+2$ functioning sites in state A_k.

To find the steady-state probabilities of the states in Figure 1, set flow-out equal to flow-in to obtain the following balance equations, one equation per state. (See Trivedi [52], for example, for the justification for obtaining the balance equations from the state diagram.) The three leftmost states have special equations:

$$\begin{aligned}[2\lambda + (n-2)\mu]A_0 &= 3\lambda A_1 + \mu B_0\\ [\lambda + (n-1)\mu]B_0 &= \lambda B_1 + 2\mu C_0 + 2\lambda A_0\\ n\mu C_0 &= \lambda C_1 + \lambda B_0.\end{aligned}$$

For the remaining top-row states $(k = 1, 2, \ldots, n-2)$,

$$[(k+2)\lambda + (n-k-2)\mu]A_k = (k+3)\lambda A_{k+1} + (n-k-1)\mu A_{k-1} + \mu B_k.$$

For the remaining middle-row states $(k = 1, 2, \ldots, n-2)$,

$$[(k+1)\lambda + (n-k-1)\mu]B_k = (k+1)\lambda B_{k+1} + (n-k-1)\mu B_{k-1} + 2\mu C_k.$$

For the remaining bottom-row states $(k = 1, 2, \ldots, n-2)$,

$$[k\lambda + (n-k)\mu]C_k = (k+1)\lambda C_{k+1} + (n-k-1)\mu C_{k-1} + \lambda B_k.$$

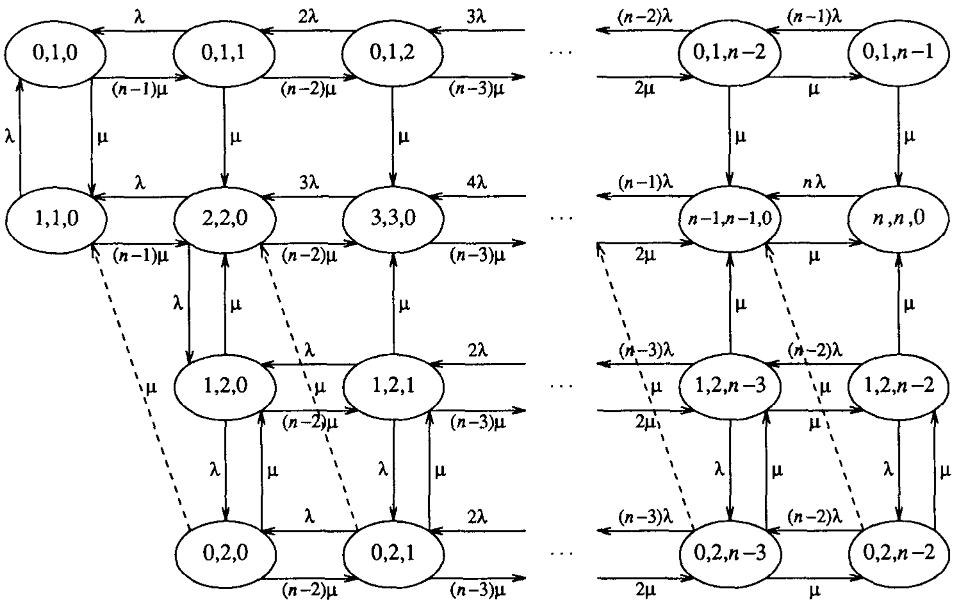

Fig. 2. The state diagram for dynamic-linear.

By defining $A_{n-1} = B_{n-1} = C_{n-1} = 0$, the above equations are seen to be correct for the right-boundary states.

One of these $3n - 3$ equations is redundant. Replace it by the equation that says the probabilities sum to one:

$$\sum_{k=0}^{n-2} (A_k + B_k + C_k) = 1.$$

We have not found a simple expression for the solution to the above system of equations. However, for fixed n and repair/failure ratio μ/λ, 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 MACSYMA® or MAPLE [13] can solve the system in terms of the repair/failure ratio μ/λ. Results for *arbitrary* n are given by Theorem 3 in Section 8.3.5.

8.3.4 *Analysis of Dynamic-Linear.* The analysis of dynamic-linear is similar to that for dynamic voting. Figure 2 shows the state diagram for dynamic-linear. It uses the same notation as was used in the figure for dynamic voting. State $(n, n, 0)$ is the start state: all n sites belong to the distinguished partition and all of them are up. As sites fail, the system moves to state (2, 2, 0) just as in dynamic voting. The novelty of dynamic-linear appears if a failure occurs from state (2, 2, 0). Suppose the site remaining up (call it site X) is the site named in the distinguished site entry. Then a distinguished partition still exists; the system

® MACSYMA is a trademark of Symbolics, Inc. It was originally developed by the Mathlab Group of the MIT Laboratory for Computer Science.

moves to state (1, 1, 0). That is, the single site X forms the distinguished partition, site X is up, and no other site is up. If site X now fails, the new state is (0, 1, 0): no site in the distinguished partition is up, site X alone still forms the distinguished partition, and no other sites are up. The states in the top row of Figure 2 are states that occur while site X remains down and other sites are repaired or fail. The second-row states of Figure 2 are the states for which the system is available for updates. The third and fourth rows of Figure 2 are analogous to the bottom two rows of the figure for dynamic voting (Figure 1).

An update request will be accepted under dynamic-linear if the network is in any of the states on the second row of Figure 2. The probability the system is in one of the second-row states can be found by setting flow-in equal to flow-out for each state and solving the resulting balance equations, just as for dynamic voting.

8.3.5 *Analytic Comparison of the Availabilities.* Throughout this subsection, we assume that the repair rate μ is larger than the failure rate λ, in other words, that sites are more likely to be up than down. This assumption is quite natural, and without it, the performance of some of the algorithms would degrade as the number of sites increases. We give results only for three or more sites since the algorithms reduce to trivial or nonsense algorithms when there are fewer sites.

Let *Voting*, *Voting-Primary*, *Dynamic*, and *Dynamic-Linear* denote the site availabilities of the algorithms they name, respectively. (*Voting* means simple voting here.) The first two of the following comparisons between the availabilities are immediate, and the third is easily argued:

(1) *Voting-Primary* = *Voting* when there are an odd number of sites;
(2) *Voting-Primary* $\geq$ *Voting*;
(3) *Dynamic-Linear* > *Dynamic*.

Theorem 3 gives the remaining comparisons of the availabilities of these algorithms. Theorem 4 extends the comparison to include the class of *all* static algorithms.

THEOREM 3. *Suppose that the repair/failure ratio* μ/λ *is greater than* 1.0. *The following statements hold for the site model.*

When there are exactly 3 *sites,*

Voting-Primary = *Voting* > *Dynamic-Linear* > *Dynamic.*

When there are exactly 4 *sites, if* $\mu/\lambda < 2.3292$ (*approximately*), *then*

Dynamic-Linear > *Voting-Primary* > *Dynamic* > *Voting*;

otherwise,

Dynamic-Linear > *Dynamic* > *Voting-Primary* > *Voting.*

When there are exactly 5 *sites, if* $\mu/\lambda < 1.3070$ (*approximately*), *then*

Dynamic-Linear > *Voting-Primary* = *Voting* > *Dynamic*;

otherwise

Dynamic-Linear > *Dynamic* > *Voting-Primary* = *Voting.*

When there are 6 *or more sites,*

$$\textit{Dynamic-Linear} > \textit{Dynamic} > \textit{Voting-Primary} \geq \textit{Voting}.$$

PROOF. Our proof for 3, 4, 5, or 6 sites is by explicit, symbolic calculations performed by MACSYMA. The proof for 7 or more sites proceeds by showing that *Dynamic* > *Voting-Primary*; the other two inequalities then follow.

