1987

A Dynamic Majority Determination Algorithm for Reconfiguration of Network Partitions

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Peter Lei Ng

Report Number:
87-712

A DYNAMIC MAJORITY DETERMINATION ALGORITHM FOR RECONFIGURATION OF NETWORK PARTITIONS

Bharat Bhargava
Peter Lei Ng

CSD-TR-712
September 1987

A Dynamic Majority Determination Algorithm for Reconfiguration of Network Partitions

Bharat Bhargava
Peter Lei Ng

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

ABSTRACT

We present a *conservative* consistency and recovery control algorithm for replicated files in the presence of network partitioning due to communication link failures. This algorithm supports partial replication, provides non-blocking operations by allowing update access to a file in that file's majority partition, and brings all copies up-to-date on all sites whenever the communication links among them are repaired. This algorithm belongs to the class of dynamic voting algorithms proposed in the recent literature. When the communication link among some partitions is reestablished, the algorithms proposed so far do not always allow the merge (reconciliation) of these partitions to form a single partition. A merge condition has to be satisfied to avoid possible inconsistencies. This is undesirable because in a system with more than one replicated file, two or more partitions cannot be integrated to form a single partition if any one of the replicated files in these partitions does not satisfy the merge condition. This restriction might cause the system to remain partitioned for a long time even if communication links are repaired. (In the previous papers, such a problem is not addressed since a system with only one replicated file is assumed.) The algorithm proposed in this paper releases such merge condition and integrates the partitions whenever the communication link failure is repaired, thus providing a higher degree of availability. This work formalizes the presentation of algorithms and data structures for implementation.

1. Introduction

1.1. Background

A distributed database (DDB) consists of a set of logical data items stored at a set of sites interconnected by a communication network. The granularity of these logical data items can be a record, a relation, a file, etc. Without loss of generality, in the following discussion, we assume the granularity of these items to be a file.

To improve performance, data availability, and reliability, certain logical files are replicated at more than one site [9,17]. A logical file is fully replicated if each site in the DDB has a copy of that file. While replication is desirable, it is impractical to fully replicate every file in a DDB [1]. It is safe to assume that some of the files are partially replicated. For replicated copies, mutual consistency must be ensured. An update to a *physical copy* (or *copy*) of a *logical file* (or *file*) must be posted on all other copies of that file. The copies of a file are *mutually consistent* if whenever an update is performed on one of these copies, any other copy of that file cannot be accessed before it is also updated correctly. While preserving mutual consistency of a file is a sufficient condition for the correct access of that file, maintaining such mutual consistency while allowing updates to that file is difficult in the presence of a network partition. A network partition occurs when the network is split into several groups of sites, such that sites in each group can communicate with each other but not with a site in another group. A *partition* of a DDB is a maximal subset of communicating sites in that DDB [16]. Under normal operation, the whole DDB is itself a single partition. Some researchers have defined a partition of a DDB at the file level [15]. Under this model, two sites are considered to be in different partitions if the version

numbers of the two copies of a file *f* stored at these two sites are different, even if these two sites are physically connected. In this paper, we consider the partition at the site level rather than at the file level due to the following reason. By defining a partition at site level, only a simple data structure, namely, a connection vector (see definition in section 2), is required at each site to keep track of the current partition configuration of the network. Connection vector is not sufficient to represent the current partition configuration for all files at a site if the file level definition is used.

When the DDB becomes partitioned, unrestricted updates to the copies of replicated files can violate the mutual consistencies of these files. Therefore, a *consistency control protocol* must be enforced for access when the network is partitioned. A *recovery control protocol* is required to reconciliate the DDB after the network is repaired.

Many algorithms have been proposed to solve these problems and a survey is given in [8]. They use one of the two approaches : the *optimistic* approach, and the *conservative* approach. An optimistic algorithm allows updates to occur freely in any partition. The mutual inconsistencies might be allowed during the period in which the network is partitioned. When the partitions are merged, inconsistencies are *detected* and *resolved.* Such algorithms are termed optimistic because it is believed that there will be only a small amount of inconsistency and it can be resolved inexpensively when merging. The inconsistencies are usually resolved by *rolling back* (undoing) some transactions.

A conservative algorithm permits updates to a file to occur in at most one partition (the majority partition). All other copies of that file in other partitions are not updated. Such algorithms avoid mutual inconsistencies at the expense of losing availability. The conservative approach has an appealing property that the recovery protocol is simple because no inconsistent

access to data can take place when the system is partitioned. The updates are propagated to the out-of-date copies. No roll backs of transactions are needed. The research in this paper contributes to the conservative approach.

1.2. Discussion of the Research Problem

The research problem is to find solutions to allow:

a) read access to the latest copy on all sites

b) to determine a unique majority partition during multiple network partitions and merges in order to allow updates.

We attack the first problem by performing the merge of the copies of the file without violating the consistency as soon as two sites with different versions of copies can communicate. The details are given in section 3. The second problem is resolved by using the idea [7] of calling the majority of the previous majority as the new majority. Of course any site can join the majority partition. The update access is restricted such that only the copies in the majority partition are allowed to update. It is possible that after multiple partitions, the number of sites in the majority (of majority)* partition may become too small (say below an unacceptable threshold). A solution suggested in [5] declares a tie among the sites in the last majority under such conditions. A new majority is established after a merge occurs involving the sites in the last majority and the sites from the minority set. Several options can be exercised to determine a unique majority. For example, if the majority of the sites considered as minority so far merge with a site(s) of the last majority, a unique majority is established.