Let us first do the case when n (number of sites) is odd, so that *Voting-Primary* is simply *Voting*. Return to Figure 1, the state diagram for dynamic voting. As before, label the top-row states $A_0, A_1, \ldots, A_{n-2}$, from left to right. Label the middle-row and bottom-row states B_0 through B_{n-2} and C_0 through C_{n-2}, respectively, again left to right. The system is available under dynamic voting in states A_0 through A_{n-2}. The system is available under simple voting if and only if more than half the sites are up. State A_k has $k + 2$ sites up; state B_k has $k + 1$ sites up; and state C_k has k sites up. Thus the system is available under simple voting if and only if it is one of states A_{h+1} through A_{n-2}, B_{h+2} through B_{n-2}, or C_{h+3} through C_{n-2}, where h is the "halfway" point $(n - 1)/2 - 2$.

In an abuse of notation, we also let these symbols denote the steady-state *probabilities* of the respective states. Then for odd n,

$$\textit{Dynamic} > \textit{Voting-Primary}$$

if and only if

$$\sum_{k=0}^{h} \frac{k+2}{n} A_k > \sum_{k=h+2}^{n-2} \frac{k+1}{n} B_k + \sum_{k=h+3}^{n-2} \frac{k}{n} C_k. \tag{1}$$

Note that we are using site availability, not system availability.

If we could solve the balance equations in Section 8.3.3 to obtain simple expressions for the A_k, B_k, and C_k, this proof might be easy. Such expressions are not apparent. This proof will show the truth of inequality (1), without using any explicit expression for the solution of the balance equations. Instead, we obtain and use the following four relations from the balance equations, where the first two are each valid for any m between 1 and $n - 2$, inclusive.

$$\lambda(m + 2)A_m = \mu(n - m - 1)A_{m-1} + \mu \sum_{k=m}^{n-2} B_k \tag{2}$$

$$\lambda m(B_m + C_m) = \mu(n - m - 1)(B_{m-1} + C_{m-1}) - \mu \sum_{k=m}^{n-2} B_k \tag{3}$$

$$2\lambda A_0 = \mu \sum_{k=0}^{n-2} B_k \tag{4}$$

$$2\mu \sum_{k=0}^{n-2} C_k = \lambda \sum_{k=0}^{n-2} B_k. \tag{5}$$

Equation (2) is obtained by summing the balance equations associated with states A_k for k from m to $n - 2$. Equation (3) is obtained similarly by summing the balance equations associated with states B_k and C_k for k from m to $n - 2$. Equation (4) is the sum of the equation associated with A_0 and equation (2) when m is 1. The final equation (5) is obtained by summing the balance equations associated with states C_k for k from 0 to $n - 2$. These equations will be sufficient for our needs.

By conducting extensive but simple algebraic manipulations to inequality (1), including the application of equations (2) and (4) to its left-hand side and equation (3) to its right-hand side, one can show that inequality (1) is equivalent to the following inequality:

$$\begin{aligned}&\sum_{k=0}^{h-1} (n - k - 2)A_k + \sum_{k=0}^{h} (k + 1)B_k - B_{h+1} + \sum_{k=h+2}^{n-2} \left[2k - n + 2 - \frac{\lambda}{\mu}\right]B_k \\ &\quad > \sum_{k=h+1}^{n-3} (n - k - 2)C_k - (h + 2)\frac{\lambda}{\mu} C_{h+2}\end{aligned} \tag{6}$$

where h is again the "halfway" point $(n - 1)/2 - 2$.

We now apply six approximations. That is to say, after we do the following manipulations to inequality (6) and then show the truth of the resulting inequality, we will have shown the truth of inequality (6) itself, hence also the truth of inequality (1), hence the result.

(1) Use only the first term of $\sum_{k=0}^{h-1} (n - k - 2)A_k$, that is, use $(n - 2)A_0$. Note that we are assuming here that $n \geq 7$, for if not, h would be 0 and the sum empty. Transform $(n - 2)A_0$ into

$$(n - 2)\frac{\mu}{2\lambda} \sum_{k=0}^{n-2} B_k$$

by using equation (4).

(2) Discard the $\sum_{k=0}^{h} (k + 1)B_k$ term.

(3) Replace the $-B_{h+1}$ term by $-\sum_{k=0}^{n-2} B_k$.

(4) Discard the

$$\sum_{k=h+2}^{n-2} \left[2k - n + 2 - \frac{\lambda}{\mu}\right]B_k$$

term. This is an acceptable approximation because each term in the sum is nonnegative, since $\lambda/\mu \leq 1$.

(5) Note that

$$\begin{aligned}\sum_{k=h+1}^{n-3} (n - k - 2)C_k &\leq \frac{(n-1)}{2} \sum_{k=h+1}^{n-3} C_k \\ &\leq \frac{(n-1)}{2} \sum_{k=0}^{n-2} C_k \\ &\leq \frac{(n-1)}{2} \frac{\lambda}{2\mu} \sum_{k=0}^{n-2} B_k\end{aligned}$$

where this last step is an application of equation (5) above. Replace the left-hand side of this string of inequalities by the right-hand side.

(6) Discard the $-(h + 2)(\lambda/\mu)C_{h+2}$ term.

The result of applying these approximations to inequality (6) is the following inequality:

$$(n-2)\frac{\mu}{2\lambda}\sum_{k=0}^{n-2} B_k - \sum_{k=0}^{n-2} B_k \geq \frac{(n-1)}{2}\frac{\lambda}{2\mu}\sum_{k=0}^{n-2} B_k \tag{7}$$

That is, showing that inequality (7) holds is a sufficient (but not necessary) condition for inequality (6) to hold. The above approximations were carefully selected to permit the next key step: divide through by $\sum_{k=0}^{n-2} B_k$. After rearranging terms, this yields the equivalent expression

$$2(n-2)(\mu/\lambda)^2 - 4\mu/\lambda - n + 1 \geq 0. \tag{8}$$

For $\mu/\lambda = 1$, this reduces to $n \geq 7$. Since the left-hand side of inequality (8) is an increasing function of μ/λ when $n \geq 3$ and $\mu/\lambda \geq 1$ (just take the partial derivative with respect to μ/λ), we are done for odd $n \geq 7$.

The preceding was under the assumption that n is odd. The proof for even n involves similar manipulations, to which we now turn. For even n, inequality (1) has an additional term

$$\frac{1}{2}\left[\frac{h+2}{n}A_h + \frac{h+2}{n}B_{h+1} + \frac{h+2}{n}C_{h+2}\right]$$

on its right-hand side, where h is now defined to be $(n/2) - 2$. By using equations (2), (3), and (4), along with extensive algebraic manipulation akin to that in the proof for odd n, one reveals the following analogue to inequality (6); showing its truth will complete the proof.

$$\begin{aligned}
&\sum_{k=0}^{h-2}(n-k-2)A_k + \sum_{k=0}^{h-1}(k+1)B_k - B_h \\
&\quad + \left[1 - \frac{\lambda}{\mu}\right]B_{h+1} + \sum_{k=h+2}^{n-2}\left[2k - n + 3 - \frac{\lambda}{\mu}\right]B_k \\
&\qquad > \sum_{k=h}^{n-3}(n-k-2)C_k - \frac{\lambda}{\mu}\left[(h+1)C_{h+1} + \frac{1}{2}(h+2)(A_h + B_{h+1} + C_{h+2})\right].
\end{aligned} \tag{9}$$