We discuss the research problem further in the following paragraphs. In a conservative algorithm, a group of sites is considered to constitute a partition if these sites can communicate with each other and all copies of each replicated file at these sites are consistent. We can distinguish two types of file access: read-only and update. A replicated file is available for updates in at most one partition, the file's majority partition. Update access to that file in other partitions is blocked. However, read-only access can be allowed in all partitions using the correctness criterion of view serializability for concurrency control [4,19].

The availability of a file in a dynamically changing network depends on how we select the majority partition after the previous one is partitioned. In conservative algorithms, under some circumstances, a majority partition of a file may not exist. For example, in the majority consensus algorithm[18], if the network splits into two equal size partitions, the majority partition is lost. None of the partition can claim to be a majority.

To improve the availability of a file, two directions can be followed. The first one is to avoid losing the majority partition. The other one is to keep the size of the majority partition above a threshold, even if the majority may be temporarily lost in the hope that a larger majority partition might be formed due to other merges. We present example 1 to illustrate this point.

Example 1. Consider the partition history of a file represented by the partition graph[15] in Fig. 1. Following the first direction, we might allow partitions ABC and AB as the majority partition. No loss of majority partition occurs in this history. But if the network remains in the configuration of AB and CDE for a long time, the file is not available in the partition CDE, which is larger in size than the majority partition AB during this period. The second direction will lead us to select partitions ABC and then CDE as the majority partition. In this case, the majority

partition is temporarily lost when ABC breaks into AB and C. Then C is merged with DE and the majority partition is reformed.

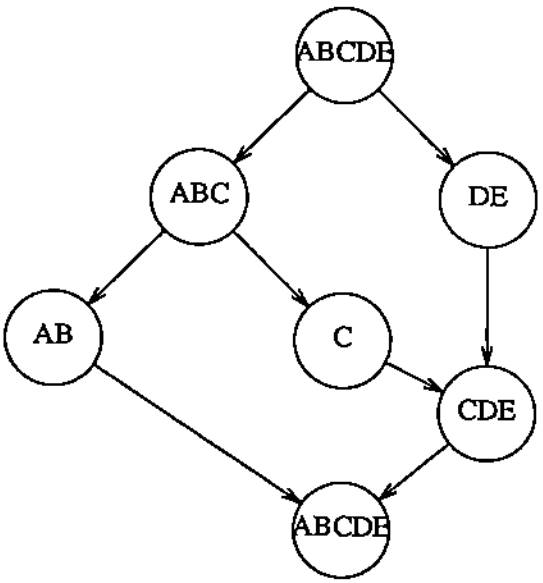

Fig. 1 Partition history of a file replicated at five sites.

In the example 1, it seems that the second method is better than the first one. But if CDE exists for a very short period, then the first method might be better. Since future behavior of a system is difficult to predict, we cannot say which method is better than the other. But we have a choice here. This issue of choice is discussed further in the section 5.

Different proposals along the first direction have been presented in some recent papers. In the dynamic vote reassignment scheme [2], each site can have more than one vote assignment. Access to a file is allowed in the partition with a majority of votes. To reduce the possibility of losing majority partition, a site in the current majority partition can just autonomously increase the weight of its votes without requiring consensus of other sites in that partition. But notification

of such an increase to the others sites within its partition is required.

In the dynamic voting schemes proposed in [7] and [11], if another partitioning occurs in the majority partition of a file, among the resulting partitions, one containing more than one half of the copies of that file in the previous majority partition is accepted as the new majority partition of that file. If there is a tie (the majority partition splits into two partitions, each with one half of the copies of that file in the previous partition), a predetermined linear order to the copies of a file, is able to break the tie [11]. Both schemes assume a connection vector at each site that reflects the current connectivity of the network. Associated with each copy of a file, a version number X and a version vector V keep the state information of that copy relative to the other copies in other partitions. One advantage of these schemes is that the cost of the file access is cheap, because the determination of whether or not a site is in a file's majority partition can be made by consulting only the local state information X and V associated with the local copy of that file. No inter-site communication is needed.

The problem with both schemes is that after the communication link among some partitions is repaired, they cannot always perform the merge (reconciliation) of these partitions to form a single partition. This is because the merge leaves the newly formed partition in such a state that the local X's and V's can no longer provide the correct state information for the majority determination. A site can incorrectly determine, by consulting the local X and V of a file, that it is in the majority partition of that file, thus two majority partitions for a file might exist at the same time. To illustrate this, an example is given in Section 4. Therefore, in these schemes it is necessary to delay the merge, even though the communication link is repaired, to avoid possible mutual inconsistency. However, there may be files that do not cause such anomaly after merge. If the merge

were performed, they would become accessible at more sites. There appears to be an undesirable phenomenon that the availability of some files is reduced by the consistency requirement of others.

Refinement to the above ideas have been proposed in [12,13]. These enhancements use simpler data structures. In [12], the connection vector and version vector are not required as in [7]. Instead, an integer called *update sites cardinality*(SC) is associated with each copy, which reflects the number of sites participating in the most recent update to that copy. In [13], the connection vector is not required, and a boolean vector called *update sites vector*(SV) is used instead of the integer version vector as in [11]. Even though simpler data structures reduce the maintanence effort, the determination of whether or not a site is in a file's majority partition, both [12] and [13] require an extra round of message passing to collect the version numbers and SC's or SV's from all other sites with which this site can communicate. Here, we see a trade-off — between spending less time to maintain simpler data structures and taking more message rounds to make the majority decision. However, these refinements still do not allow arbitrary merge of sites. Even if physically connected they pretend that such sites are in different partition. In this paper several of these problems have been resolved.

We now present the details of our algorithm by presenting the assumptions, definitions and data structures in section 2. The algorithm is presented in Section 3. Section 4 compares our algorithm with the algorithms proposed in [7,11,12,13]. Discussion and some possible improvements are given in Section 5.

2. Assumptions, Definitions and Data Structures

Assumptions:

(1) All sites can detect network partitioning using some mechanism such as time out. If the network consists of k sites, each site will have a connection vector of k elements that reflects the current connectivity of the network. For example, if the network consists of three sites, s_1,s_2, s_3, the connection vector at s_1 with value <1, 0, 1> denotes that s_1 is currently connected with s_3 but separated from s_2. (Note that when we say "A is connected with B", we mean that A can communicate with B. A and B might not be connected directly.) The table of all connection vectors in the network is reflexive, symmetric and transitive. That is, the relation "*is-connected-with*" is an equivalence relation.

(2) Each site processes messages in a FIFO order relative to every sending site. There is no loss of messages between connected sites. All messages from a site arrive in the order as sent by that site. Messages arrive without transmission error.

(3) The system runs a correct concurrency control protocol [3] that ensures the serializability of transactions in each partition.

Data Structures:

Suppose that a file f is replicated at n sites. We associate a replication vector S and a linear order vector L with file f and a version number X, a version vector V, and a marker vector M with each copy of the file f.

The *replication vector* of f, denoted by $S = <s_1,s_2,...,s_n>$, is a vector of the names (or identification number) of the n sites at which f is replicated. The *linear order vector* of f, denoted by $L = <l_1,l_2,..., l_n>$, is a vector of n distinct integers that defines the linear order among the copies of f at these n sites. The elements in S and L are position related, that is, l_i is the order of the copy at site s_i. For instance, associated with file f in Example 1, one possible values of S and L could be $S = <A,B,C>$ and $L = <2,3,1>$, indicating that the linear order, associated with f, of these three sites is B > A > C. When f is created, S and L are decided, and will not be altered unless we want to change the topological distribution of f and/or redefine its linear order. S is replicated at every site in the system. L is replicated at each site that contains a copy of f.

The *version number* X of a copy is an integer that records the number of successful updates to that copy. Since all copies of f in a partition are mutually consistent, the version numbers of all these copies are identical. The *current version number* of f is the largest version number of all copies of f.

The version vector of the copy at site s_i, denoted by $V = <v_1,v_2,...,v_n>$, is a vector of n integers. Because s_i is always connected with itself, $v_i = 0$. If s_i and s_j are still connected, $v_j = 0$ Note that we are defining the version vector of the copy at s_i. If s_i and s_j are separated, v_j will have the value of the version number X at the time when s_i was isolated from s_j. Since all copies of f in a partition are mutually consistent, the version vectors of all these copies are identical. Again, following the example 1, let the version vector at site B have a value of $V = <2,0,0>$. This version vector tells that B is currently connected with (i.e., in the same partition as) C, and A is isolated from BC since the version number X was equal to 2. In this case, the version vector at site C will have the same value as the V above. The version vector in site A will be $V = <0,2,2>$.

The marker vector of a copy, denoted by M = $<m_1,m_2,...,m_n>$, is a vector of n booleans. Each element m_i has a boolean value of either T (indicating that the copy at site s_i is marked) or F (indicating that the copy at s_i is unmarked).

The X, V, and M associated with a copy are stored at the same site as that copy. Initially, X and all elements in V are set to 0; all elements in M are set to F (unmarked).

Majority Partition:

A copy is *current* if it is unmarked and its version number equals the current version number. Note that a copy will not be considered current if it is marked, no matter what value its version number has. The *majority partition* of f is a partition that either contains the majority of the current copies of f, or contains exactly one half of the current copies of f and one of these copies is higher, in the linear order of f, than all other current copies of f in other partitions.

3. Our Approach and the Algorithm

In this section, we present an algorithm that allows arbitrary merges while mutual consistency is still maintained. The data structures our algorithm assumes are a connection vector C at each site, and the five data structures S, L, X, V, and M introduced in last section associated with each copy of a file.