We next apply approximations. As before, use only the first term of

$$\sum_{k=0}^{h-2}(n-k-2)A_k$$

(this is legal because $n \geq 8$), transforming it by equation (4) into

$$(n-2)\frac{\mu}{2\lambda}\sum_{k=0}^{n-2} B_k.$$

Replace the $-B_k$ term by $-\sum_{k=0}^{n-2} B_k$. Discard the other three (nonnegative) terms on the left-hand side of inequality (9). Replace

$$\sum_{k=h}^{n-3}(n-k-2)C_k$$

by

$$\frac{n}{2}\frac{\lambda}{2\mu}\sum_{k=0}^{n-2} B_k$$

using equation (5) as before. Discard the other term on the right-hand side of inequality (9). All this yields

$$(n-2)\frac{\mu}{2\lambda}\sum_{k=0}^{n-2} B_k - \sum_{k=0}^{n-2} B_k \geq \frac{n}{2}\frac{\lambda}{2\mu}\sum_{k=0}^{n-2} B_k$$

that is,

$$2(n-2)(\mu/\lambda)^2 - 4\mu/\lambda - n \geq 0.$$

At $\mu/\lambda = 1$, this reduces to $n \geq 8$. Since the left-hand side of the above inequality is an increasing function of μ/λ when $n \geq 3$ and $\mu/\lambda \geq 1$ (just take the partial derivative with respect to μ/λ), we are done for even $n \geq 8$. □

THEOREM 4. *Assume that the update requests arrive at sites uniformly, that is, any given update request has probability* $1/n$ *of arriving at any given site, independently of whatever partitioning is occurring, where n is the number of sites. Assume that the repair rate* μ *is greater than the failure rate* λ. *Then under the site model dynamic-linear has greater site availability than* any *static algorithm if there are four or more sites. The same is true if one uses system availability instead of site availability as the metric.*

PROOF. The statement that dynamic-linear has greater site availability than voting-primary is just a recapitulation of Theorem 3. The extension to system availability follows from Theorem 2 in Section 8.2. Barbara and Garcia-Molina have shown [4] that the system availability of voting-primary is greater than the system availability of any other static voting algorithm if $\mu > \lambda$. Their same proof applies to nonvoting static algorithms as well. That is, it applies to all coteries, not just those that correspond to vote assignments. (See Section 6 for an explanation of coteries.) Furthermore, their proof can easily be modified to work for site availability as well as system availability, if one assumes that updates arrive at sites uniformly. This condition is necessary because coteries other than simple voting do not treat the sites uniformly; those coteries might thereby benefit from a nonuniform arrival distribution. □

Recall that we are using a nonstandard measure of availability, namely, the limiting probability that an update arriving at some arbitrary site will be processed. Theorem 2 in Section 8.2 says that if we were to use the traditional measure—the long-term probability that a distinguished partition exists—the superiority of the dynamic algorithms would be further enhanced. In fact, under the traditional measure of availability, dynamic-linear is better than all static algorithms when there are three sites, as well as when there are four or more sites. Note that the two measures conflict in their advice when there are three sites: according to site availability, one should use simple voting, while system availability says that dynamic-linear is better.

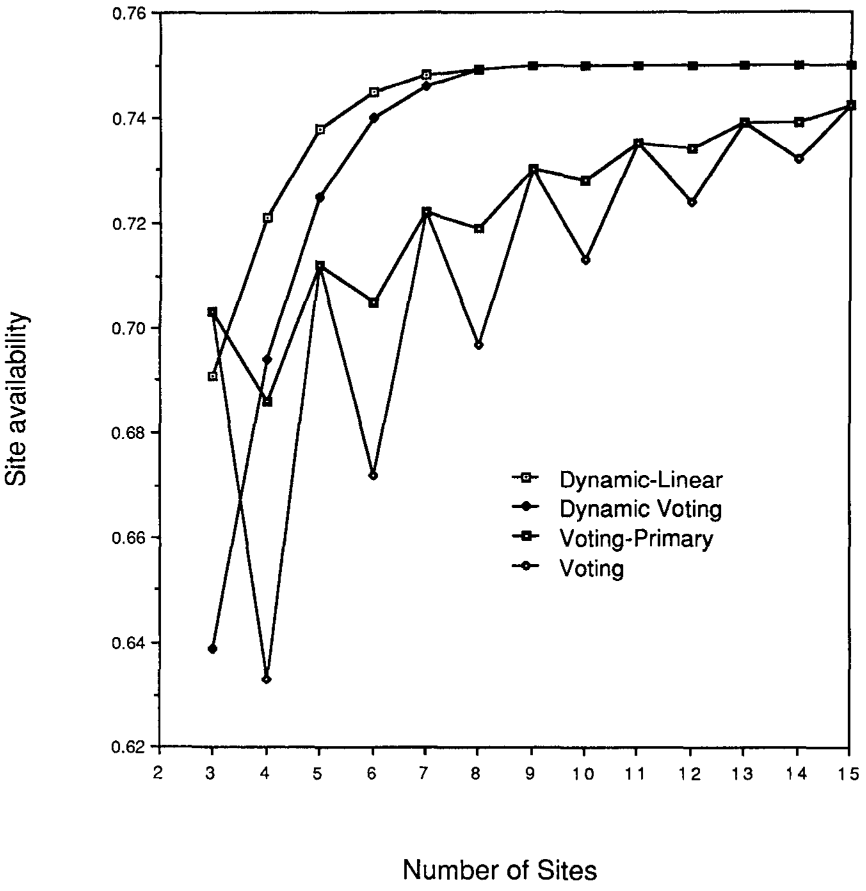

Fig. 3. Repair rate is three times failure rate.

8.3.6 *Numerical Comparison of the Availabilities.* The previous subsection established the qualitative superiority of the dynamic voting algorithms over static algorithms. This subsection displays the quantitative difference by graphing the performance of the algorithms at several sample points. Note that the scaling of the axes varies from graph to graph.

The graph depicted by Figure 3 shows availability graphed against the number n of sites, when the repair rate μ is three times the failure rate λ. Note that for three sites, voting is better than the two dynamic algorithms, while the reverse is true for four or more sites. As n grows large, the availability of each algorithm converges to $\mu/(\mu + \lambda)$ (=0.75 for Figure 3), the probability that the site at which the update arrives is a functioning site. However, the convergence of the two dynamic algorithms differs from that of the two static algorithms in two respects.

(1) The two dynamic algorithms approach the asymptotic value somewhat more quickly than the two static algorithms do. This effect is most pronounced when, as in Figure 3, the repair/failure ratio is small.

(2) The availability of each of the two dynamic algorithms increases *monotonically* as the number of sites increases. The nonmonotonicity of the static algorithms is most pronounced for simple voting with a small number of sites.

The repair/failure ratio for Figure 3 is much smaller than one would encounter in practice. The relative behavior of the four algorithms is unchanged for larger repair/failure ratios, although the absolute difference between the algorithms shrinks as this ratio increases. For instance, for a repair/failure ratio of 50.0, all four algorithms are within 0.00001 of the maximum possible availability by the time the number of sites has reached 7.

The graph in Figure 4 shows site availability for repair/failure ratios one might expect to encounter in practice when there are four sites. Here the improvement in going from voting-primary to dynamic-linear is small but still noticeable; it about equals the improvement in going from simple voting to voting-primary. The curve for dynamic voting is so close to that for dynamic-linear that it is omitted.

The graph shown by Figure 5 illustrates the availabilities of all four algorithms for small repair/failure ratios when there are five sites. This graph illustrates the crossover behavior stated in Theorem 3, which occurs only when there are four or five sites.

Formally, any explanation for characteristics of these graphs must come from the inequalities in the proofs of Theorems 3 and 4; the complicated nature of those inequalities defies simple explanations. Informally, the advantage of the dynamic algorithms over the static algorithms is that the former can survive more failures before rejecting updates. The disadvantage is that when dynamic algorithms finally start rejecting updates, it will take them longer to return (via repairs) to a state in which updates can again be accepted. This advantage increases when either the number of sites or the repair/failure ratio increases. Apparently, the number of sites is the more important of these two factors, with respect to the question of which algorithm is best. Thus, the dynamic algorithms do well when there are many sites, even if the repair/failure ratio is small. Likewise, the static algorithms do well when there are few sites, even if the repair/failure ratio is large. In between these extremes is the five-site case: there the repair/failure ratio determines the outcome of the comparison of dynamic voting and static voting. This may be the explanation of the crossover behavior shown in Figure 5.