The algorithm consists of four major procedures, ISMAJORITY, PARTITION, RESOLVE, and MERGE. It enforces that update access to a file is allowed only in the majority partition of that file. Procedure ISMAJORITY determines if a copy of a file is in the majority partition of that file, consulting only local state information S, L, X, V and M associated with that copy. The

function of procedures PARTITION, RESOLVE and MERGE is to modify the local state information of the files whenever a communication link failure/repair is detected so that ISMAJORITY, when invoked can use this updated information to make correct decisions. The procedure PARTITION is invoked whenever a site detects a partitioning. It changes the version vectors of the local copies to reflect the occurence of this partitioning. Procedure MERGE, which is the heart of our algorithm, reconciles two or more partitions into one whenever the communication link between these partitions is repaired. It can be initiated at any site. It calls Procedure RESOLVE to resolve the X's, V's, and M's of each file, propagates the missed updates to each copy, and modifies the marker vectors to keep the state information consistent.

To access a file f, a site consults the local replication vector S associated with f to see if f is replicated at that site. If it is not, a remote access has to be performed. A site that has a copy of f and is currently connected with the local site (by consulting S and C, the connection vector) is chosen to perform the access. Any remote access mechanism can be used here. If the local site contains a copy of f, then ISMAJORITY is invoked to check if this site is in the majority partition of f. If ISMAJORITY returns yes and the access is a read, the local copy is fetched. If ISMAJORITY returns yes and the access is a write, then all copies of f in this majority partition are updated (by using a concurrency control protocol). When a copy of a file is updated, its version number X is incremented by 1. If ISMAJORITY returns no, the update access is rejected. However, the read-only acesses are allowed to proceed on all sites. Procedure ISMAJORITY is given in Fig. 2.

When a site detects a network partitioning, it calls Procedure PARTITION to modify the version vectors of those local copies of replicated files that are affected by this partitioning. Suppose k files $f_1, f_2, \ldots, f_k$ are replicated at this site (without loss of generality, call it site A). Fig. 3

Procedure ISMAJORITY (assume invoked at site A)

Input : X - version number of local copy of *f*.
V - version vector of local copy of *f*.
S - replication vector of *f*.
M - marker vector of local copy of *f*.
L - linear order vector of *f*.

Output: "yes" if A is in the majority partition of *f*.
"no" otherwise.

Method: First, compute E, the largest element of V, which is the value of the version number X at the time when the previous partitioning occurred. If the current X is greater than E, then A is obviously in *f*'s majority partition because this copy has been updated since last partitioning, so return "yes". Else, it must be the case that X = E. In this case, we need to compute Set1, the set of unmarked sites that are in the same partition as A, and Set2, the set of unmarked sites that were most recently separated from A (Note that we consider only unmarked sites when computing Set1 and Set2). If the size of Set1 is greater than that of Set2, then the partition that site A is in contains a majority copies of current copies of *f*, therefore it constitutes a majority partition of *f*, so return "yes". If both Set1 and Set2 have the same size, and there is a site in Set1 that is higher in *f*'s linear order than all sites in Set2, then the partition that site A is in is the majority partition of *f*, and "yes" is returned. In all other cases, the partition is not the majority partition of *f*, and "no" is returned.

```
function ISMAJORITY(X,V,S,M,L) : boolean ;
  begin
    E ← MAX{V[i]| V[i] ∈ V};
    if X > E then return "yes" fi;
    /* otherwise, X = E */
    Set1 ← {S[i]| V[i] = 0 and M[i] = F};
    Set2 ← {S[i]| V[i] = E and M[i] = F};
    if| Set1| >| Set2| then return "yes" fi;
    if| Set1| =| Set2| and
      there exists a site S[i] in Set1 such that S[i] > S[j] for all S[j] in Set2
          /* by consulting L */
      then return "yes";
    fi;
    return "no";
  end.
```

Fig. 2. Definition of Procedure ISMAJORITY.

gives the definition of Procedure PARTITION.

When we merge two or more partitions into a single partition, we need to *resolve* and update the version numbers, the version vectors, and the marker vectors of these copies. Assume

Procedure PARTITION (assume invoked at site A)

Input : C - The connection vector.

$X_1, X_2, ..., X_k$ - version numbers of the k copies of replicated files $f_1 f_2, ... f_k$.

$V_1, V_2, ..., V_k$ - version vectors of $f_1 f_2, ... f_k$.

$S_1, S_2, ..., S_k$ - replication vectors of $f_1 f_2, ... f_k$.

Output: $V_1, V_2, ..., V_k$ - the new version vectors of $f_1 f_2, ... f_k$.

Method: For each V_i, $1 \le i \le k$, for each $V_i[j] \in V_i$, do the following : if $S_i[j]$, the site associated with $V_i[j]$, is still connected with A, or is separated from A before this partitioning (value $V_i[j] \neq 0$), then do nothing; else it must be the case that $S_i[j]$ is separated from A due to this partitioning, so we set $V_i[j]$ to equal the current value of version number X_i.

```
procedure PARTITION(C; X1,X2,...,Xk; S1,S2,...,Sk; var V1,V2,...,Vk);
  begin
    for i = 1 to k do
      for j = 1 to | Vi | do
        if Vi[j] == 0 and
            Si[j] is now separated from A (by consulting connection vector C)
          then Vi[j] ← Xi;
        fi
      od
    od
  end.
```

Fig. 3. Definition of Procedure PARTITION.

k partitions $P_1, P_2, ..., P_k$ are to be merged to form a new partition P. The version number, the version vector, and the marker vector of P_i are X_i, V_i, and M_i, respectively. Procedure RESOLVE is defined in Fig. 4. The example 2 illustrates how the procedure RESOLVE works.

Example 2. Consider a file f with S = <A,B,C,D>. Suppose we want to merge partition AB and partition C to form a new partition ABC, and the version vectors of f in these two partitions at this point are <0,0,8,10> and <8,8,0,8>, respectively. Resolving these two version vectors gives a new version vector V = <0,0,0,10>. Sites A, B, and C are in the new partition ABC. So their corresponding values in V are 0. Site D is not in partition ABC, therefore its value in V is set to equal 10, the maximum of 10 and 8.

Procedure RESOLVE (invoked at the site that initiates the merge)

Input : $X_1, X_2, ..., X_k$ - version numbers to resolve.
$V_1, V_2, ..., V_k$ - version vectors to resolve.
$M_1, M_2, ..., M_k$ - marker vector to resolve.

Output: X - the new version number.
V - the new version vector.
M - the new marker vector.

Method: The new version number X of P is set to equal the maximum of all X_i's, i = 1,2,...,k. The new version vector V can be constructed as follows. For each V[j] ∈ V, do the following : if S[j] ∈ P, set V[j] = 0; otherwise, set V[j] to equal the maximum of all V_i[j]'s, i = 1,2,...,k. The new marker vector M is formed by ORing the M_i's componentwise.

```
procedure RESOLVE(X_1,...,X_k; V_1,...,V_k; M_1,..., M_k; var X; V; M);
  begin
    X ← MAX{X_i | i = 1,2,...,k};
    for j = 1 to | V | do
      if MIN{ V_i[j] | i = 1,2,...,k} == 0          /* site S[j] is in partition P */
        then V[j] ← 0
        else V[j] ← MAX{ V_i[j] | i = 1,2,...,k}
      fi
    od
    for j = 1 to | M | do
      M[j] ← OR { M_i[j] | i = 1,2,...,k}
    od
  end.
```

Fig. 4. Definition of Procedure RESOLVE.

In addition to resolving the X's, V's and M's when merging two or more partitions, we should make all copies of each file *f* in these partitions identical to the latest version of *f* in these partitions (some ideas on how to perform this task are discussed in [4]). If the resulting partition is not the majority partition of *f*, we mark those copies that had a version number less than the resolved version number so that they will not be counted as current copies when invoking ISMAJORITY. If the resulting partition is a new majority partition of *f*, we unmark all copies of *f* in this partition because they are now current. Finally, the old X's, the old V's, and the old M's of these copies are replaced by the resolved and modified new values. Procedure MERGE in Fig. 5

performs all merge operations mentioned above.

Procedure MERGE (invoked at the site that initiates a merge)

Input : $P_1,P_2,...,P_k$ - partitions to be merged (each P_i is a set of site names).

Output: P - a partition which is the merger of $P_1,P_2,...,P_k$.

Method :For each replicated file f, do the following: first, request a site in each partition P_i to send the X_i, V_i, and M_i of its copy of f. After having received all these X_i's, V_i's, and M_i's, invoke RESOLVE to resolve them. Next, make all copies in P identical to version X. If P is now the majority partition, then unmark all copies in P. If P is not the majority partition, mark those copies in P that had a version number less than the new version number X. Last, broadcast the new X, V, and M to all the sites in P that have a copy of f.

```
procedure MERGE(P_1,P_2,...,P_k)
  begin
    for each replicated file f do
      for i ← 1 to k do
        if there is a site s ∈ P_i that contains a copy of f
          then request s to send the X, V, and M of that copy;
               receive and store them in X_i, V_i, and M_i
          else /* No site in P_i has a copy of f. Give X_i, V_i, M_i dummy values */
               X_i ← 0; V_i ← <0,0,...,0>; M_i ← <F,F,...,F>
        fi
      od;
      RESOLVE(X_1,X_2,...,X_k, V_1,V_2,...,V_k, M_1,M_2,...,M_k, X, V, M);
      make all copies of f in P consistent with version X;
      E ← MAX{v| v ∈ V}; /* latest version in other partitions when separated from P */
      Set1 ← {S[i]| V[i] = 0 and M[i] = F and version number of f at S[i] = E};
      Set2 ← {S[i]| V[i] = E and M[i] = F};
      if one of the k partitions P_1,P_2,...,P_k is the majority partition of f or
        |Set1| > |Set2| or (|Set1| = |Set2| and
          there exists a site S[i] in Set1 such that S[i] > S[j] for all S[j] in Set2)
        then          /* the new P is the majority partition of f */
          for each V[i] ∈ V where V[i] = 0 do
            M[i] ← F;                    /* unmark the copy at S[i] */
          od
        else    /* P is not the majority partition. Mark off those non-current copies */
          for each V[i] ∈ V do
            if V[i] = 0 and version number of f at S[i] < X then M[i] ← T fi;
          od
      fi;
      broadcast X, V, and M to each site in P that has a copy of f;
    od;
  end.
```

Fig. 5. Definition of Procedure MERGE.

We now present example 3 to show how our algorithm works.