Here is a simple argument to prove that dynamic voting has lower availability than static voting when there are three sites. Static voting and dynamic voting with three sites behave identically unless exactly two sites are up. In that case, static voting permits updates, while dynamic voting permits updates only if the two sites that are up both have up-to-date copies of the file. That is, any update accepted by dynamic voting is accepted by static voting, while the reverse is not true, when there are three sites.

The nonmonotonicity of the static algorithms can be explained easily by an example. In the seven-site case, simple voting rejects updates when there are four or more failures of the seven sites. In the eight-site case, again four or more

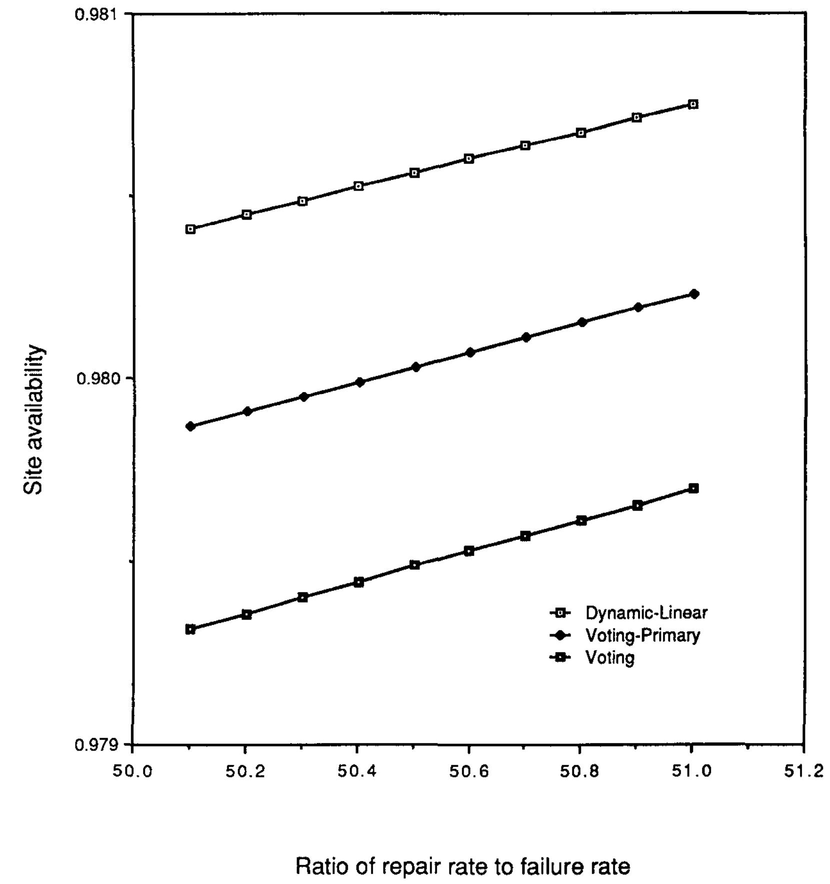

Fig. 4. Four sites.

failures cause updates to be rejected, but now these failures are drawn from a larger set (eight sites instead of seven). Four failures from eight sites is more likely than four failures from seven sites, if sites are more likely to be up than down. This explains why the availability of simple voting is lower at eight sites than at seven sites. Increasing the number of sites from seven to nine, however, increases availability, by a similar argument. Thus availability of simple voting increases, but not monotonically.

The nonmonotonicity of voting-primary is a consequence of using site availability as our measure. Under system availability, the availability of voting-primary with n sites is the same as its availability with $n - 1$ sites, for even n. The dynamic voting algorithms have elements of the nonmonotonicity of the static algorithms, but this is apparently overwhelmed by the fact that adding a site permits the dynamic algorithms to survive one more failure before updates are first rejected.

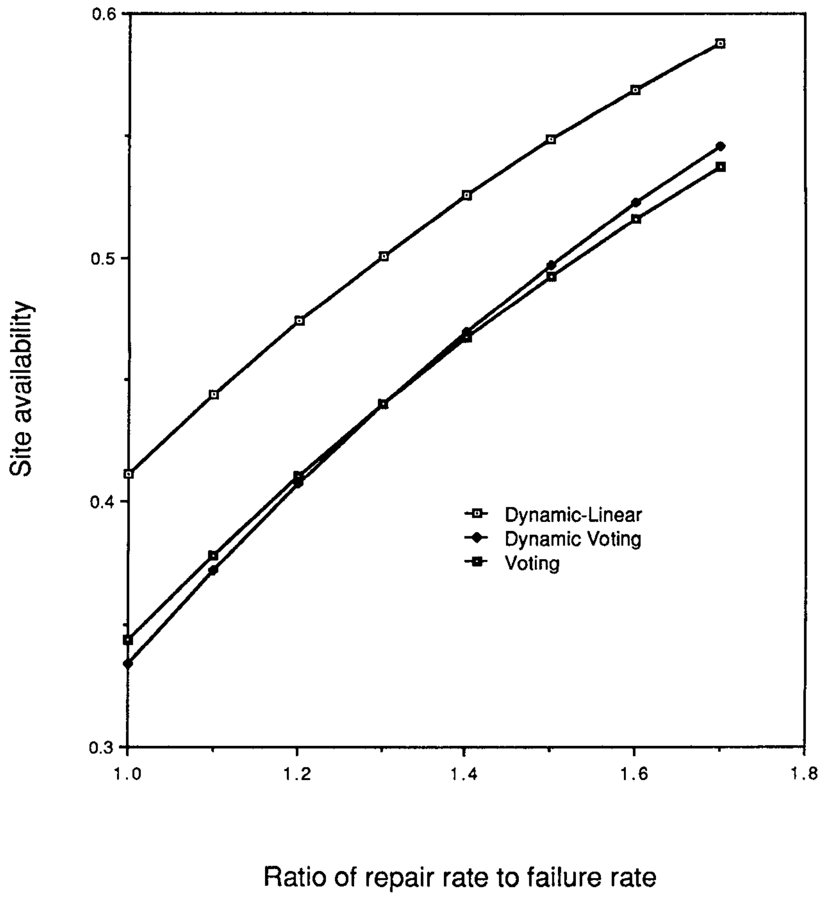

Fig. 5. Five sites.

8.4 The Link Model

8.4.1 *Specification of the Model.* The structure inherent in the site model permitted us to prove a strong, analytic theorem comparing our two dynamic algorithms to the class of all static algorithms. However, the site model has a serious flaw: it excludes partitioning, the very problem our algorithms are designed to handle! In this subsection we present a *link model* that permits partitioning. The results we have achieved in the link model are not as strong as those we have achieved in the site model, but once again, they support the superiority of dynamic-linear over static algorithms. By considering two complementary models, and acquiring the same qualitative result in each, we strengthen our confidence that the result will hold in the real world as well.

The link model is exactly the same as the site model except that *links* fail instead of *sites*. Here are the details of the link model.

(1) The network is modeled by a connected graph: nodes of the graph denote sites and edges denote bidirectional communication links. This graph, called the

initial graph, is a parameter of the model. As such, it can be any connected graph. The sites in the network are infallible, but each link is subject to failure and repair. The partitions of the network at any instant are the connected components of the graph obtained from the initial graph by erasing all links that are down at that instant.

(2) The failures at the various links form independent Poisson processes with *failure rate* λ. For any given link that is up (functioning), the probability it goes down (fails) at or before the next t time units is $1 - e^{-\lambda t}$. Similarly, the repairs at the various links form independent Poisson processes with *repair rate* μ.

(3) Updates are instantaneous. We ignore communication delays in the commit protocol.

(4) Updates arrive frequently: after any failure or repair, an update always arrives at one of the sites with an up-to-date copy of the file 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 sites with up-to-date copies of the file communicate to determine the new status of the system before the next failure or repair.

The link model is quite akin to the site model, and its use is justified on similar grounds. Both models treat failures and repairs homogeneously and with Markovian assumptions. Both models ignore the effect of communication delays. Both models assume updates arrive frequently with respect to failures and repairs. However, the link model introduces an additional parameter: the underlying network topology, as described by the initial graph.

Even when there are only a few sites, many initial graphs are possible. Of the many possible graphs, all but the regular graphs have an element of asymmetry in their treatment of the sites. These factors greatly complicate the analysis of our algorithms. To cope with this complexity, we restrict our attention to the eight possible biconnected topologies for five-site networks. These topologies are shown in Figure 6. These same topologies have been used both for nonstochastic analysis of static algorithms [5, 26] and for simulation of dynamic algorithms [7, 8].

The link model treats updates as if they were instantaneous. Thus, as in the site model, we ignore the distinction between logical and physical version numbers and speak simply of "the version number."

8.4.2 *Analysis of Static Algorithms.* Recall from Section 6 that a *coterie* is a list of groups of sites, such that any two groups intersect and no group is a proper subset of another group. A static algorithm is any algorithm that, implicitly (as in weighted voting) or explicitly, defines a group of sites to be a distinguished partition if and only if the group is a superset of a set in the coterie being used. In the context of the link model, the set of static algorithms is in one-to-one correspondence with the set of coteries.

There are 81 possible coteries in a five-site network and thus 81 static algorithms to analyze in the link model. Eight of these coteries are listed in Figure 7 (taken from [26]); the others are obtained from those eight by permuting sites. For each of the eight topologies in Figure 6, and for each of the 81 static algorithms, we computed the site and system availabilities of the static algorithm.

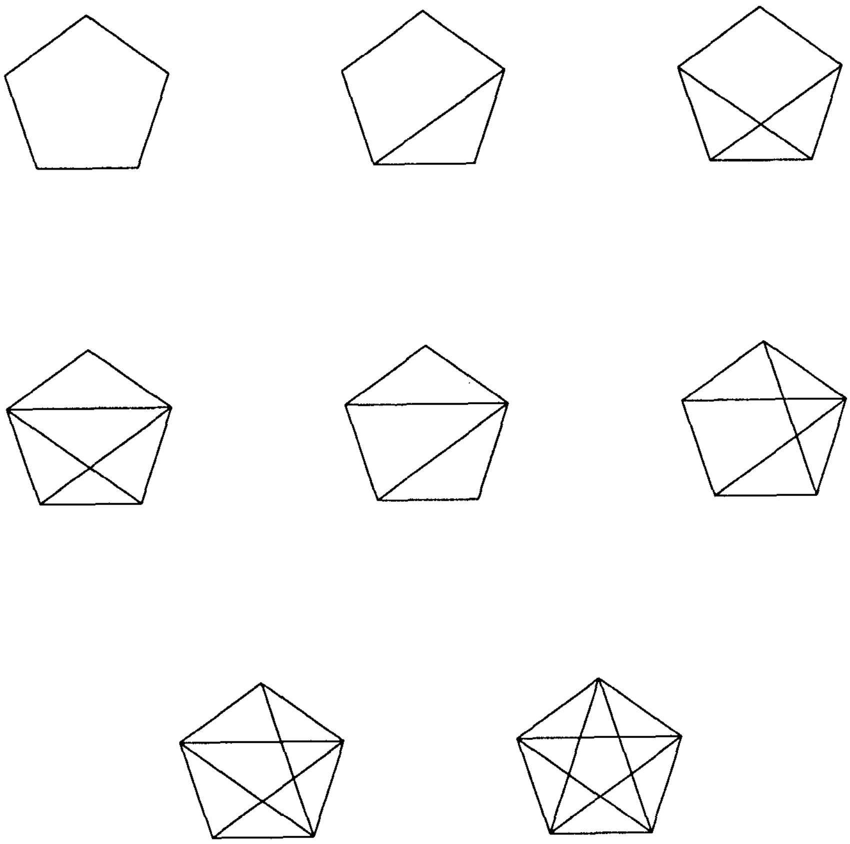

Fig. 6. The eight biconnected five-node graphs.

Coterie 1:	*{}*
Coterie 2:	*{a}*
Coterie 3:	*{a,b} {a,c} {b,c}*
Coterie 4:	*{a,b,c} {a,d} {b,d} {c,d}*
Coterie 5:	*{a,b,c} {b,d} {c,d} {b,c,e} {a,d,e}*
Coterie 6:	*{a,b,c} {c,d} {b,c,e} {a,d,e} {a,c,e} {b,d,a} {b,d,e}*
Coterie 7:	*{a,b,c} {b,c,e} {a,d,e} {a,c,e} {b,d,a}*
	{b,d,e} {a,b,e} {c,d,a} {c,d,b} {c,d,e}
Coterie 8:	*{a,b,c,d} {a,e} {b,e} {c,e} {d,e}*

Fig. 7. Eight of the 81 coteries for five sites a, b, c, d, and e.

(See Section 8.2 for the definitions of these two measures of availability.) This subsection describes the simple method by which that computation was done.

Let G denote an initial graph and consider a static algorithm, with associated coterie C, for which we wish to compute site and system availabilities. For either measure of availability, it suffices to know the steady-state probability distribution of the size of the distinguished partition. To determine which partition, if any, is defined by the algorithm to be the distinguished one at time t, and hence to know the size of that partition, one needs to know only the initial graph; the groups that form coterie C; and which links are up at time t. Thus we can model the action of the algorithm by letting the *state* of the network at time t be the list of links that are up at time t.

The mean time to failure of a functioning link is $1/\lambda$. The mean time to repair of a failed link is $1/\mu$. It follows that for the Poisson process describing the behavior of the links, the probability that any given link is up at any particular time is

$$\frac{1/\lambda}{1/\lambda + 1/\mu}, \quad \text{that is,} \quad \frac{\mu}{\lambda + \mu}.$$

Let m denote the number of links in the initial graph G, so that there are 2^m possible states of the system. For any state with u links up and the remaining $m - u$ links down, its probability is

$$\left[\frac{\mu}{\lambda + \mu}\right]^u \left[\frac{\lambda}{\lambda + \mu}\right]^{m-u}.$$

The system availability of a given static algorithm is computed by summing over all states that contain a distinguished partition whose size is nonzero, weighting each state by its probability given above. The site availability is computed likewise, weighting each term additionally by the size of the distinguished partition divided by the number of sites (five). Note that the availability will vary from one static algorithm to another because the associated coteries define different sets of distinguished partitions.

8.4.3 *Analysis of Dynamic Voting and Dynamic-Linear.* Static algorithms permit an analysis using only the evolving status of the links of the network. The dynamic algorithms, however, use the current vote assignment as well as the current status of the links to perform their periodic vote reassignment. The *state* of a dynamic algorithm must specify not only what links are up, but also which nodes have what votes.