Example 3. In this example, we trace the partition history of file f depicted in Fig. 1. Assume the linear order of f is such that B > A > C (let S = <A,B,C> and L = <2,3,1>). Suppose that after f is created, two updates are successfully performed on f. The current state of f is as follows :

A,B,C
X = 2
V = <0,0,0>
M = <F,F,F>

Since all copies in a partition have the same version number and the same version vector, in order to simplify the notation, we have listed only one X, one V and one M for all copies of f in a partition. Suppose at this instance, a network partitioning separates A from B and C. After all three sites execute the procedure PARTITION, we have the following state.

A	B,C
X = 2	X = 2
V = <0,2,2>	V = <2,0,0>
M = <F,F,F>	M = <F,F,F>

At this moment, if site A invokes ISMAJORITY, then Set1 will be set to {A} and Set2 will be set to {B,C}. If B or C invokes ISMAJORITY, it will find that Set1 = {B,C} and Set2 = {A}. Thus partition BC constitutes a majority partition of f. Assume three more updates are performed on f before B is isolated from C. We have the following state.

A	B	C
X = 2	X = 5	X = 5
V = <0,2,2>	V = <2,0,5>	V = <2,5,0>
M = <F,F,F>	M = <F,F,F>	M = <F,F,F>

Both partition B and partition C contain one half of the current copies ($|Set1| = |Set2| = 1$ if ISMAJORITY is invoked by B or C). Since we assume B > C in the linear order of *f*, partition B is now the majority partition. Here note that the majority partition of a file does not have to contain a majority copies of that file.

Now suppose that *f* is updated three more times, and then the partition A and the partition C are merged into a single partition AC. We have the following situation.

A,C	B
X = 5	X = 8
V = <0,5,0>	V = <2,0,5>
M = <T,F,F>	M = <F,F,F>

Note that A's marker in partition AC has been changed to T (marked). If A or C later invokes ISMAJORITY, it will find that Set1 = {C} and Set2 = {B}. Site A will not be included in Set1 because it is marked. Since B > C, partition AC is not the majority partition of *f*. Also note that the copy at site A has been updated from version 2 to version 5. But AC is not the majority partition of *f*. So why should we update this copy? We give our reason for doing such an update, based on the notion of view serializability [4], for increasing the availability for read access The correctness criterion based on view serializability requires the following conditions:

1. The updates do not creat a cycle in the conflict graph [14].

and

2. The read-only accesses considered one at a time in the conflict graph do not creat a cycle.

Consider the transaction processing in a bank. There are many user transactions that would just like to read the database values. In other words, the users like to get a view of a correct database state. When the network is partitioned, even if the most up-to-date view is not available, an earlier version may be acceptable. For example, if one calls a bank to find the balance in the account, the following answer may be acceptable: your balance is this amount, however some checks may not have been processed. The cause of unprocessed checks may be the delays due to a failure of some part of the system or network partition. Of course, when the actually goes to withdraw the funds, the transaction becomes an update and could be rejected. Therefore, it is better to keep the balance as up-to-date as possible by performing the update mentioned above. As discussed in [4], the availability during network partitions can be increased using the correctness criterion called *"view serializability"* for concurrency control where read-only transactions are treated differently than the update transactions.

4. Comparison with Previous Algorithms

We have presented an algorithm that provides a higher degree of availability than the algorithms proposed in [7,11,12,13]. The major difference between our algorithm and the previous ideas is that our algorithm supports partial replication (by introducing the replication vector) and merges the partitions as soon as the communication link among them is repaired (by introducing the marker vector M). The four previous algorithms do not always allow the merge of an arbitrary subset of the partitions in the system. To see what might happen if arbitrary merges are allowed in their algorithms(also see examples given in [7,11,12,13]), let's consider once again the

last partition state of Example 3. The marker vector is absent in their algorithms.

A,C	B
X = 5	X = 8
V = <0,5,0>	V = <2,0,5>

Both A and C can determine that they belong to the majority partition (since Set1 = {A,C} and Set2 = {B}). So can B (since X > E). Thus two majority partitions exist for the same file at the same time. Therefore, in this case, their algorithms simply do not allow partition A to merge with partition C to avoid such inconsistency.

If *f* is the only replicated file in the DDB, this restriction might be acceptable because even if we allow partition A and partition C to merge, the new partition should not be allowed to update *f* anyway. As far as availability is concerned, we lose nothing. But availability will be lost if the DDB contains more than one replicated file. Let's suppose that another file *g* is also replicated at these three sites, but site C is higher than site B in *g*'s linear order. Partition C will be the majority partition of *g* before the merge is attempted. If partition A is allowed to merge with partition C, *g* will be accessible again at site A. Since *f* does not allow partition A and partition C to merge, we see that the availability of *g* is affected by the existence of *f*.

When we consider a DDB with many replicated files, such restriction might become quite severe and unacceptable, as Example 4 illustrates.