We wrote a program that, given an initial graph, computes and draws the state diagram for dynamic-linear on that initial graph, under the link model. The state diagram for dynamic-linear with the ring topology is shown in Figure 8. The state diagrams for the other topologies are much larger, varying from 49 states to 159 states. Each state is depicted by its graph, with solid lines denoting links that are functioning and dashed lines denoting failed links. For each state, the set of nodes surrounded by a small circle or triangle forms the list of sites with up-to-date copies of the file in that state, that is, the list of sites with votes. The node surrounded by a triangle is the node in the distinguished site entry, when there are an even number of sites with votes. The solid double-arrows between states depict state transitions caused by failures (one direction) or repairs (the other direction). The dashed arrows between states depict one-directional state transitions caused by repairs. State A is the initial state (all links functioning). For clarity we have omitted from Figure 8 the arc labels that indicate the magnitudes of the state transitions. For instance, the transition from state F to

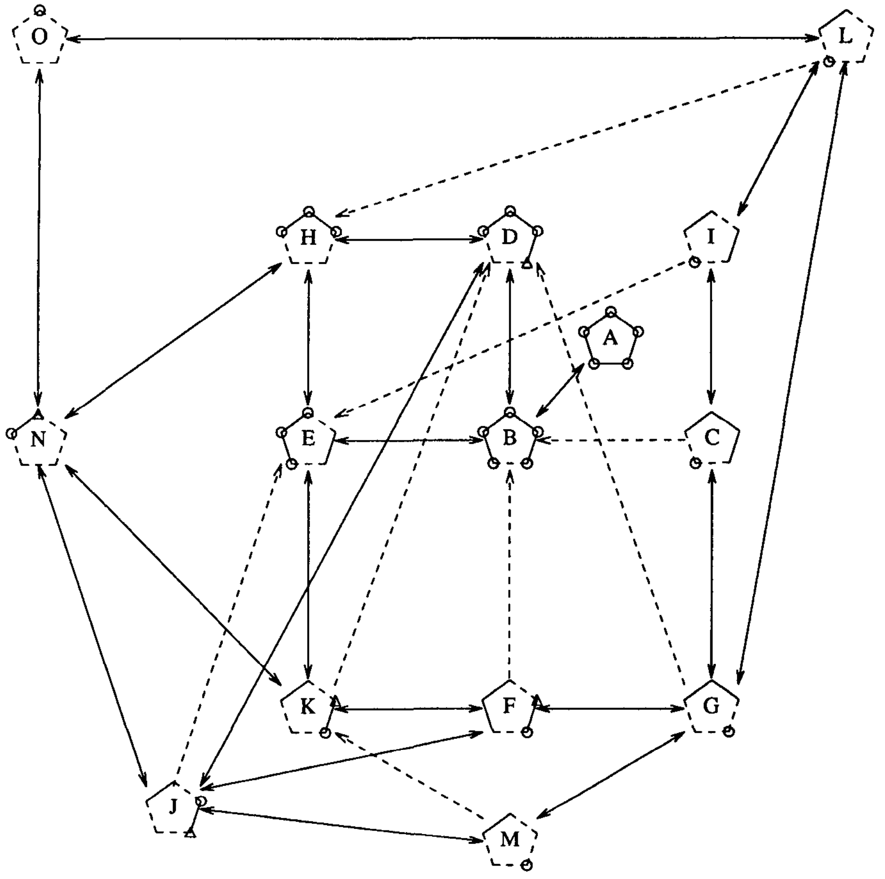

Fig. 8. The state diagram for a five-element ring.

state B has magnitude 2μ, because the repair of either of the two dashed links in state F enacts a transition to state B.

The balance equations are acquired (mechanically) from the state diagrams by setting flow-into each state equal to flow-out. For example, if we let A, B, C, ... denote the probabilities of states A, B, C, ..., then the equation corresponding to state B is

$$5\lambda A + 2\mu C + 2\mu D + 2\mu E + 2\mu F = [4\lambda + \mu]B.$$

The balance equations are in terms of parameters μ and λ. When the number of equations is small, they can be solved effectively by a symbolic manipulator like MACSYMA or MAPLE. For larger systems, your favorite program for solving systems of linear equations numerically will do the job for fixed repair/failure ratio μ/λ.

8.4.4 *Summary of Results for the Link Model.* For each of the eight biconnected five-site graphs, we computed the site availabilities of dynamic-linear and all 81 static algorithms applicable to five-site graphs, for repair/failure

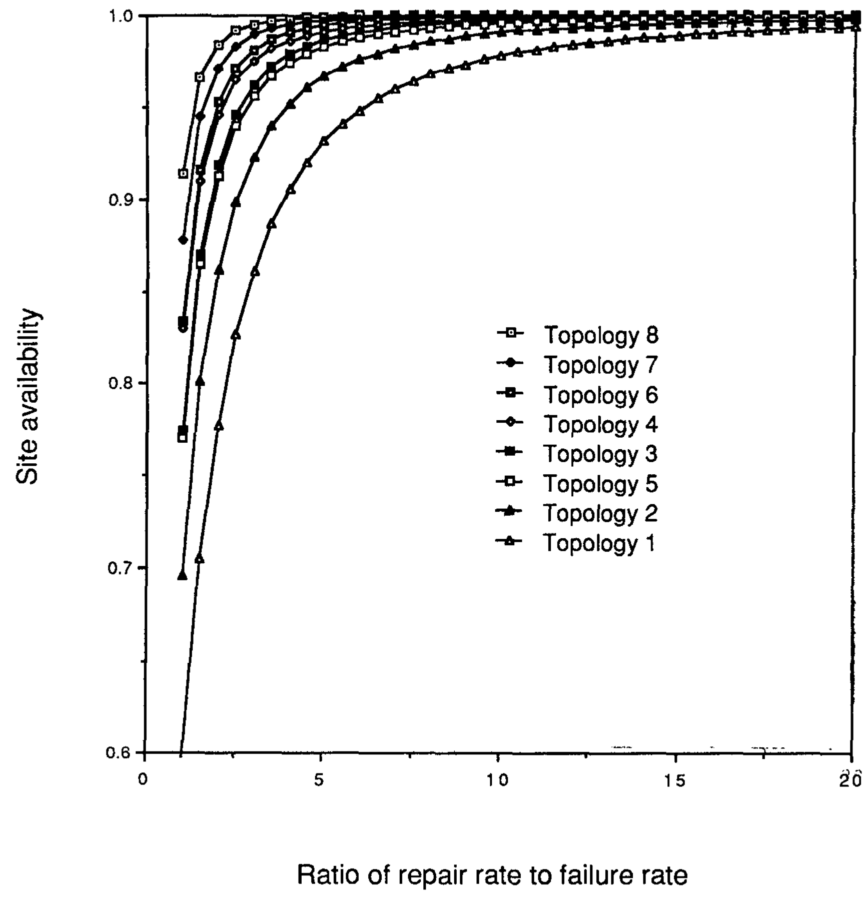

Fig. 9. Dynamic-linear under the eight topologies.

ratios varying from 1.0 to 20.0 in increments of 0.1. In each of these cases, dynamic-linear had larger availability than the best static algorithm.

Figures 9 and 10 present the results graphically. Each graph depicts site availability versus the ratio of the repair rate μ to the failure rate λ.

The graph of Figure 9 shows the effect of the topology on the site availability of dynamic-linear. The eight curves are for the eight biconnected, five-site topologies studied. The topologies are numbered as they are presented in Figure 6, left to right and top to bottom. For example, topology 1 is a ring and topology 8 is a complete graph. Each curve encompasses the entire range of repair/failure ratios studied. As one would expect, the more interconnected the topology, the higher the availability. For each topology, the site availability increases sharply as the repair/failure ratio increases from 1.0. The rate of increase is large and nearly constant for a while, then decreases steadily, soon becoming near zero. Of course, the availability for all eight topologies converges to 1.0 as the repair/failure ratio grows large. The steepness of the initial ascent and the point at

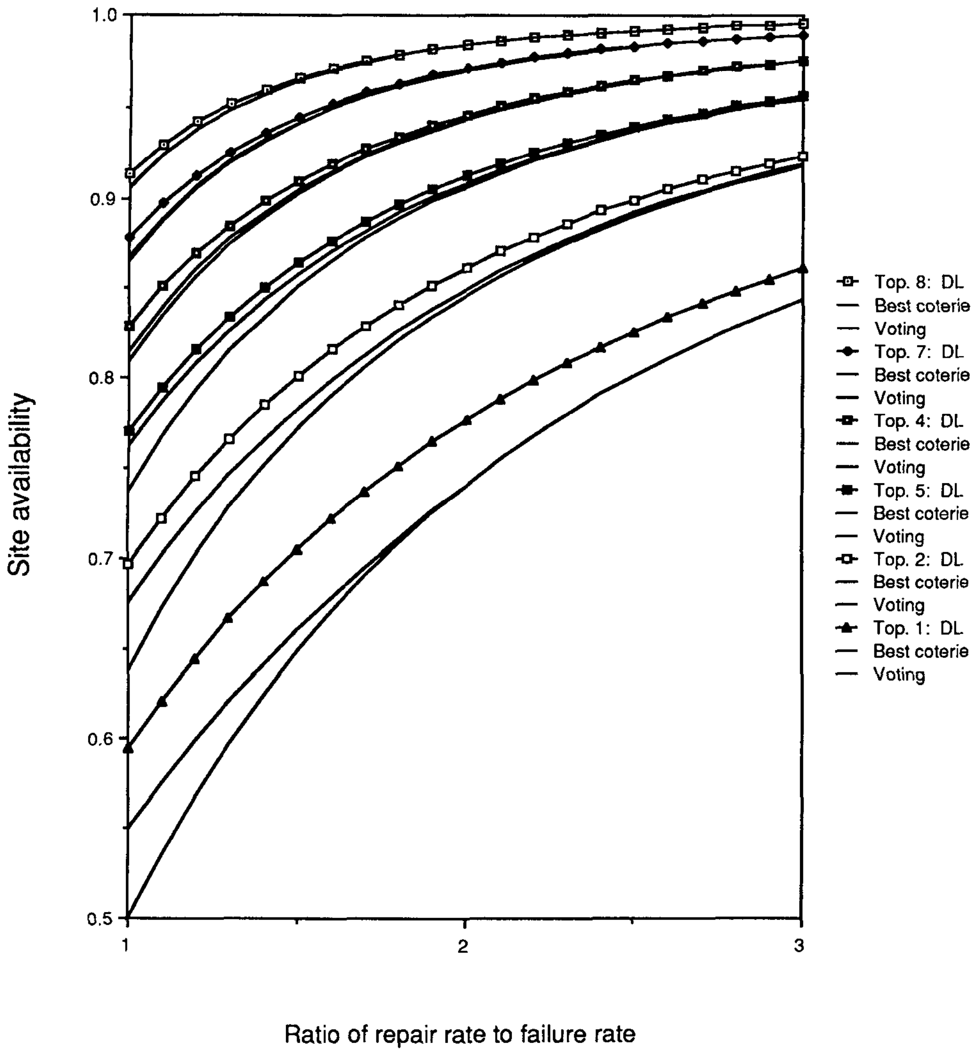

Fig. 10. Dynamic-linear, best coterie, and voting.

which the slope becomes less than 1.0 varies from topology to topology. The more interconnected the topology, the steeper the initial descent and the quicker the slope falls below 1.0.

The graph of Figure 10 is an enlargement of the left-most portion of Graph 4, but with curves for the best static algorithm and simple voting included. The curves for topologies 3 and 6 are omitted; those curves lie very close to the curves for topologies 5 and 4, respectively. For each topology, the curve for dynamic-linear lies above the curve for the best static algorithm, which in turn lies above the curve for simple voting. (The curves for the best static algorithm and voting are essentially indistinguishable for each of the topologies 7 and 8.) For each topology, simple voting becomes the best static algorithm once the repair/failure ratio is sufficiently large. The less connected the topology, the greater the superiority of dynamic-linear over the best static algorithm. In particular, one has the most to gain by using dynamic-linear in a ring topology.

9. SUMMARY AND OPEN PROBLEMS

We have proposed two algorithms, dynamic voting and dynamic-linear, that maintain the consistency of a replicated database in the face of network partitioning. We believe that dynamic-linear is superior to static algorithms for the following reasons.

(1) Dynamic-linear is as easy to implement as static voting.
(2) The message complexity of dynamic-linear is only slightly greater than the message complexity of static voting.
(3) Dynamic-linear has greater availability than any static algorithm.

The claim for greater availability is supported by two models, one in which sites fail and one in which links fail. Theorem 4 in the site model is unique among results providing analysis of dynamic vote reassignment: no other theorem to date applies to an unbounded number of sites. Other results have been only for small, fixed values for the number of sites.

Here are some open problems that remain to be solved.

(1) Barbara and Garcia-Molina have shown [4] that the optimal static algorithm, under the site model, is the voting-primary algorithm, if sites are more likely to be up than down. We have shown (Theorem 3) that voting-primary is bested by dynamic-linear, when there are four or more sites. The question remains: what is the optimal *dynamic* algorithm, under the site model? A hybrid algorithm—use dynamic voting when there are four or more sites with current copies of the replicated file, but switch to static three-site voting when the number of current copies falls to three—provides better availability than dynamic voting, for reasonable repair/failure ratios [32]. What algorithm will conclude this series of small improvements?

(2) In practice one will encounter asymmetric networks that are less than completely connected and in which some sites and links are more reliable than others. What is the optimal assignment of votes for weighted dynamic voting? For dynamic coteries? When such optimal assignments are difficult to determine, what heuristic assignments are reasonable and what bounds can be placed on the performance of the heuristics?

(3) The site and link models are each idealized. Does the relative standing of dynamic-linear and static algorithms change when communication delays are introduced? We think not (because all these algorithms face similar communication delays in the commit protocol), but offer no proof. What happens if update arrivals are not frequent relative to repairs and failures? We believe that a modest relaxation of the frequent-update modeling assumption will not seriously degrade the availabilities of our dynamic algorithms, but again offer no proof.

REFERENCES

1. AHAMAD, M., AND AMMAR, M. Performance characterization of quorum-consensus algorithms for replicated data. In *Proceedings of the Symposium on Reliability in Distributed Software and Database Systems* (1987), pp. 161–168.
2. ALSBERG, P. A., AND DAY, J. D. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering* (1976), 562–570.
3. ATTAR, R., BERNSTEIN, P. A., AND GOODMAN, N. Site initialization, recovery, and backup in a distributed database system. In *Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks* (1982), pp. 185–202.
4. BARBARA, D., AND GARCIA-MOLINA, H. The reliability of voting mechanisms. Tech. Rep. TR 330, Dept. of Electrical Engineering and Computer Science, Princeton University, Princeton, N.J., 1984.
5. BARBARA, D., AND GARCIA-MOLINA, H. The vulnerability of vote assignments. *ACM Trans. Comput. Syst. 4*, 3 (Aug. 1986), 187–213.
6. BARBARA, D., GARCIA-MOLINA, H., AND SPAUSTER, A. Increasing availability under mutual exclusion constraints with dynamic vote assignment. Tech. Rep. CS-TR-056-86, Dept. of Computer Science, Princeton University, Princeton, N.J., Nov. 1986.
7. BARBARA, D., GARCIA-MOLINA, H., AND SPAUSTER, A. Policies for dynamic vote reassignment. In *Proceedings of the IEEE Conference on Distributed Computing* (1986). IEEE, New York, 1986, pp. 37–44.
8. BARBARA, D., GARCIA-MOLINA, H., AND SPAUSTER, A. Protocols for dynamic vote reassignment. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing* (1986). ACM, New York, 1986, pp. 195–205.
9. BERNSTEIN, P. A., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Comput. Surv. 13*, 2 (June 1981), 185–221.
10. BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Mass., 1987.
11. BLAUSTEIN, B. T., AND KAUFMAN, C. W. Updating replicated data during communication failures. In *Proceedings of the 11th International Conference on Very Large Data Bases* (1985), pp. 49–58.
12. CERI, S., AND PELAGATTI, G. *Distributed Databases: Principles and Systems*. McGraw-Hill, New York, 1984.
13. CHAR, B. W., FEE, G. J., GEDDES, K. O., GONNET, G. H., AND MONAGAN, M. B. A tutorial introduction to Maple. *J. Symbolic Comput. 2*, 2 (1986), 179–200.
14. COOPER, E. C. Analysis of distributed commit protocols. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1982). ACM, New York, 1982, pp. 175–183.
15. DAVCEV, D., AND BURKHARD, W. Consistency and recovery control for replicated files. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (1985). ACM, New York, 1985, pp. 87–96.
16. DAVIDSON, S. B. Optimism and consistency in partitioned distributed database systems. *ACM Trans. Database Syst. 9*, 3 (Sept. 1984), 456–481.
17. DAVIDSON, S. B., GARCIA-MOLINA, H., AND SKEEN, D. Consistency in partitioned networks. *ACM Comput. Surv. 17*, 3 (Sept. 1985), 341–370.
18. DUGAN, J., AND CIARDO, G. Stochastic petri net analysis of a replicated file system. In *Proceedings of the International Workshop on Petri Nets and Performance Models* (Aug. 1987).
19. EAGER, D. L., AND SEVCIK, K. C. Achieving robustness in distributed database systems. *ACM Trans. Database Syst. 8*, 3 (Sept. 1983), 354–381.
20. EL ABBADI, A., SKEEN, D., AND CHRISTIAN, F. An efficient, fault-tolerant protocol for replicated data management. In *Proceedings of the 4th ACM Symposium on Principles of Database Systems* (1985). ACM, New York, 1985, pp. 215–228.
21. EL ABBADI, A., AND TOUEG, S. Availability in partitioned replicated databases. In *Proceedings of the 5th ACM Symposium on Principles of Database Systems* (1986). ACM, New York, 1986, pp. 240–251.
22. EL ABBADI, A., AND TOUEG, S. Maintaining availability in partitioned replicated databases. Tech. Rep. TR-87-857, Dept. of Computer Science, Cornell University, Ithaca, N.Y., 1987.