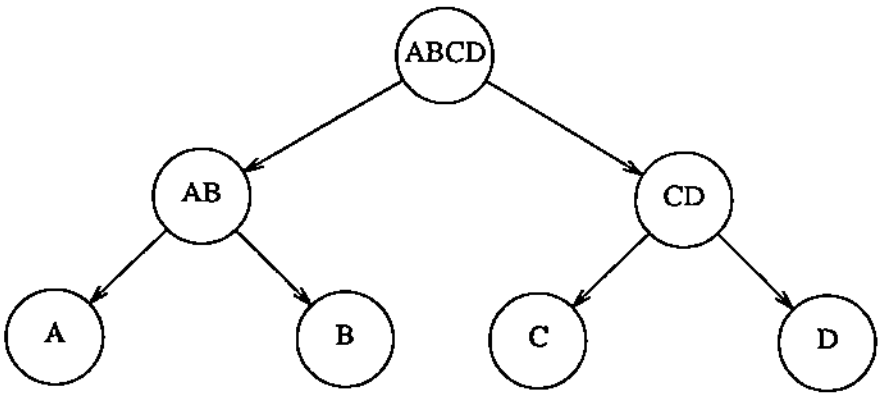

Fig. 6. Partition graph of file f_i, i = A,B,C,or D.

Example 4. Suppose that four files f_A, f_B, f_C, and f_D are replicated at four sites A, B, C, and D. The linear order of file f_i, where i = A, B, C, or D, is such that site i has the highest order. Consider the partition history in Fig. 6. (All four files have the same partition graph in this case.) Each current partition is the majority partition of one of the files. Out of 11 possible merge combinations, only three of them are allowed. They are, A merges with B, C merges with D, or A, B, C, and D merge at one step. Any other attempt will fail. Note that the procedures for any updates performed after various partitions are similar to those in example 3.

By using the marker vector M to distinguish the current copies, our algorithm is able to perform arbitrary merges while still maintaining mutual consistency, thus providing a higher degree of availability than the previous four algorithms. Another drawback of [12,13] is that they do not allow a site to update its copy if this site does not belong to that file's majority partition. This restriction makes the alternative (allowing read-only accesses in non-majority partition mentioned at the end of last section) less appealing.

5. Discussion

In our algorithm and the previous four algorithms, high availability is achieved by avoiding the loss of the majority partition. However, in some cases, the loss of majority partition is unavoidable. We say that a partitioning is simple if it splits a partition into exactly two partitions. Otherwise, it is called a multiple partitioning. A multiple partitioning might cause a file inaccessible everywhere, as the partition graph in Fig. 7 illustrates. In the class of dynamic voting algorithms, no solution exists for such problem since when a site is isolated from the other two sites, it has no way to determine if the other two sites are still connected or separated. So, the worst assumption that the other two sites are still connected must be made. No partition will claim itself the majority partition.

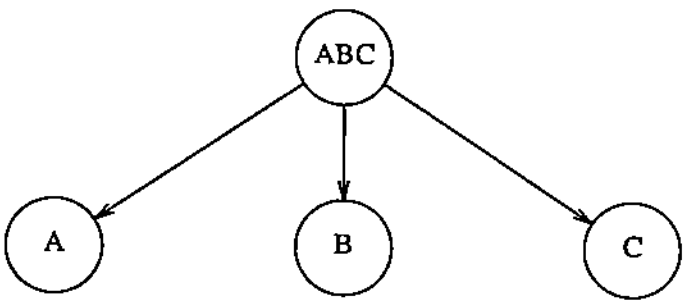

Fig. 7. A multiple partitioning that causes *f* inaccessible.

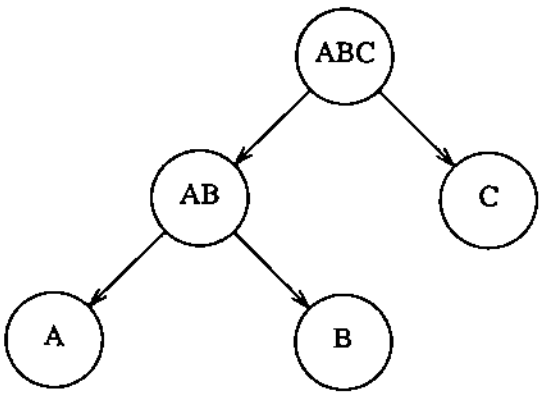

Fig. 8. Two simple partitionings that cause f inaccessible.

Fig. 8 gives another example which shows that the loss of the majority partition might be caused by two consecutive simple partitionings if file f is not updated during the period between the two partitionings. Note that if f is rarely updated, it might become inaccessible everywhere even if the second partitioning occurs long after the first one. We modify procedure PARTITION as follows to avoid such loss of majority partition.

In the procedure PARTITION, after the version vector V_i for each file f_i is updated, a call to ISMAJORITY is immediately made to check if this site is in f_i's majority partition. If it is, the version number X_i is incremented by 1. A disadvantage of this scheme is that additional dummy updates might result. However, another flag can be used to indicate such an increase in version number and avoid dummy updates. With this modification, our algorithm guarantees that there is always exactly one majority partition for each file at any given time if simple partitioning is the only type of partitioning in the system.

A reader might notice that, in our algorithm, if the majority partition is lost (due to the occurence of a multiple partitioning), the only way to reconstruct it back is by merging the sites from the latest majority partition, that is, the one before the majority partition is lost. Merges of other sites are allowed but cannot result in a new majority partition. In [5] we have suggested an alternative to allow the reconstruction of the majority partition as quickly as possible, using the notions of tie and threshold.