23. FISCHER, M. J., AND MICHAEL, A. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings of the ACM Symposium on Principles of Database Systems* (1982). ACM, New York, 1982, pp. 70–75.
24. GARCIA-MOLINA, H. Elections in a distributed computing system. *IEEE Trans. Computers C-31*, 1 (Jan. 1982), pp. 48–59.
25. GARCIA-MOLINA, H. Reliability issues for fully replicated distributed databases. *IEEE Comput. 15*, 9 (Sept. 1982), pp. 34–42.
26. GARCIA-MOLINA, H., AND BARBARA, D. How to assign votes in a distributed system. *J. ACM 32*, 4 (Oct. 1985), 841–860.
27. GIFFORD, D. K. Weighted voting for replicated data. In *Proceedings of the 7th Symposium on Operating System Principles* (1979), pp. 150–162.
28. GRAY, J. N. Notes on database operating systems. In R. Bayer, R.M. Graham, and G. Seegmuller, Eds., *Lecture Notes in Computer Science*, vol. 60. Springer-Verlag, New York, 1978, pp. 394–481.
29. JAJODIA, S., AND MEADOWS, C. A. Mutual consistency in decentralized distributed systems. In *Proceedings of the IEEE 3rd International Conference on Data Engineering* (1987). IEEE, New York, 1987, pp. 396–404.
30. JAJODIA, S., AND MUTCHLER, D. A pessimistic consistency control algorithm for replicated files which achieves high availability. *IEEE Trans. Softw. Eng. 15*, 1 (Jan. 1989), 39–46.
31. JAJODIA, S., AND MUTCHLER, D. Dynamic voting. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1987). ACM, New York, 1987, pp. 227–238.
32. JAJODIA, S., AND MUTCHLER, D. Enhancements to the voting algorithm. In *Proceedings of the 13th International Conference on Very Large Data Bases* (1987), pp. 399–406.
33. JAJODIA, S., AND MUTCHLER, D. Integrating static and dynamic voting protocols to enhance file availability. In *Proceedings of the 4th IEEE International Conference on Data Engineering* (1988). IEEE, New York, 1988, pp. 144–153.
34. KOHLER, W. H. A survey of techniques for synchronization and recovery in decentralized computer systems. *ACM Comput. Surv. 13*, 2 (1981), 149–183.
35. LAMPORT, L. The implementation of reliable distributed multiprocess systems. *Comput. Networks 2* (1978), 95–114.
36. LAMPSON, B., AND STURGIS, H. Crash recovery in a distributed data storage system. Tech. Rep., Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, Calif., 1976.
37. MINOURA, T., AND WIEDERHOLD, G. Resilient extended true-copy token scheme for a distributed database system. *IEEE Trans. Softw. Eng. SE-8*, 3 (1982), 173–189.
38. PÂRIS, J.-F. Voting with a variable number of copies. In *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing* (1986), pp. 50–55.
39. PÂRIS, J.-F. Voting with witnesses: A consistency scheme for replicated files. In *Proceedings of the IEEE International Conference on Distributed Computing* (1986). IEEE, New York, 1986, pp. 606–621.
40. PARKER, D. S., JR., POPEK, G. J., RUDISIN, G., STOUGHTON, A., WALKER, B. J., WALTON, E., CHOW, J. M., EDWARDS, D., KISER, S., AND KLINE, C. Detection of mutual inconsistency in databases. *IEEE Trans. Softw. Eng. SE-9*, 3 (1983), 240–247.
41. PEASE, M., SHOSTAK, R., AND LAMPORT, L. Reaching agreement in the presence of faults. *J. ACM 27*, 2 (Apr. 1980), 228–234.
42. RAMARAO, K. V. S. Detection of mutual inconsistency in distributed databases. In *Proceedings of the 3rd IEEE International Conference on Data Engineering* (1987). IEEE, New York, 1987, pp. 405–411.
43. RAMARAO, K. V. S. Transaction atomicity in the presence of network partitions. In *Proceedings of the 4th IEEE International Conference on Data Engineering* (1988). IEEE, New York, 1988, pp. 512–519.
44. SARIN, S. K., BLAUSTEIN, B. T., AND KAUFMAN, C. W. System architecture for partition-tolerant distributed databases. *IEEE Trans. Comput. C-34*, 12 (1985), 1158–1163.
45. SCHLICTING, R., AND SCHNEIDER, F. Fail-stop processors: An approach to designing fault-tolerant distributed computing systems. *ACM Trans. Comput. Syst. 1*, 3 (1983), 222–238.
46. SEGUIN, J., SERGEANT, G., AND WILMS, P. A majority consensus algorithm for the consistency of duplicated and distributed information. In *Proceedings of the IEEE International Conference on Distributed Computing Systems* (1979). IEEE, New York, 1979, pp. 617–624.

47. SELINGER, P. G. Replicated data. In Distributed Databases. I. W. Draffen and F. Poole, Eds. Cambridge University Press, Cambridge, 1980.
48. SKEEN, D., AND STONEBRAKER, M. A formal model of crash recovery in a distributed system. *IEEE Trans. Softw. Eng. SE-9*, 3 (1983), 219–228.
49. SKEEN, D., AND WRIGHT, D. Increasing availability in partitioned database systems. In *Proceedings of the 3rd ACM Symposium on Principles of Database Systems* (1984). ACM, New York, 1984, pp. 290–299.
50. THOMAS, R. H. A majority consensus approach to concurrency control. *ACM Trans. Database Syst. 4*, 2 (June 1979), 180–209.
51. THOMAS, R. H. A solution to the concurrency control problem for multiple copy databases. In *Proceedings of IEEE Compcon* (Spring 1978). IEEE, New York, 1978, pp. 56–62.
52. TRIVEDI, K. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications.* Prentice-Hall, Englewood Cliffs, N.J., 1982.
53. WRIGHT, D. D. Managing distributed databases in partitioned networks. Tech. Rep. 83-572, Dept. of Computer Science, Cornell University, Ithaca, N.Y., 1983.
54. WRIGHT, D. D. On merging partitioned databases. *ACM SIGMOD Rec. 13*, 4 (1983), 6–14.

Received March 1988; revised October 1988; accepted March 1989