We have tried to achieve high availability by avoiding the loss of the majority partition. As mentioned in the introduction, high availability can also be achieved by following the second direction, that is, to keep the size of the majority partition above a threshold to prevent the majority partition from getting too small (such an idea is examined in depth in [5]). To see which method provides a higher availability than the other for a given partition history, we need to give the availability of a file a quantitative definition. Assume that a file f is replicated at n sites, and each copy of f has the same weight, that is, the probability of each copy of f being accessed is equally likely. We define the availability of f, denoted by A_f, as

$$A_f = \frac{\sum_{i=1}^{i=n} \left(\textit{length of time during which } f \textit{ is available at site } s_i \right)}{n \times \textit{duration of the history}}$$

As an example, for the history in Fig. 1, assuming the history starts at time 0, the first partitioning (ABCDE breaks into ABC and DE) occurs at time 2, the second partitioning (ABC breaks into AB and C) occurs at time 3, the first merging occurs at time 4, and the second merging (AB merges with CDE) occurs at time 20. If the first method (ABC and AB are majority) is used, the availability of f will be

$$A_f^1 = \frac{20\ (for\ A) + 20\ (for\ B) + 3\ (for\ C) + 2\ (for\ D) + 2\ (for\ E)}{5 \times 20} = \frac{47}{100}$$

Similarly, the availability for second method (ABC and CDE are majority) will be

$$A_f^2 = \frac{3 + 3 + 19 + 18 + 18}{5 \times 20} = \frac{61}{100}$$

Therefore, the second method gives a higher availability than the first one for this specific history. But if the first merging occurs at time 19 instead of at time 4, the availability of method 1 will still be the same, while the availability of method 2 will be

$$A_f^2 = \frac{3 + 3 + 4 + 3 + 3}{100} = \frac{16}{100}$$

The first method works better in this case.

We are investigating a design of an efficient scheme that can adapt to changing environment (e.g. configuration of the network) by selecting appropriate algorithms to allow high availability. The experiments in RAID system [ ] are attempting to answer the performance and feasibility questions.

References

[1] D. Barbara, and H. Garcia-Molina, "How expensive is data replication : An example," TR 286, Dept. of Elec. Eng. and Comp. Sc., Princeton University, June, 1981.

[2] D. Barbara, H. Garcia-Molina, and A. Spauster, "Protocols for dynamic vote reassignment," TR 037-86, Comp. Sc. Dept., Princeton University, May, 1986

[3] P. A. Bernstein, and N. Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys*, Vol. 13, No. 2, pp. 185-221, June, 1981.

[4] B. Bhargava, "Transaction processing and consistency control of replicated copies during failures," *Journal of Management Information Systems*, Oct., 1987

[5] B. Bhargava, "Tie declaration and a new majority establishment during network reconfiguration," working paper, Computer Science Department, Purdue University, May, 1987.

[6] B. Bhargava, J. Riedl, and A. Royappa, "The RAID Distributed Database System," TR 691, Computer Science Department, Purdue University, August, 1987.

[7] D. Davcev, and W. Burkhard, "Consistency and recovery control for replicated files," *Proc. of the 10th ACM Symp. on Operating Systems Principles*, pp. 87-96, Orcas Island, Wash., Dec., 1985.

[8] S. B. Davidson, H. Garcia-Molina, and D. Skeen, "Consistency in partitioned networks," *ACM Computing Surveys*, Vol. 17, No. 3, pp. 341-370, Sept., 1985.

[9] C. Ellis, "A robust algorithm for updating duplicated databases," *Proc. of the Second Berkeley Workshop on Distributed Management of Data and Computer Networks*, pp. 146-158, Berkeley, Calif., May, 1977.

[10] H. Garcia-Molina, "Performance of update algorithm for replicated data in a distributed database," Ph.D. Dissertation, Stanford University, 1979.

[11] S. Jajodia, "Managing replicated files in partitioned distributed database systems," *Third IEEE Conf. on Data Engineering*, Los Angeles, Feb. 1987.

[12] S. Jajodia, and D. Mutchler, "Dynamic voting,"*ACM SIGMOD-87*, San Francisco, May, 1987.

[13] S. Jajodia, and D. Mutchler, "Enhencements to the voting algorithm," To appear in *VLDB*, Brighton, UK, Sept., 1987

[14] C. H. Papadimitriou, "The serializability of concurrent database updates," *JACM*, Vol. 26, No. 4, pp. 631-653, Oct., 1979.

[15] D. Parker, et al., "Detection of mutual inconsistencies in distributed systems," *IEEE Trans. Softw. Eng.*, Vol. SE-9, No. 3, pp. 240-247, May, 1983.

[16] J. A. Stankovic, K. Ramamritham, and W. H. Kohler, "A review of current research and critical issues in distributed system software," in Bhargava, B.(editor) *Concurrency and Reliability in Distributed Systems*, Van Nostrand & Reinhold, 1987.

[17] M. Stonebraker, "Concurrency control and consistency of multiple copies of data in distributed INGRES," *IEEE Trans. Softw. Eng.*, Vol. SE-5, No. 3, pp. 180-194, May, 1979.

[18] R. Thomas, "A majority consensus approach to concurrency control," *ACM Transaction on Database Systems*, Vol. 4, No. 2, pp. 180-209, 4, 1979.

[19] M. Yannakakis, "Serializability by locking," *JACM*, vol. 31, pp. 227-244, April, 1978.